Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R

Michael Hahsler
Southern Methodist University

Matthew Bolaños
Microsoft Corporation

John Forrest
Microsoft Corporation

Abstract

In recent years, data streams have become an increasingly important area of research for the computer science, database and statistics communities. Data streams are ordered and potentially unbounded sequences of data points created by a typically non-stationary data generating process. Common data mining tasks associated with data streams include clustering, classification and frequent pattern mining. New algorithms for these types of data are proposed regularly and it is important to evaluate them thoroughly under standardized conditions.

In this paper we introduce **stream**, a research tool that includes modeling and simulating data streams as well as an extensible framework for implementing, interfacing and experimenting with algorithms for various data stream mining tasks. The main advantage of **stream** is that it seamlessly integrates with the large existing infrastructure provided by R. In addition to data handling, plotting and easy scripting capabilities, R also provides many existing algorithms and enables users to interface code written in many programming languages popular among data mining researchers (e.g., C/C++, Java and Python). In this paper we describe the architecture of **stream** and focus on its use for data stream clustering research. **stream** was implemented with extensibility in mind and will be extended in the future to cover additional data stream mining tasks like classification and frequent pattern mining.

Keywords: data streams, data mining, clustering.

Note: A previous version of this manuscript was published in the Journal of Statistical Software (Hahsler, Bolaños, and Forrest 2017a).

1. Introduction

Typical statistical and data mining methods (e.g., clustering, regression, classification and frequent pattern mining) work with "static" data sets, meaning that the complete data set is available as a whole to perform all necessary computations. Well known methods like k-means clustering, linear regression, decision tree induction and the APRIORI algorithm to find frequent itemsets scan the complete data set repeatedly to produce their results (Hastie, Tibshirani, and Friedman 2001). However, in recent years more and more applications need to

work with data which are not static, but are the result of a continuous data generating process which is likely to evolve over time. Some examples are web click-stream data, computer network monitoring data, telecommunication connection data, readings from sensor nets and stock quotes. These types of data are called data streams and dealing with data streams has become an increasingly important area of research (Babcock, Babu, Datar, Motwani, and Widom 2002; Gaber, Zaslavsky, and Krishnaswamy 2005; Aggarwal 2007). Early on, the statistics community also recognized the importance of the emerging field of statistical analysis of massive data streams (see Keller-McNulty (2004)).

A data stream can be formalized as an ordered sequence of data points

$$Y = \langle \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots \rangle,$$

where the index reflects the order (either by explicit time stamps or just by an integer reflecting order). The data points themselves are often simple vectors in multidimensional space, but can also contains nominal/ordinal variables, complex information (e.g., graphs) or unstructured information (e.g., text). The characteristic of continually arriving data points introduces an important property of data streams which also poses the greatest challenge: the size of a data stream is potentially unbounded. This leads to the following requirements for data stream processing algorithms:

- Bounded storage: The algorithm can only store a very limited amount of data to summarize the data stream.
- Single pass: The incoming data points cannot be permanently stored and need to be processed at once in the arriving order.
- Real-time: The algorithm has to process data points on average at least as fast as the data is arriving.
- Concept drift: The algorithm has to be able to deal with a data generating process which evolves over time (e.g., distributions change or new structure in the data appears).

Most existing algorithms designed for static data are not able to satisfy all these requirements and thus are only usable if techniques like sampling or time windows are used to extract small, quasi-static subsets. While these approaches are important, new algorithms to deal with the special challenges posed by data streams are needed and have been introduced over the last decade.

Even though R represents an ideal platform to develop and test prototypes for data stream mining algorithms, R currently does only have very limited infrastructure for data streams. The following are some packages available from the Comprehensive R Archive Network[1] related to streams:

Data sources: Random numbers are typically created as streams (see e.g., **rstream** (Leydold 2015) and **rlecuyer** (Sevcikova and Rossini 2012)). Financial data can be obtained via packages like **quantmod** (Ryan 2016). Intra-day price and trading volume can be considered a data stream. For Twitter, a popular micro-blogging service, packages like **streamR** (Barbera 2014) and **twitteR** (Gentry 2015) provide interfaces to retrieve life Twitter feeds.

[1] https://CRAN.R-project.org/

Statistical models: Several packages provide algorithms for iteratively updating statistical models, typically to deal with very large data. For example, **factas** (Bar 2014) implements iterative versions of correspondence analysis, PCA, canonical correlation analysis and canonical discriminant analysis. For clustering, **birch** (Charest, Harrington, and Salibian-Barrera 2012) implements BIRCH, a clustering algorithm for very large data sets. The algorithm maintains a clustering feature tree which can be updated in an iterative fashion. Although BIRCH was not developed as a data stream clustering algorithm, it first introduced some characteristics needed for efficiently handling data streams. Unfortunately, the **birch** package is no longer maintained and was removed recently from CRAN. **rEMM** (Hahsler and Dunham 2015) implements a stand-alone version of a pure data stream clustering algorithm enhanced with a methodology to model a data stream's temporal structure. Very recently **RMOA** (Wijffels 2014) was introduced. The package interfaces data stream classification algorithms from the MOA framework (see existing tools discussed in Section 2.4), however, the package focuses not on data streams but on static data sets that do not fit into main memory.

Distributed computing frameworks: With the development of Hadoop[2], distributed computing frameworks to solve large scale computational problems have become very popular. **HadoopStreaming** (Rosenberg 2012) is available to use map and reduce scripts written in R within the Java-based Hadoop framework. However, contrary the word streaming in its name, **HadoopStreaming** does not refer to data streams. As Hadoop itself, **HadoopStreaming** is used for batch processing and streaming in the name refers only to the internal usage of pipelines for "streaming" the input and output between the Hadoop framework and the used R scripts. A distributed framework for realtime computation is Storm. Storm builds on the idea of constructing a computing topology by connecting spouts (data stream sources) with a set of bolts (computational units). **RStorm** (Kaptein 2013) provides an environment to prototype bolts in R. Spouts are represented as data frames. Bolts developed in **RStorm** can currently not directly be used in Storm, but this is planned for the future (Kaptein 2014).

Even in the stream-related packages discussed above, data is still represented by data frames or matrices which is suitable for static data but not ideal to represent streams.

In this paper we introduce the package **stream** (Hahsler, Bolaños, and Forrest 2017b) which provides a framework to represent and process data streams and use them to develop, test and compare data stream algorithms in R. We include an initial set of data stream generators and data stream clustering algorithms in this package with the hope that other researchers will use **stream** to develop, study and improve their own algorithms.

The paper is organized as follows. We briefly review data stream mining in Section 2. In Section 3 we cover the basic design principles of the **stream** framework. Sections 4, 5 and 6 introduce details about creating data stream sources, performing data stream mining tasks, and evaluating data stream clustering algorithms, respectively. Each of the three sections include example code. Section 7 we provides comprehensive examples performing an experimental comparison of several data stream clustering algorithms and clustering a large, high-dimensional data set. Section 8 concludes the paper.

[2] https://hadoop.apache.org/

2. Data stream mining

Due to advances in data gathering techniques, it is often the case that data is no longer viewed as a static collection, but rather as a potentially very large dynamic set, or stream, of incoming data points. The most common data stream mining tasks are clustering, classification and frequent pattern mining (Aggarwal 2007; Gama 2010). In this section we will give a brief introduction to these data stream mining tasks. We will focus on clustering, since this is also the current focus of package **stream**.

2.1. Data stream clustering

Clustering, the assignment of data points to (typically k) groups such that points within each group are more similar to each other than to points in different groups, is a very basic unsupervised data mining task. For static data sets, methods like k-means, k-medoids, hierarchical clustering and density-based methods have been developed among others (Jain, Murty, and Flynn 1999). Many of these methods are available in tools like R, however, the standard algorithms need access to all data points and typically iterate over the data multiple times. This requirement makes these algorithms unsuitable for large data streams and led to the development of data stream clustering algorithms.

Over the last 10 years many algorithms for clustering data streams have been proposed (see Silva, Faria, Barros, Hruschka, Carvalho, and Gama (2013) for a current survey). Most data stream clustering algorithms deal with the problems of unbounded stream size, and the requirements for real-time processing in a single pass by using the following two-stage online/offline approach introduced by Aggarwal, Han, Wang, and Yu (2003).

1. Online: Summarize the data using a set of k' micro-clusters organized in a space efficient data structure which also enables fast look-up. Micro-clusters were introduced for CluStream by Aggarwal et al. (2003) based on the idea of cluster features developed for clustering large data sets with the BIRCH algorithm (Zhang, Ramakrishnan, and Livny 1996). Micro-clusters are representatives for sets of similar data points and are created using a single pass over the data (typically in real time when the data stream arrives). Micro-clusters are often represented by cluster centers and additional statistics such as weight (local density) and dispersion (variance). Each new data point is assigned to its closest (in terms of a similarity function) micro-cluster. Some algorithms use a grid instead and micro-clusters are represented by non-empty grid cells (e.g., D-Stream by Tu and Chen (2009) or MR-Stream by Wan, Ng, Dang, Yu, and Zhang (2009)). If a new data point cannot be assigned to an existing micro-cluster, a new micro-cluster is created. The algorithm might also perform some housekeeping (merging or deleting micro-clusters) to keep the number of micro-clusters at a manageable size or to remove information outdated due to a change in the stream's data generating process.

2. Offline: When the user or the application requires a clustering, the k' micro-clusters are reclustered into $k \ll k'$ final clusters sometimes referred to as macro-clusters. Since the offline part is usually not regarded time critical, most researchers use a conventional clustering algorithm where micro-cluster centers are regarded as pseudo-points. Typical reclustering methods involve k-means or clustering based on the concept of reachability introduced by DBSCAN (Ester, Kriegel, Sander, and Xu 1996). The algorithms are often modified to take also the weight of micro-clusters into account.

The most popular approach to adapt to concept drift (changes of the data generating process over time) is to use the exponential fading strategy introduced first for DenStream by Cao, Ester, Qian, and Zhou (2006). Micro-cluster weights are faded in every time step by a factor of $2^{-\lambda}$, where $\lambda > 0$ is a user-specified fading factor. This way, new data points have more impact on the clustering and the influence of older points gradually disappears. Alternative models use sliding or landmark windows. Details of these methods as well as other data stream clustering algorithms are discussed in the survey by Silva et al. (2013).

2.2. Outlier detection

The outlier detection in data streams is a popular task, often used for risk management, e.g., fraud and intrusion detection. From the end-user point of view, outliers are important and meaningful data points that are standing out from the usual populations (clusters) that can be found in data streams (Silva et al. 2013). This differentiation between clusters and outliers can be statistically or density-based.

We build special outlier detectors to detect them in big data streams. Detecting small statistical (or density) differences between clusters and outliers is the key feature of a good outlier detector. Outlier detectors that can detect outlier smaller statistical or density variations are better. This leads us to the outlier detector breaking point, which is the smaller statistical or density variation for which the outlier detector still can detect some outlier.

End-user interest in outliers requires that detected outliers get reported back to the user in a limited time frame, which is directly correlated with data stream velocity.

Such a time requirement requires a more fine-grained approach than the previously described two-stage approach. For outlier detectors, such as Continuous Outlier Detection (COD), Micro-cluster Continuous Outlier Detection (MCOD) (Kontaki, Gounaris, Papadopoulos, Tsichlas, and Manolopoulos (2016)), and Statistical Hierarchical Clustering (SHC) (Krleža, Vrdoljak, and Brčić (2020)), this means processing each data point retrieved from the input data stream in two steps. In the first step, outlier detectors are trying to classify the input data point. If the input data point does not belong to any known cluster (population), the outlier detector must decide whether the data point represents a new outlier.

The evolving nature of the input data stream causes outliers to become inliers, which represents an issue while trying to assess the outlier detection correctness. In such cases, outlier detectors must have outlier tracking capabilities, which allows users to re-check each outlier individually and determine whether a previously detected outlier is still an outlier, or it evolved into an inlier in the meantime.

2.3. Other popular data stream mining tasks

Classification, learning a model in order to assign labels to new, unlabeled data points is a well studied supervised machine learning task. Methods include naive Bayes, k-nearest neighbors, classification trees, support vector machines, rule-based classifiers and many more (Hastie et al. 2001). However, as with clustering these algorithms need access to the complete training

data several times and thus are not suitable for data streams with constantly arriving new training data and concept drift.

Several classification methods suitable for data streams have been developed. Examples are Very Fast Decision Trees (VFDT) (Domingos and Hulten 2000) using Hoeffding trees, the time window-based Online Information Network (OLIN) (Last 2002) and On-demand Classification (Aggarwal, Han, Wang, and Yu 2004) based on micro-clusters found with the data-stream clustering algorithm CluStream (Aggarwal et al. 2003). For a detailed discussion of these and other methods we refer the reader to the survey by Gaber, Zaslavsky, and Krishnaswamy (2007).

Another common data stream mining task is frequent pattern mining. The aim of frequent pattern mining is to enumerate all frequently occurring patterns (e.g., itemsets, subsequences, subtrees, subgraphs) in large transaction data sets. Patterns are then used to summarize the data set and can provide insights into the data. Although finding all frequent patterns in large data sets is a computationally expensive task, many efficient algorithms have been developed for static data sets. A prime example is the APRIORI algorithm (Agrawal, Imielinski, and Swami 1993) for frequent itemsets. However, these algorithms use breath-first or depth-first search strategies which results in the need to pass over each transaction (i.e., data point) several times and thus makes them unusable for the case where transactions arrive and need to be processed in a streaming fashion. Algorithms for frequent pattern mining in streams are discussed in the surveys by Jin and Agrawal (2007), Cheng, Ke, and Ng (2008) and Vijayarani and Sathya (2012).

2.4. Existing tools

MOA[3] (short for Massive Online Analysis) is a framework implemented in Java for stream classification, regression and clustering (Bifet, Holmes, Kirkby, and Pfahringer 2010). It was the first experimental framework to provide easy access to multiple data stream mining algorithms, as well as to tools for generating data streams that can be used to measure and compare the performance of different algorithms. Like WEKA (Witten and Frank 2005), a popular collection of machine learning algorithms, MOA is also mainly developed by the University of Waikato and its graphical user interface (GUI) and workflow are similar to those of WEKA. Classification results are shown as text, while clustering results have a visualization component that shows both the evolution of the clustering (in two dimensions) and various performance metrics over time (Kranen, Kremer, Jansen, Seidl, Bifet, Holmes, and Pfahringer 2010).

SAMOA[4] (Scalable Advanced Massive Online Analysis) is a recently introduced tool for distributed stream mining with Storm or the Apache S4 distributed computing platform. Similar to MOA it is implemented in Java, and supports the basic data stream mining tasks of clustering, classification and frequent pattern mining. Some MOA clustering algorithms are interfaced in SAMOA. SAMOA currently does not provide a GUI.

Another distributed processing framework and streaming machine learning library is Jabatus[5]. It is implemented in C++ and supports classification, regression and clustering. For clustering it currently supports k-means and Gaussian Mixture Models (version 0.5.4).

[3] https://moa.cms.waikato.ac.nz/
[4] https://incubator.apache.org/projects/samoa.html
[5] http://jubat.us/en/

Commercial data stream mining platforms include IBM InfoSphere Streams and Microsoft StreamInsight (part of MS SQL Server). These platforms aim at building applications using existing data stream mining algorithms rather than developing and testing new algorithms.

MOA is currently the most complete framework for data stream clustering research and it is an important pioneer in experimenting with data stream algorithms. MOA's advantages are that it interfaces with WEKA, provides already a set of data stream classification and clustering algorithms and it has a clear Java interface to add new algorithms or use the existing algorithms in other applications.

A drawback of MOA and the other frameworks for R users is that for all but very simple experiments custom Java code has to be written. Also, using MOA's data stream mining algorithms together with the advanced capabilities of R to create artificial data and to analyze and visualize the results is currently very difficult and involves running code and copying data manually. The recently introduce R-package **RMOA** (Wijffels 2014) interfaces MOA's data stream classification algorithms, however, it focuses on processing large data sets that do not fit into main memory and not on data streams.

3. The stream framework

The **stream** framework provides an R-based alternative to MOA which seamlessly integrates with the extensive existing R infrastructure. Since R can interface code written in many different programming languages (e.g., C/C++, Java, Python), data stream mining algorithms in any of these languages can be easily integrated into **stream**. **stream** is based on several packages including **fpc** (Hennig 2014), **clue** (Hornik 2017), **cluster** (Maechler, Rousseeuw, Struyf, Hubert, and Hornik 2014), **clusterGeneration** (Qiu and Joe. 2015), **MASS** (Venables and Ripley 2002), **proxy** (Meyer and Buchta 2017), and others. The **stream** extension package **streamMOA** (Hahsler and Bolanos 2015) also interfaces the data stream clustering algorithms already available in MOA using the **rJava** package by Urbanek (2016).

We will start with a very short example to make the introduction of the framework and its components easier to follow. After loading **stream**, we create a simulated data stream with data points drawn from three random Gaussians in 2D space. Note that we set the random number generator seed every time when we create simulated data sets to get reproducible results.

```
R> library("stream")
R> set.seed(1000)
R> stream_orig <- DSD_Gaussians(k = 3, d = 2)
```

Next, we apply a filter to transform the stream. Here, we will scale the stream to z-scores. We use here a pipe, but `DSF_Scale()` can also be called with the input stream as its first argument.

```
R> stream <- stream_orig %>% DSF_Scale()
```

Now, we create an instance of the density-based data stream clustering algorithm D-Stream which uses grid cells as micro-clusters. We specify the grid cell size (`gridsize`) as .1 and

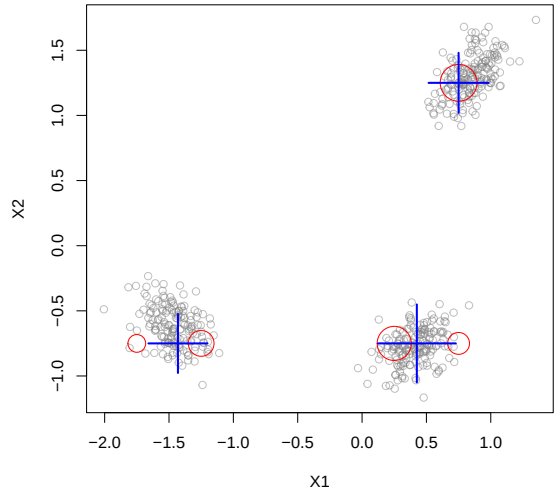


Figure 1: Data stream clustering result of D-Stream on a simple simulated data set with three random Gaussians. Micro-clusters are shown as circles and macro-clusters are shown as crosses (size represents weight).

require that the density of a grid cell (`Cm`) needs to be at least 1.2 times the average cell density to become a micro-cluster. Then we update the model with the next 500 data points from the stream.

```
R> dstream <- DSC_DStream(gridsize = .5, Cm = 1.2)
R> update(dstream, stream, n = 500)
```

Finally, we perform reclustering using k-means with three clusters and plot the resulting micro and macro clusters (see Figure 1).

```
R> km <- DSC_Kmeans(k = 3)
R> recluster(km, dstream)
R> plot(km, stream, type = "both")
```

As shown in this example, the **stream** framework consists of three main components:

1. Data stream data (DSD) simulates or connects to a data stream.
2. Data stream filter (DSF) one or more filters to transform the stream.
3. Data stream task (DST) performs a data stream mining task. In the example above, we performed twice a data stream clustering (DSC) task.

Figure 2 shows a high level view of the interaction of the components. We start by creating a DSD object and a DST object. Then the DST object starts receiving data form the DSD object. At any time, we can obtain the current results from the DST object. DSTs can implement any type of data stream mining task (e.g., classification or clustering).

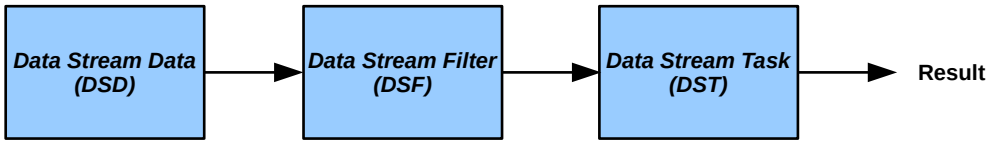


Figure 2: A high level view of the **stream** architecture.

Since stream mining is a relatively young field and many advances are expected in the near future, the object oriented framework in **stream** was developed with easy extensibility in mind. We are using the S3 class system (Chambers and Hastie 1992) throughout and, for performance reasons, the R-based algorithms are implemented using reference classes. The framework provides for each of the two core components a lightweight interface definition (i.e., an abstract class) which can be easily implemented to create new data stream types or to interface new data stream mining algorithms. Developers can also extend the infrastructure with new data mining tasks. Details for developers interested in extending **stream** can be found in the package's vignette and manual pages (Hahsler et al. 2017b). In the following we will concentrate on describing the aspects of the framework which are important to a users interested in dealing with data streams and performing data stream mining tasks in R.

4. Data stream data (DSD)

4.1. Introduction

The first step in the **stream** workflow is to select a data stream implemented as a data stream data (DSD) object. This object can be a management layer on top of a real data stream, a wrapper for data stored in memory or on disk, or a generator which simulates a data stream with know properties for controlled experiments. Figure 3 shows the relationship (inheritance hierarchy) of the DSD classes as a UML class diagram (Fowler 2003). All DSD classes extend the abstract base class `DSD`. There are currently two types of DSD implementations, classes which implement R-based data streams (`DSD_R`) and MOA-based stream generators (`DSD_MOA`) provided in **streamMOA**. Note that abstract classes define interfaces and only implement common functionality. Only implementation classes can be used to create objects (instances). This mechanism is not enforced by S3, but is implemented in **stream** by providing for all abstract classes constructor functions which create an error.

The package **stream** provides currently the following set of DSD implementations:

- Simulated streams with static structure.
  - `DSD_BarsAndGaussians` generates two uniformly filled rectangular and two Gaussians clusters with different density.
  - `DSD_Gaussians` generates randomly placed static clusters with random multivariate Gaussian distributions. Allows generating and marking outliers for outlier detectors.

  - `DSD_mlbenchData` provides streaming access to machine learning benchmark data sets found in the `mlbench` package (Leisch and Dimitriadou 2012).
  - `DSD_mlbenchGenerator` interfaces the generators for artificial data sets defined in the `mlbench` package.
  - `DSD_Target` generates a ball in circle data set.
  - `DSD_UniformNoise` generates uniform noise in a d-dimensional (hyper) cube.

- Simulated streams with concept drift.

  - `DSD_Benchmark`, a collection of simple benchmark problems including splitting and joining clusters, and changes in density or size. This collection is indented to grow into a comprehensive benchmark set used for algorithm comparison.
  - `DSD_MG`, a generator to specify complex data streams with concept drift. The shape as well as the behavior of each cluster over time (changes in position, density and dispersion) can be specified using keyframes (similar to keyframes in animation and film making) or by mathematical functions.
  - `DSD_RandomRBFGeneratorEvents` (**streamMOA**) generates streams using radial base functions with noise. Clusters move, merge and split.

- Connectors to real data and streams.

  - `DSD_Memory` provides a streaming interface to static, matrix-like data (e.g., a data frame, a matrix) in memory which represent a fixed portion of a data stream. Matrix-like objects also include large objects potentially stored on disk like `ffdf` from package **ff** (Adler, Gläser, Nenadic, Oehlschlägel, and Zucchini 2014) or `big.matrix` from package **bigmemory** (Kane, Emerson, and Weston 2013). Any matrix-like object which implements at least row subsetting with `"["` and `dim()` can be used. Using these, stream mining algorithms (e.g., clustering) can be performed on data that does not fit into main memory. In addition, `DSD_Memory` can directly create a static copy of a portion of another DSD object to be replayed in experiments several times.
  - `DSD_ReadCSV` reads data line by line in text format from a file or an open connection and makes it available in a streaming fashion. This way data that is larger than the available main memory can be processed. Connections can be used to read from real-time data streams.
  - `DSD_ReadDB` provides an interface to an open result set from a SQL query to a relational database. Any of the many database management systems with a **DBI** interface (R Special Interest Group on Databases 2014) can be used.

- In-flight stream operations.

  - `DSD_ScaleStream` can be used to standardize (centering and scaling) data in a data stream in-flight.

All DSD implementations share a simple interface consisting of the following two functions:

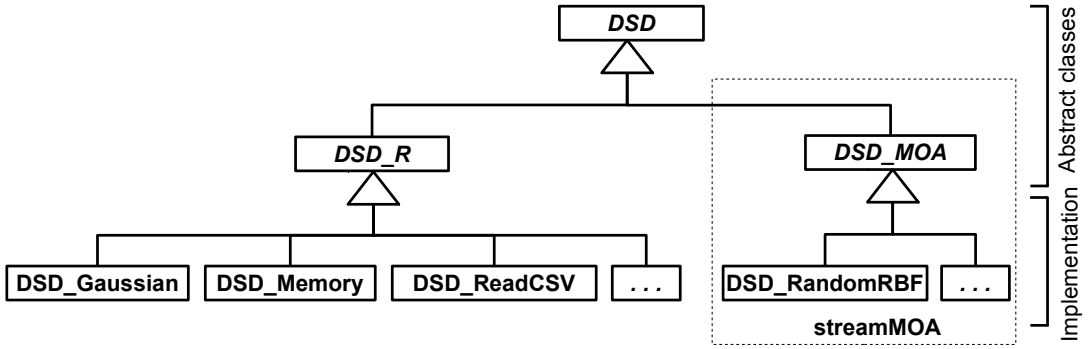


Figure 3: Overview of the data stream data (DSD) class structure.

1. A creator function. This function typically has the same name as the class. By definition the function name starts with the prefix `DSD_`. The list of parameters depends on the type of data stream it creates. The most common input parameters for the creation of DSD classes for clustering are `k`, the number of clusters (i.e., dense areas), and `d`, the number of dimensions. A full list of parameters can be obtained from the help page for each class. The result of this creator function is not a data set but an object representing the stream's properties and its current state.

2. A data generating function
   `get_points(x, n = 1, outofpoints = c("stop", "warn", "ignore") , info = TRUE, ...)`.
   This function is used to obtain the next data point (or next `n` data points) from the stream represented by object `x`. Parameter `outofpoints` controls how to deal with a stream which runs out of points (the stream source does not provide more points at this time). For `"warn"` and `"ignore"` all (possibly zero) available points are returned. For clustering data, the data points are returned as a data frame with each row representing a single data point. For other types of data streams (e.g., transaction data for frequent pattern mining), the returned points might be represented in a different, appropriate way (e.g., as a list).

Next to these core functions several utility functions like `print()`, `plot()` and `write_stream()`, to save a part of a data stream to disk, are provided by **stream** for class `DSD` and are available for all data stream sources. Different data stream implementations might have additional functions implemented. For example, `DSD_Gaussians`, `DSD_Memory` and `DSD_ReadCSV` provide `reset_stream()` to reset the position in the stream to its beginning.

Next we give some examples of how to manage data streams using **stream**. In Section 4.2 we start with creating a data stream using different implementations of the DSD class. The second example in Section 4.4 shows how to save and read stream data to and from disk. Section 4.5 gives examples for how to reuse the same data from a stream in order to perform comparison experiments with multiple data stream mining algorithms on exactly the same data. All examples contain the complete code necessary for replication.

4.2. Example: Creating a data stream

```
R> library("stream")
R> set.seed(1000)
R> stream <- DSD_Gaussians(k = 3, d = 3, noise = .05, p = c(.5, .3, .1))
R> stream

Gaussian Mixture (d = 3, k = 3)
Class: DSD_Gaussians, DSD_R, DSD
```

After loading the **stream** package we call the creator function for the class `DSD_Gaussians` specifying the number of clusters as $k = 3$ and a data dimensionality of $d = 3$ with an added noise of 5% of the generated data points. Each cluster is represented by a multivariate Gaussian distribution with a randomly chosen mean (cluster center) and covariance matrix. New data points are requested from the stream using `get_points()`. When a new data point is requested from this generator, a cluster is chosen randomly (using the probability weights in `p`) and then a point is drawn from the multivariate Gaussian distribution given by the mean and covariance matrix of the cluster. Noise points are generated in a bounding box from a d-dimensional uniform distribution. The following instruction requests $n = 10$ new data points.

```
R> p <- get_points(stream, n = 10)
R> p

      X1     X2     X3 .class
1  0.818 0.0721 0.1557      1
2  0.521 0.0899 0.6376      2
3  0.613 0.0218 0.5209      2
4  0.787 0.0864 0.0822      1
5  0.990 0.5088 0.6938      3
6  0.760 0.0935 0.1135      1
7  0.995 0.2360 0.1947     NA
8  0.550 0.0161 0.5907      2
9  0.571 0.0840 0.5301      2
10 0.591 0.0724 0.5423      2
```

The result is a data frame containing the data points as rows. For evaluation it is often important to know the ground truth, i.e., from which cluster each point was created. Many generators also return the ground truth (class or cluster label) and other information as columns starting with ‘.’. Note that the data was created by a generator with 5% noise. Noise points do not belong to any cluster and thus have a class label of `NA`.

Next, we plot 500 points from the data stream to get an idea about its structure.

```
R> plot(stream, n = 500)
```

The resulting scatter plot matrix is shown in Figures 4. The assignment values are automatically used to distinguish between clusters using color and different plotting symbols. Noise points are plotted as gray dots. The data can also be projected on its first two principal components using `method="pc"`.

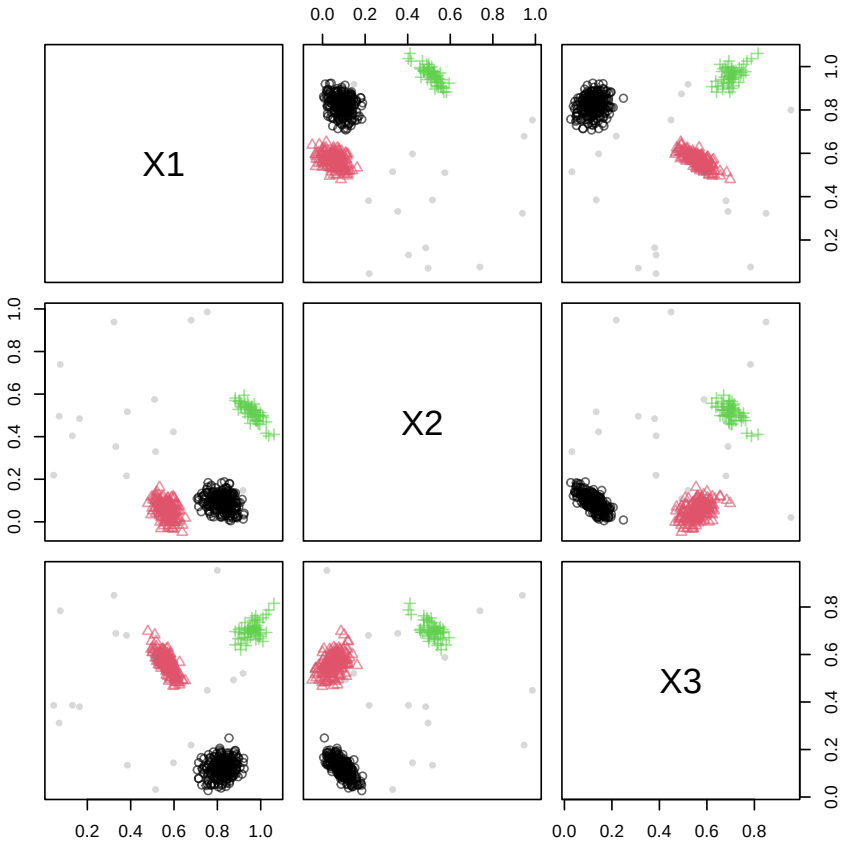


Figure 4: Plotting 500 data points from the data stream.

```
R> plot(stream, n = 500, method = "pc")
```

Figures 5 show the projected data.

Stream also supports data streams which contain concept drift. Several examples of such data stream generators are collected in `DSD_Benchmark`. We create an instance of the first benchmark generator which creates two clusters moving in two-dimensional space. One moves from top left to bottom right and the other one moves from bottom left to top right. Both clusters overlap when they meet exactly in the center of the data space.

```
R> set.seed(1000)
R> stream <- DSD_Benchmark(1)
R> stream

Benchmark 1: Two clusters moving diagonally from left to right,
meeting in the center (d = 2, k = 2, 5% noise).Class: DSD_MG, DSD_R, DSD
With 3 clusters in 2 dimensions. Time is 1
```

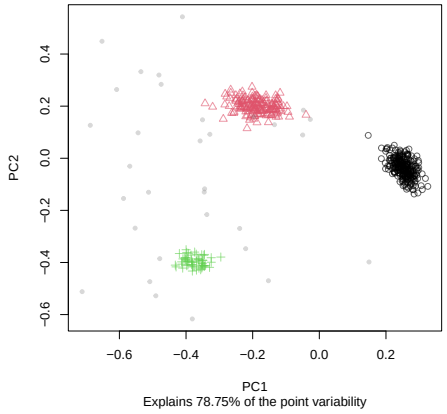


Figure 5: Plotting 500 data points from the data stream projected onto its first two principal components.

To show concept drift, we request four times 250 data points from the stream and plot them. To fast-forward in the stream we request 1400 points in between the plots and ignore them.

```
R> for(i in 1:4) {
+   plot(stream, 250, xlim = c(0, 1), ylim = c(0, 1))
+   tmp <- get_points(stream, n = 1400)
+ }
```

Figure 6 shows the four plots where clusters move over time. Arrows are added to highlight the direction of cluster movement. An animation of the data can be generated using `animate_data()`. We use `reset_stream()` to start the animation at the beginning of the stream.

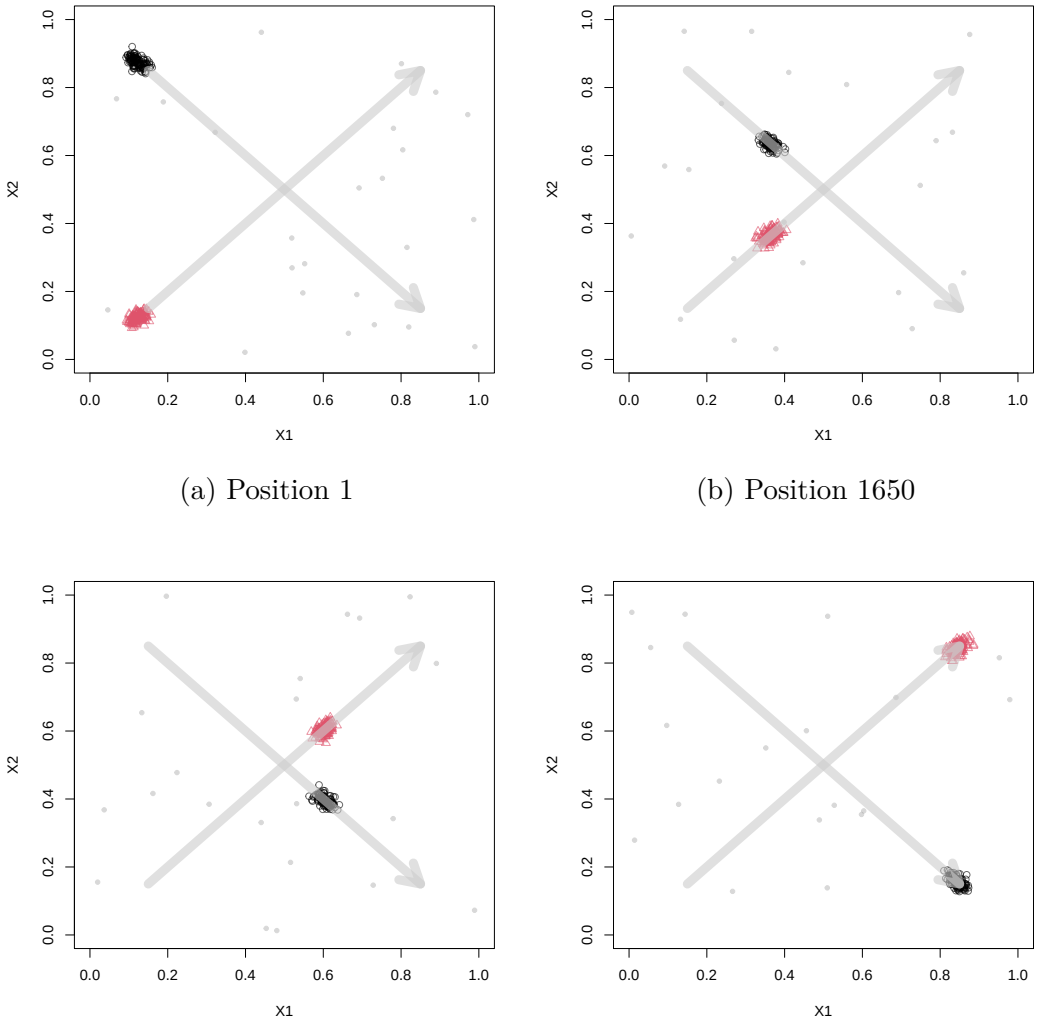
```
R> reset_stream(stream)
R> animate_data(stream, n = 10000, horizon = 100,
+   xlim = c(0, 1), ylim = c(0, 1))
```

Animations are recorded using package **animation** (Xie 2015) and can be replayed using `ani.replay()`.

```
R> library("animation")
R> animation::ani.options(interval = .1)
R> ani.replay()
```

Animations can also be saved as an animation embedded in a HTML document or an animated image in the Graphics Interchange Format (GIF) which can easily be used in presentations.

```
R> saveHTML(ani.replay())
R> saveGIF(ani.replay())
```



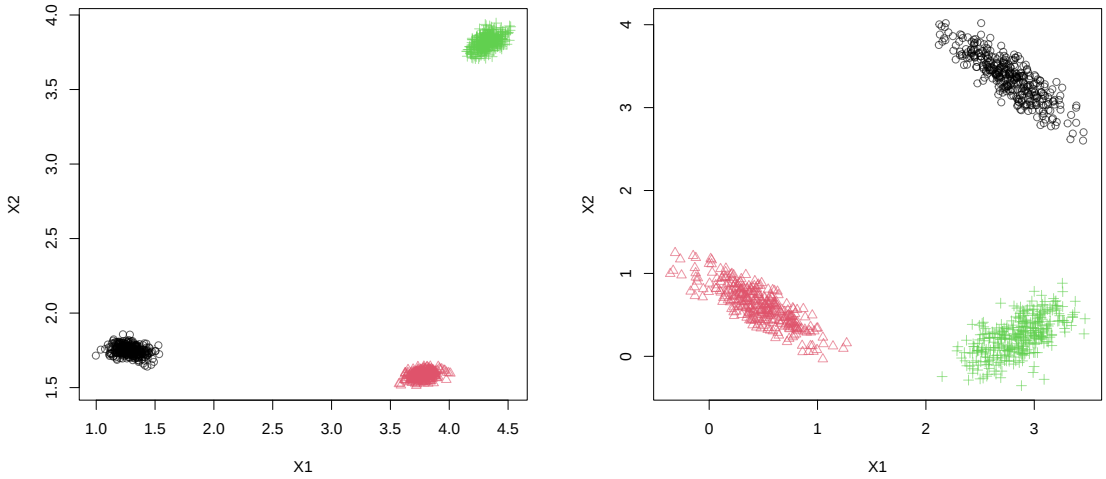
(a) Position 1

(b) Position 1650

(c) Position 3300

(d) Position 4950

Figure 6: Data points from `DSD_Benchmark(1)` at different positions in the stream. The two arrows are added to highlight the direction of movement.



(a) Variance between 0 and 0.01 (b) Variance between .05 and 0.1

Figure 7: Data points from `DSD_Gaussians` having maximal variance limit and space limits.

More formats for saving the animation are available in package **animation**.

4.3. Example: Advanced statistical data streams

`DSD_Gaussians` has capabilities to generate more complex statistical data streams. In the previous examples, we used simple cluster and outlier generating capabilities and Euclidean distance for their separation.

Maximal variance and space limitations

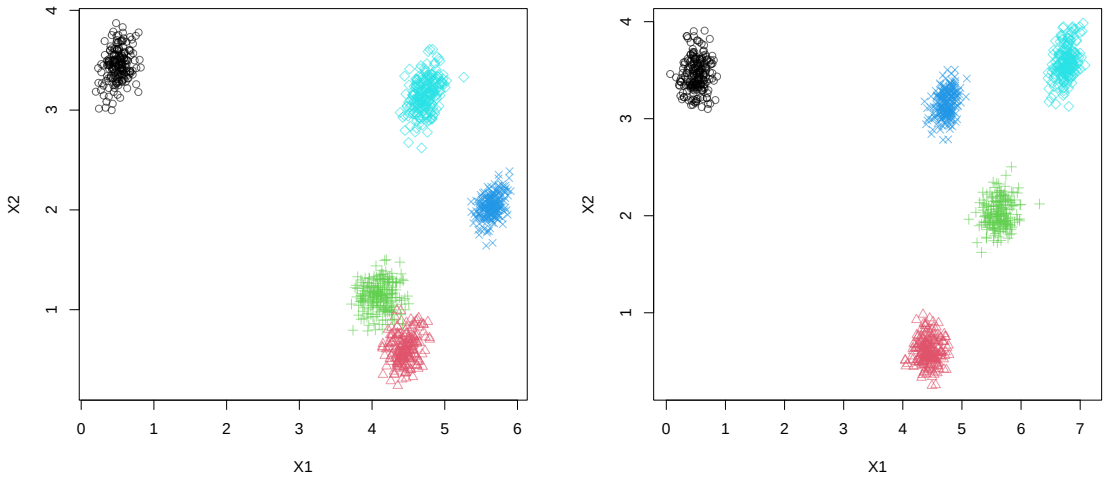
In case we do not predefine covariance matrices by using `sigma` parameter, `DSD_Gaussians` can randomly generate covariance matrices. Maximal variance used to generate covariance matrices can be limited, which comes together with space limitation to fit clusters.

```
R> library("stream")
R> set.seed(1000)
R> stream1 <- DSD_Gaussians(k = 3, d = 2, variance_limit = c(0, 0.01),
+                           space_limit = c(0, 5))
R> stream2 <- DSD_Gaussians(k = 3, d = 2, variance_limit = c(.05, .1),
+                           space_limit = c(0, 5))
```

Next, we plot 1000 points from the data stream, which can be seen in Figure 7. As seen in Figure 7b, we can experience overlapping of clusters due to high maximal variance limit.

Keeping clusters sufficiently separated

To keep cluster from overlapping we can use two separation distance measures: Euclidean and Mahalanobis (the statistical distance). While Euclidean distance can be used to some extent,



(a) Mahalanobis separation = 3 (b) Mahalanobis separation = 10

Figure 8: Data points from `DSD_Gaussians` having distinct Mahalanobis separation values.

it might not keep clusters cleanly separated at all times, since cluster size highly depend on the related covariance matrix. This is the reason why we want do use statistical distance (Mahalanobis) to control cluster separation.

```
R> library("stream")
R> set.seed(1000)
R> stream1 <- DSD_Gaussians(k = 5, d = 2, variance_limit = c(0.01, 0.03),
+                           space_limit = c(0, 7),
+                           separation_type = "Mahalanobis",
+                           separation = 3)

R> set.seed(1000)
R> stream2 <- DSD_Gaussians(k = 5, d = 2, variance_limit = c(0.01, 0.03),
+                           space_limit = c(0, 7),
+                           separation_type = "Mahalanobis",
+                           separation = 10)
```

Plots comprising 1000 points from the data stream can be seen in Figure 8.

4.4. Example: Reading and writing data streams

Although data streams are potentially unbounded by definition and thus storing the complete stream is infeasible, it is often useful to store parts of a stream on disk. For example, a small part of a stream with an interesting feature can be used to test how a new algorithm handles this particular case. **stream** has support for reading and writing parts of data streams through R connections which provide a set of functions to interface file-like objects including files, compressed files, pipes, URLs or sockets (R Foundation 2011).

We start the example by creating a DSD object.

```
R> library("stream")
R> set.seed(1000)
R> stream <- DSD_Gaussians(k = 3, d = 5)
```

Next, we write 100 data points to disk using `write_stream()`.

```
R> write_stream(stream, "data.csv", n = 100, sep = ",")
```

`write_stream()` accepts a DSD object, and then either a connection or a file name. The instruction above creates a new file called `dsd_data.csv`. The `sep` parameter defines how the dimensions in each data point (row) are separated. Here a comma is used to create a comma separated values file. The actual writing is done by R's `write.table()` function and additional parameters are passed on. Data points are requested blockwise (defaults to 100,000 points) from the stream and then written to the connection. This way the only restriction for the size of the written stream are limitations at the receiving end (e.g., the available storage).

Finally, parameters `class` and `write_outliers` can be used to control writing of the class information and outlier marks. These two details are stored in fields named "class" and "outlier" respectively, and can be read again.

The `DSD_ReadCSV` object is used to read a stream from a connection or a file. It reads only the specified number of data points at a time using the `read.table()` function. Since, after the read data is processed, e.g., by a data stream clustering algorithm, it is removed from memory, we can efficiently process files larger than the available main memory in a streaming fashion. In the following example we create a data stream object representing data stored as a compressed CSV-file in the package's examples directory.

```
R> file <- system.file("examples", "kddcup10000.data.gz", package = "stream")
R> stream_file <- DSD_ReadCSV(gzfile(file),
+   take = c(1, 5, 6, 8:11, 13:20, 23:41, .class = 42), k = 7)
R> stream_file

Data Stream from Connection (d = 34, k = 7)
Class: DSD_ReadStream, DSD_R, DSD
connection: kddcup10000.data.gz (opened)
```

Using `take`, `class`, and `outlier` we define which columns should be used as data, which column contains the ground truth assignment, and which column contains outlier marks. We also specify the true number of clusters k and outliers o. Ground truth and number of clusters do not need to be specified if they are not available or no evaluation is planned. Note that at this point no data has been read in. Reading only occurs when `get_points` is called.

```
R> get_points(stream_file, n = 5)

  V1  V5    V6 V8 V9 V10 V11 V13 V14 V15 V16 V17 V18 V19 V20 V23 V24 V25
1  0 215 45076  0  0   0   0   0   0   0   0   0   0   0   0   1   1   0
2  0 162  4528  0  0   0   0   0   0   0   0   0   0   0   0   2   2   0
3  0 236  1228  0  0   0   0   0   0   0   0   0   0   0   0   1   1   0
```

```
4  0 233  2032  0  0   0   0   0   0   0   0   0   0   0   0   2   2   0
5  0 239   486  0  0   0   0   0   0   0   0   0   0   0   0   3   3   0
  V26 V27 V28 V29 V30 V31 V32 V33 V34 V35  V36 V37 V38 V39 V40 V41  .class
1   0   0   0   1   0   0   0   0   0   0 0.00   0   0   0   0   0 normal.
2   0   0   0   1   0   0   1   1   1   0 1.00   0   0   0   0   0 normal.
3   0   0   0   1   0   0   2   2   1   0 0.50   0   0   0   0   0 normal.
4   0   0   0   1   0   0   3   3   1   0 0.33   0   0   0   0   0 normal.
5   0   0   0   1   0   0   4   4   1   0 0.25   0   0   0   0   0 normal.
```

For clustering it is often necessary to normalize data first. Streams can be scaled and centered in-flight using `DSD_ScaleStream`. The scaling and centering factors are computed from a set of points (by default 1000) from the beginning of the stream.

```
R> stream_scaled <- DSD_ScaleStream(stream_file, center = TRUE, scale = TRUE)
R> get_points(stream_scaled, n = 5)
```

```
  V1     V5     V6 V8 V9    V10 V11 V13 V14 V15 V16 V17 V18 V19 V20  V23
1  0 -0.228 -0.190  0  0 -0.142   0   0   0   0   0   0   0   0   0 3.38
2  0 -0.201 -0.494  0  0 -0.142   0   0   0   0   0   0   0   0   0 3.73
3  0 -0.201 -0.499  0  0 -0.142   0   0   0   0   0   0   0   0   0 4.08
4  0 -0.147 -0.527  0  0 -0.142   0   0   0   0   0   0   0   0   0 4.43
5  0 -0.201 -0.486  0  0 -0.142   0   0   0   0   0   0   0   0   0 4.78
   V24 V25 V26 V27 V28 V29 V30   V31    V32  V33 V34 V35    V36  V37 V38
1 3.17   0   0   0   0   0   0 -0.29 -0.447 1.74   0   0 -0.345 1.46   0
2 3.51   0   0   0   0   0   0 -0.29 -0.395 1.78   0   0 -0.345 1.46   0
3 3.85   0   0   0   0   0   0 -0.29 -0.343 1.81   0   0 -0.393 1.46   0
4 4.19   0   0   0   0   0   0 -0.29 -0.291 1.84   0   0 -0.393 1.46   0
5 4.53   0   0   0   0   0   0 -0.29 -0.239 1.88   0   0 -0.393 1.46   0
  V39 V40 V41  .class
1   0   0   0 normal.
2   0   0   0 normal.
3   0   0   0 normal.
4   0   0   0 normal.
5   0   0   0 normal.
```

4.5. Example: Replaying a data stream

An important feature of **stream** is the ability to replay portions of a data stream. With this feature we can capture a special feature of the data (e.g., an anomaly) and then adapt our algorithm and test if the change improved the behavior on exactly that data. Also, this feature can be used to conduct experiments where different algorithms need to be compared using exactly the same data.

There are several ways to replay streams. As described in the previous section, we can write a portion of a stream to disk with `write_stream()` and then use `DSD_ReadCSV` to read the stream portion back every time it is needed. However, often the interesting portion of the stream is small enough to fit into main memory or might be already available as a matrix or a data frame in R. In this case we can use the DSD class `DSD_Memory` which provides a stream interface for a matrix-like objects.

For illustration purposes, we use data for four major European stock market indices available in R as a data frame.

```
R> data("EuStockMarkets", package = "datasets")
R> head(EuStockMarkets)

Time Series:
Start = c(1991, 130)
End = c(1991, 135)
Frequency = 260
      DAX  SMI  CAC FTSE
1991 1629 1678 1773 2444
1992 1614 1688 1750 2460
1992 1607 1679 1718 2448
1992 1621 1684 1708 2470
1992 1618 1687 1723 2485
1992 1611 1672 1714 2467
```

Next, we create a `DSD_Memory` object. The number of true clusters k is unknown.

```
R> replayer <- DSD_Memory(EuStockMarkets, k = NA)
R> replayer

Memorized Stream
Class: DSD_Memory, DSD_R, DSD
Contains 1860 data points - currently at position 1 - loop is FALSE
```

Every time we get a point from replayer, the stream moves to the next position (row) in the data.

```
R> get_points(replayer, n = 5)

   DAX  SMI  CAC FTSE
1 1629 1678 1773 2444
```

```
2 1614 1688 1750 2460
3 1607 1679 1718 2448
4 1621 1684 1708 2470
5 1618 1687 1723 2485
```

```
R> replayer
```

```
Memorized Stream
Class: DSD_Memory, DSD_R, DSD
Contains 1860 data points - currently at position 6 - loop is FALSE
```

Note that the stream is now at position 6. The stream only has 1854 points left and the following request for more than the available number of data points results in returning all remaining points and a warning.

```
R> points <- get_points(replayer, n = 2000)
R> dim(points)
```

```
[1] 1855    4
```

Note that this behavior can be changed to an error or ignoring the problem with the parameter `outofpoints` when constructing table-based data streams like `DSD_Memory`, `DSD_ReadStream`, `DSD_ReadStream`, and `DSD_ReadDB`. `DSD_Memory` and `DSD_ReadCSV` can also be created to loop indefinitely, i.e., start over once the last data point is reached. This is achieved by passing `loop = TRUE` to the creator function. The current position in the stream for those two types of DSD classes can also be reset to the beginning of the stream or, for `DSD_Memory`, to an arbitrary position via `reset_stream()`. Here we set the stream to position 100.

```
R> reset_stream(replayer, pos = 100)
R> replayer
```

```
Memorized Stream
Class: DSD_Memory, DSD_R, DSD
Contains 1860 data points - currently at position 100 - loop is FALSE
```

`DSD_Memory` also accepts other matrix-like objects. This includes data shared between processes or data that is too large to fit into main memory represented by memory-mapped files using `ffdf` objects from package **ff** (Adler et al. 2014) or `big.matrix` objects from package **bigmemory** (Kane et al. 2013). In fact any object that provides basic matrix functions like `dim()` and subsetting with `"["` can be used.

5. Data stream task (DST)

After choosing a DSD class to use as the data stream source, the next step in the workflow is to define a data stream task (DST). In **stream**, a DST refers to any data mining task that can be applied to data streams. The design is flexible enough for future extensions including

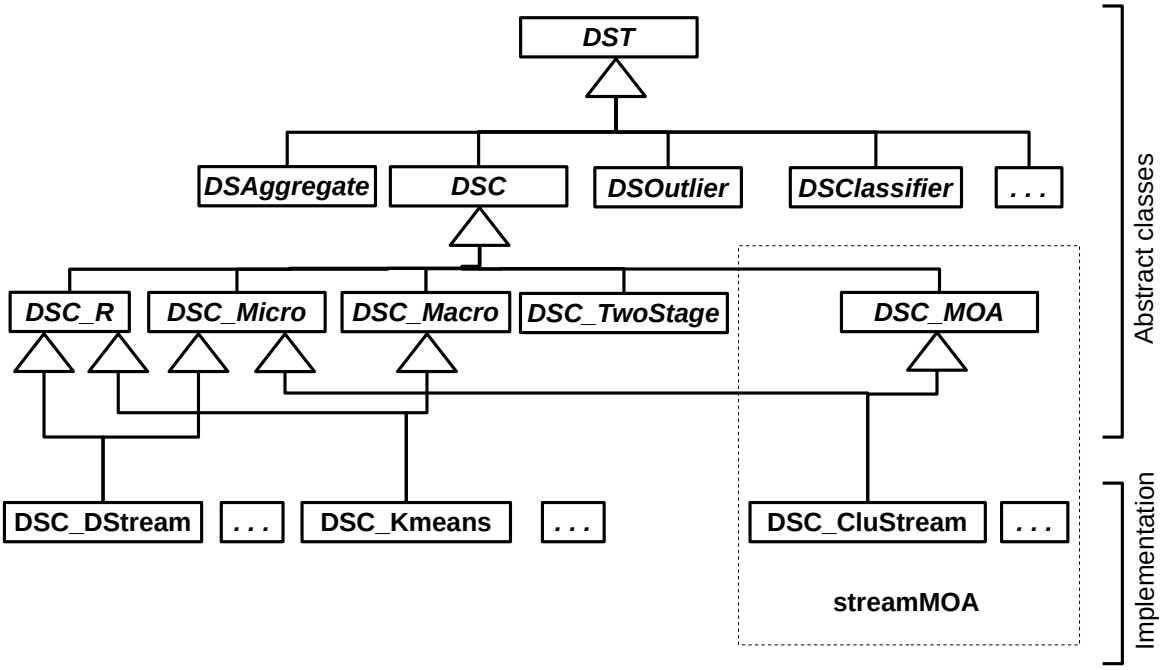


Figure 9: Overview of the data stream task (DST) class structure with subclasses for data stream operators (DSO), clustering (DSC), classification (DSClass) and frequent pattern mining (DSFPM).

even currently unknown tasks. Figure 9 shows the class hierarchy for DST. It is important to note that the DST base class is shown merely for conceptual purpose and is not directly visible in the code. The reason is that the actual implementations of data stream operators (DSO), clustering (DSC), classification (DSClass) or frequent pattern mining (DSFPM) are typically quite different and the benefit of sharing methods would be minimal.

DST classes implement mutable objects which can be changed without creating a copy. This is more efficient, since otherwise a new copy of all data structures used by the algorithm would be created for processing each data point. Mutable objects can be implemented in R using environments or the recently introduced reference class construct (see package **methods** by the R Core Team (2014)). Alternatively, pointers to external data structures in Java or C/C++ can be used to create mutable objects.

We will restrict the following discussion to data stream clustering (DSC) since **stream** currently focuses on this task. **stream** currently provides moving windows and sampling from a stream as data stream operators (DSO). The operators provide simple functionality which can be used by other tasks and we will discuss them in the context of clustering. Packages which cover the other tasks using the **stream** framework are currently under development.

5.1. Introduction to data stream clustering (DSC)

Data stream clustering algorithms are implemented as subclasses of the abstract class `DSC` (see Figure 9). First we differentiate between different interfaces for clustering algorithms. `DSC_R`

provides a native R interface, while `DSC_MOA` (available in **streamMOA**) provides an interface to algorithms implemented for the Java-based MOA framework. DSCs implement the online process as subclasses of `DSC_Micro` (since it produces micro-clusters) and the offline process as subclasses of `DSC_Macro`. To implement the typical two-stage process in data stream clustering, **stream** provides `DSC_TwoStage` which can be used to combine any available micro and macro-clustering algorithm.

The following functions can be used for objects of subclasses of DSC:

- A creator function which creates an empty clustering. Creator function names by definition start with the prefix `DSC_`.
- `update(dsc, dsd, n = 1, verbose = FALSE, ...)` which accepts a DSC object and a DSD object. It requests the `n` data points from `dsd` and adds them to the clustering in `dsc`.
- `nclusters(x, type = c("auto", "micro", "macro"), ...)` returns the number of clusters currently in the DSC object. This is important since the number of clusters is not fixed for most data stream clustering algorithms.

  DSC objects can contain several clusterings (e.g., micro and macro-clusters) at the same time. The default value for `type` is `"auto"` and results in `DSC_Micro` objects to return micro-cluster information and `DSC_Macro` objects to return macro-cluster information. Most `DSC_Macro` objects also store micro-clusters and using `type` these can also be retrieved. Some `DSC_Micro` implementations also have a reclustering procedure implemented and `type` also allows the user to retrieve macro-cluster information. Trying to access cluster information that is not available in the clustering results in an error. `type` is also available for many other functions.
- `get_centers(x, type = c("auto", "micro", "macro"), ...)` returns the centers of the clusters of the DSC object. Depending on the clustering algorithm the centers can be centroids, medoids, centers of dense grids, etc.
- `get_weights(x, type = c("auto", "micro", "macro"), ...)` returns the weights of the clusters in the DSC object `x`. How the weights are calculated depends on the clustering algorithm. Typically they are a function of the number of points assigned to each cluster.
- `get_assignment(dsc, points, type = c("auto", "micro", "macro"), method = c("auto", "model", "nn"), ...)` returns a cluster assignment vector indicating to which cluster each data point in `points` would be assigned. The assignment can be determined by the model (e.g., point falls inside the radius of the micro-cluster) or via nearest neighbor assignment (`"nn"`). `method = "auto"` selects model-based assignment if available and otherwise defaults to nearest neighbor assignment. Note that model-based assignment might result in some points not being assigned to any cluster (i.e., an assignment value of `NA`) which indicates a noise data point.
- `get_copy(x)` creates a deep copy of a DSC object. This is necessary since clusterings are represented by mutable objects (R-based reference classes or external data structures). Calling this function results in an error if a mechanism for creating a deep copy is not available for the used DSC implementation.

- `plot(x, dsd = NULL, ..., method = "pairs", dim = NULL, type = c("auto", "micro", "macro", "both", "outliers", "all")` (see manual page for more available parameters) plots the centers of the clusters and marks detected outliers. There are 3 available plot methods: `"pairs"`, `"scatter"`, `"pc"`. Method `"pairs"` is the default method and produces a matrix of scatter plots that plots all attributes against one another (this method defaults to a regular scatter plot for `d = 2`). Method `"scatter"` takes the attributes specified in `dim` (the first two if `dim` is unspecified) and plots them in a scatter plot. Lastly, method `"pc"` performs Principle Component Analysis (PCA) on the data and projects the data onto a 2-dimensional plane for plotting. Parameter `type` controls plotting of cluster and outlier markings. User can select to plot micro- (`micro`), macro-clusters (`macro`), both micro and macro clusters (`both`), outliers (`outliers`), and everything (`all`). If a DSD object is provides as `dsd`, then some example data points are plotted in the background in light gray.

- `print(x, ...)` prints common attributes of the DSC object. This includes a short description of the underlying algorithm and the number of clusters that have been calculated.

We can add `DSC_Outlier` abstract class anywhere between `DSC` abstract class and a concrete clusterer implementation class. This means that the clusterer has additional outlier detection capabilities. The following functions can be used for objects of subclasses of `DSC_Outlier`:

- `clean_outliers(x, ...)` instructs the outlier detector to clean up the outlier list.

- `get_outlier_positions(x, ...)` returns positions of currently detected outliers.

- `recheck_outlier(x, outlier_correlated_id, ...)` invokes re-checking whether previously detected outlier identifier by `outlier_correlated_id` is still an outlier (TRUE) or has become an inlier in the meantime (FALSE).

- `noutlier(x, ...)` returns the number of current outliers.

- `print(x, ...)` prints out DSC object details that include detected outliers.

- `get_assignment(x, points, type=c("auto", "micro", "macro"), method=c("auto", "nn", "model"), outlier_threshold=0.05, ...)` returns a data frame comprising cluster assignments for related data point in `points` argument. As an addition, attributes `outliers` and `outliers_corrid` are returned with assignment data frame. Attribute `outliers` comprises outlier marks, while attribute `outliers_corrid` comprises outlier identifiers.

All single-pass clusterers must have abstract class `DSC_SinglePass` anywhere between abstract class `DSC` and a concrete clusterer class. `DSC_SinglePass` abstract class indicates that the clusterer does automatic model update automatically when processing each data point retrieved from the input data stream (`DSD`). It is necessary that single-pass clusterers override implementations of the following methods:

- `update(dsc, dsd, n = 1, verbose = FALSE, ...)`

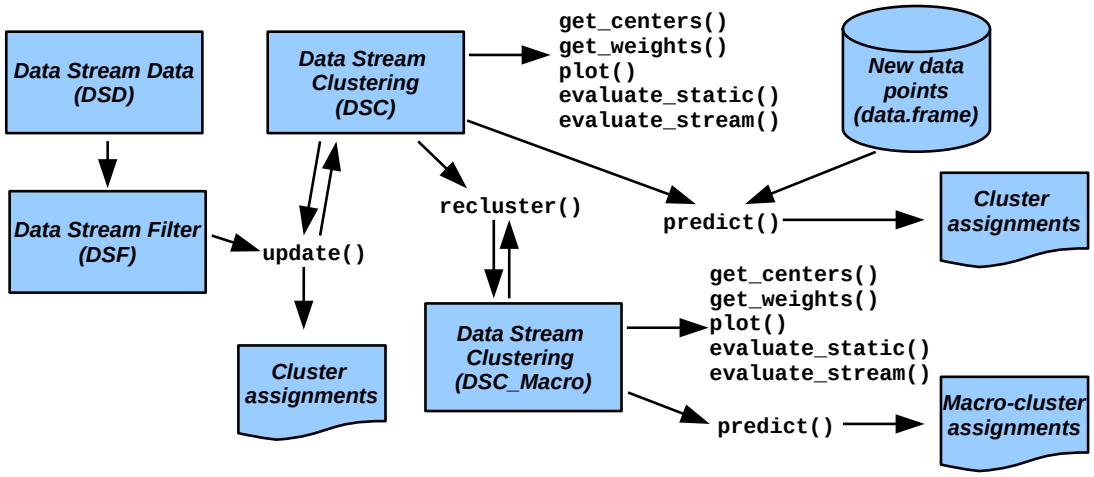


Figure 10: Interaction between the DSD and DSC classes.

- `get_assignment(dsc, points, type=c("auto", "micro", "macro"), method=c("auto", "nn", "model"), ...)`

Figure 10 shows the typical use of `update()` and other functions. Clustering on a data stream (DSD) is performed with `update()` on a DSC object. This is typically done with a `DSC_Micro` object which will perform its online clustering process and the resulting micro-clusters are available from the object after clustering (via `get_centers()`, etc.). Note, that DSC classes implement mutable objects and thus the result of `update()` does not need to be reassigned to its name.

Reclustering (the offline component of data stream clustering) is performed with

```
recluster(macro, micro, type="auto", ...),
```

where `micro` and `macro` are objects of class `DSC`. Here the centers in `micro` are used as pseudo-points by the `DSC_macro` object `macro`. After reclustering the macro-clusters can be inspected (using `get_centers()`, etc.) and the assignment of micro-clusters to macro-clusters is available via `microToMacro()`.

The following data stream clustering algorithms are currently available:

- `DSC_CluStream` (**streamMOA**) interfaces the MOA implementation of the CluStream algorithm by Aggarwal et al. (2003). The algorithm maintains a user-specified number of micro-clusters. The number of clusters is held constant by merging and removing clusters. The suggested reclustering method is weighted k-means.

- `DSC_ClusTree` (**streamMOA**) interfaces the MOA implementation of the ClusTree algorithm by Kranen, Assent, Baldauf, and Seidl (2009). The algorithm organizes micro-clusters in a tree structure for faster access and automatically adapts micro-cluster sizes based on the variance of the assigned data points. Either k-means or reachability from DBSCAN can be used for reclustering.

- `DSC_DenStream` (**streamMOA**) interfaces MOA's implementation of the DenStream algorithm by Cao et al. (2006). DenStream estimates the density of micro-clusters in a user-specified neighborhood. To suppress noise, it also organizes micro-clusters based on their weight as core and outlier micro-clusters. Core Micro-clusters are reclustered using reachability from DBSCAN.

- `DSC_DStream` implements the D-Stream algorithm by Chen and Tu (2007). D-Stream uses a grid to estimate density in grid cells. For reclustering adjacent dense cells are merged to form macro-clusters. Alternatively, the concept of attraction between grids cells can be used for reclustering (Tu and Chen 2009).

- `DSC_Sample` provides a clustering interface to the data stream operator `DSO_Sample`. It selects a user-specified number of representative points from the stream via Reservoir Sampling (Vitter 1985). It keeps an unbiased sample of all data points seen thus far using the algorithm by McLeod and Bellhouse (1983). For evolving data streams it is more appropriate to bias the sample toward more recent data points. For biased sampling, the method called Algorithm 2.1 by Aggarwal (2006) is also implemented.

- `DSC_DBSTREAM` (Hahsler and Bolaños 2016) implements an extension of the simple data stream clustering algorithm called tNN threshold nearest-neighbors (tNN) which was developed for package **rEMM** by Hahsler and Dunham (2015, 2010). Micro-clusters are defined by a fixed radius (threshold) around their center. Reachability from DBSCAN is used for reclustering.

- `DSC_Window` provides a clustering interface to the data stream operator `DSO_Window`. It implements the sliding window and the dampened window models (Zhu and Shasha 2002) which keep a user-specified number (window length) of the most recent data points of the stream. For the dampened window model, data points in the window have a weight that deceases exponentially with age.

Although the authors of most data stream clustering algorithms suggest a specific reclustering method, in **stream** any available method can be applied. For reclustering, the following clustering algorithms are currently available as subclasses of `DSC_Macro`:

- `DSC_DBSCAN` interfaces the weighted version of DBSCAN (Ester et al. 1996) implemented in package **dbscan** (Hahsler 2017).

- `DSC_Hierarchical` interfaces R's `hclust` function.

- `DSC_Kmeans` interface R's k-means implementation and a version of k-means where the data points (micro-clusters) are weighted by the micro-cluster weights, i.e., a micro-cluster representing more data points has more weight.

- `DSC_Reachability` uses DBSCAN's concept of reachability for micro-clusters. Two micro-clusters are directly reachable if they are closer than a user-specified distance `epsilon` from each other (they are within each other's `epsilon`-neighborhood). Two micro-clusters are reachable and therefore assigned to the same macro-cluster if they are connected by a chain of directly reachable micro-clusters. Note that this concept is related to hierarchical clustering with single linkage and the dendrogram cut at he height of epsilon.

For outlier detection, the following clustering algorithms are currently available as subclasses of `DSC_SinglePass` and `DSC_Outlier`:

- `DSC_MCOD` (**streamMOA**) interfaces the MOA implementation of the MCOD algorithm by Kontaki et al. (2016). This is a micro-clusterer and outlier detector algorithm. For the macro-clustering it needs an additional macro-clusterer algorithm, to improve clustering results.

All single-pass and outlier examples are given in the **streamMOA** package, since `DSC_MCOD` is currently the only algorithm that was implemented to support such functionalities.

Some non-outlier detecting data clustering algorithms create small clusters for noise or outliers in the data. **stream** provides `prune_clusters(dsc, threshold = .05, weight = TRUE)` to remove a given percentage (given by `threshold`) of the clusters with the least weight. The percentage is either computed based on the number of clusters (e.g., remove 5% of the number of clusters) or based on the total weight of the clustering (e.g., remove enough clusters to reduce the total weight by 5%). The default `weight = TRUE` is based on the total weight. The resulting clustering is a static copy (`DSC_Static`). Further clustering cannot be performed with this object, but it can be used as input for reclustering and for evaluation. Pruning is also available in many macro-clustering algorithms as parameter `min_weight` which excludes all micro-clusters with a weight less than the specified value before reclustering.

To specify a full data stream clustering process with an arbitrarily chosen online and offline algorithm, **stream** implements a special DSC class called `DSC_TwoStage` which can combine any `DSC_Micro` and `DSC_Macro` implementation into a two-stage process.

In the following section we give a short example for how to cluster a data stream.

5.2. Example: Clustering a data stream

In this example we show how to cluster data using DSC implementations. First, we create a data stream (three Gaussian clusters in two dimensions with 5% noise).

```
R> library("stream")
R> set.seed(1000)
R> stream <- DSD_Gaussians(k = 3, d = 2, noise = .05)
```

Next, we prepare the clustering algorithm. We use here `DSC_DStream` which implements the D-Stream algorithm (Tu and Chen 2009). D-Stream assigns points to cells in a grid. For the example we use a gridsize of 0.1.

```
R> dstream <- DSC_DStream(gridsize = .1, Cm = 1.2)
R> dstream

D-Stream
Class: DSC_DStream, DSC_Micro, DSC_R, DSC
Number of micro-clusters: 0
Number of macro-clusters: 0
```

After creating an empty clustering, we are ready to cluster data from the stream using the `update()` function. Note, that `update()` will implicitly alter the mutable DSC object so no reassignment is necessary.

```
R> update(dstream, stream, n = 500)
R> dstream
```

```
D-Stream
Class: DSC_DStream, DSC_Micro, DSC_R, DSC
Number of micro-clusters: 10
Number of macro-clusters: 3
```

After clustering 500 data points, the clustering contains 10 micro-clusters. Note that the implementation of D-Stream has built-in reclustering and therefore also shows macro-clusters. The first few micro-cluster centers are:

```
R> head(get_centers(dstream))
```

```
    X1   X2
1 0.15 0.35
2 0.25 0.35
3 0.25 0.45
4 0.35 0.35
5 0.75 0.25
6 0.75 0.35
```

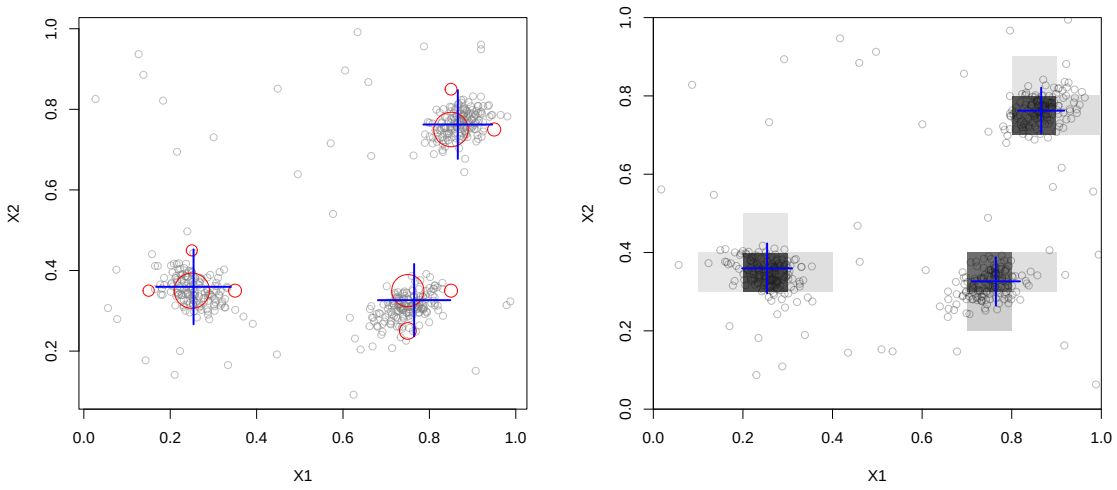
It is often helpful to visualize the results of the clustering operation.

```
R> plot(dstream, stream)
```

For the grid-based D-Stream algorithm there is also a second type of visualization available which shows the used dense and transitional grid cells as gray squares.

```
R> plot(dstream, stream, grid = TRUE)
```

The resulting plots are shown in Figure 11. In Figure 11(a) the micro-clusters are plotted in red on top of gray data points. The size of the micro-clusters indicates the weight, i.e., the number of data points represented by each micro-cluster. In Figure 11(b) the micro-clusters are shown as dense grid cells (density is coded with gray values).



(a) (b)

Figure 11: Plotting the micro-clusters produced by D-Stream together with the original data points. Shown as (a) micro-clusters and as (b) dense grid cells.

6. Evaluation of data stream clustering

6.1. Introduction

Evaluation of data stream mining is an important issue. The evaluation of conventional clustering is discussed in the literature extensively and there are many evaluation criteria available. For an overview we refer the reader to the popular books by Jain and Dubes (1988) and Kaufman and Rousseeuw (1990). However, this evaluation only measures how well the algorithm learns static structure in the data. Data streams often exhibit concept drift and it is important to evaluate how well the algorithm is able to adapt to these changes. The evaluation of data stream clustering is still in its infancy. The current state of the evaluation of data stream mining methods including clustering is described in the books by Aggarwal (2007) and Gama (2010), and the papers by Kremer, Kranen, Jansen, Seidl, Bifet, Holmes, and Pfahringer (2011) and Gama, Sebastião, and Rodrigues (2013).

In the following we will discuss how **stream** can be used to evaluate clustering algorithms in terms of learning static structures and clustering dynamic streams.

6.2. Evaluation of clustering static data streams

Evaluation of how well an algorithm is able to learn static structures in a data stream which does not exhibit concept drift is performed in **stream** via

```
evaluate_static(dsc, dsd, measure, n = 100, type = c("auto", "micro",
                                    "macro"),
    assign = "micro", assignmentMethod = c("auto", "model", "nn"),
                    noise = c("class", "exclude"), ...),
```

where `dsc` is the evaluated clustering. `n` data points are taken from `dsd` and used for evaluation. The evaluation measure is specified in `measure`. Several measures can be specified as a vector of character strings. For evaluation, the points are assigned to the clusters in the clustering in `dsc` using `get_assignment()`. By default the points are assigned to micro-clusters, but it is also possible to assign them to macro-cluster centers instead (`assign = "macro"`). New points can be assigned to clusters by the rule used in the clustering algorithm (`assignmentMethod = "model"`) or using nearest-neighbor assignment (`"nn"`). If the assignment method is set to `"auto"` then model assignment is used when available and otherwise nearest-neighbor assignment is used. The initial assignments are aggregated to the level specified in `type`. For example, for a macro-clustering, the initial assignments will be made by default to micro-clusters and then these assignments will be translated into macro-cluster assignments using the micro- to macro-cluster relationships stored in the clustering and available via `microToMacro()`. This separation between assignment and evaluation type is especially important for data with non-spherical clusters where micro-clusters are linked together in chains produced by a macro-clustering algorithm based on hierarchical clustering with single-link or reachability. How noise is handled is controlled by `noise`. Noise points in the data can be considered forming their own class. This is typically appropriate for external validity measures, however, for some internal validity measures using noise points is problematic since the noise data points will not form a compact cluster and thus negatively effect measures like the sum of squares. Therefore, for some internal measures, it is more consistent to exclude noise points.

Clustering evaluation measures can be categorized into internal and external cluster validity measures. Internal measures evaluate properties of the clustering. A simple measure to evaluate the compactness of (spherical) clusters in a clustering is the within-cluster sum of squares, i.e., the sum of squared distances between each data point and the center of its cluster (method `"SSQ"`). External measures use the ground truth (i.e., true partition of the data into groups) to evaluate the agreement of the partition created by the clustering algorithm with a known true partition. In the following we will enumerate the evaluation measures (passed on as `measure`) available in **stream**. We will not describe each measure here since most of them are standard measures which can be found in many text books (e.g., Jain and Dubes 1988; Kaufman and Rousseeuw 1990) or in the documentation supplied with the packages **fpc** (Hennig 2014), **clue** (Hornik 2017) and **cluster** (Maechler et al. 2014). Measures currently available for `evaluate_static()` (method name are under quotation marks and the package that implements the evaluation measure is shown in parentheses) include:

- Information items.
  - `"numMicroClusters"` Number of micro-clusters
  - `"numMacroClusters"` Number of macro-clusters
  - `"numClasses"` Number of classes (i.e., groups in the ground truth)
- Noise-related items.
  - `"noisePredicted"` Number data points predicted as noise
  - `"noiseActual"` Number of data points which are actually noise
  - `"noisePrecision"` Precision of the predicting noise (i.e., number of correctly predicted noise points over the total number of points predicted as noise)

- Internal evaluation measures.
  - `"SSQ"` Within cluster sum of squares. Assigns each non-noise point to its nearest center from the clustering and calculates the sum of squares
  - `"silhouette"` Average silhouette width (actual noise points which stay unassigned by the clustering algorithm are removed; regular points that are unassigned by the clustering algorithm will form their own noise cluster) (**cluster**))
  - `"average.between"` Average distance between clusters (**fpc**)
  - `"average.within"` Average distance within clusters (**fpc**)
  - `"max.diameter"` Maximum cluster diameter (**fpc**)
  - `"min.separation"` Minimum cluster separation (**fpc**)
  - `"ave.within.cluster.ss"` a generalization of the within-clusters sum of squares (half the sum of the within-cluster squared dissimilarities divided by the cluster size) (**fpc**)
  - `"g2"` Goodman and Kruskal's Gamma coefficient (**fpc**)
  - `"pearsongamma"` Correlation between distances and a 0-1-vector where 0 means same cluster, 1 means different clusters (**fpc**)
  - `"dunn"` Dunn index (minimum separation over maximum diameter) (**fpc**)
  - `"dunn2"` Minimum average dissimilarity between two cluster over maximum average within-cluster dissimilarity (**fpc**)
  - `"entropy"` entropy of the distribution of cluster memberships (**fpc**)
  - `"wb.ratio"` average.within over average.between (**fpc**)
- External evaluation measures.
  - `"precision"`, `"recall"`, `"F1"`. A true positive (TP) decision assigns two points in the same true cluster also to the same cluster, a true negative (TN) decision assigns two points from two different true clusters to two different clusters. A false positive (FP) decision assigns two points from the same true cluster to two different clusters. A false negative (FN) decision assigns two points from the same true cluster to different clusters.

    $$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$
    $$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

    The F1 measure is the harmonic mean of precision and recall.
  - `"purity"` Average purity of clusters. The purity of each cluster is the proportion of the points of the majority true group assigned to it (Cao et al. 2006).
  - `"Euclidean"` Euclidean dissimilarity of the memberships (**clue**),
  - `"Manhattan"` Manhattan dissimilarity of the memberships (**clue**)
  - `"Rand"` Rand index (**clue**)
  - `"cRand"` Rand index corrected for chance (**clue**)

  - `"NMI"` Normalized Mutual Information (**clue**)
  - `"KP"` Katz-Powell index (**clue**)
  - `"angle"` Maximal cosine of the angle between the agreements (**clue**)
  - `"diag"` Maximal co-classification rate (**clue**)
  - `"FM"` Fowlkes and Mallows's index (**clue**)
  - `"Jaccard"` Jaccard index (**clue**)
  - `"PS"` Prediction Strength (**clue**)
  - `"vi"` Variation of Information (VI) index (**fpc**)

- Outlier evaluation measures.

  - `"OutlierJaccard"`. A variant of the Jaccard index that can be applied assess the outlier detection correctness (Krleža et al. 2020). It can be applied only to data streams that mark outliers (for example `DSD_Gaussians`). The most sensible use of this measure is with outlier detectors, e.g., those clusterers that inherit `DSC_Outlier`. However, the `OutlierJaccard` can be calculated for all those clusterers that enclose outliers in small clusters, which are were **not** pruned by `prune_clusters`. A true positive (TP) decision is made for outliers that were both marked by the input data stream generator and found by the clusterer instance, A false positive (FP) decision is made for data points that were **not** marked by the input data stream generator, yet the clusterer instance marked it is an outlier. A undetected (UND) decision is made for outliers that were marked by the input data stream generator, yet the clusterer instance failed to recognized it as an outlier. The `OutlierJaccard` index is calculated as follows:

$$\text{OJI} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{UND}}$$

`evaluate_static()` is appropriate if the data stream does not evolve significantly from the data that is used to learn the clustering to the data that is used for evaluation. The approach described next might be more appropriate for streams which exhibit significant concept drift.

6.3. Evaluation of clustering of dynamic data streams

For dynamic data streams it is important to evaluate how well the clustering algorithm is able to adapt to concept drift which results in changes in the cluster structure. Aggarwal et al. (2003) have introduced an evaluation scheme for data stream clustering which addresses these issues. In this approach a horizon is defined as a number of data points. The data stream is split into consecutive horizons. After a horizon is clustered, the points in the next horizon are each assigned to the closest centroid and the sum of squares is reported as an internal measure of cluster quality. Later on, this scheme was used by others (e.g., by Tu and Chen (2009)). Cao et al. (2006) and Wan et al. (2009) also use this scheme for the external measure of average purity of clusters. Here for each (micro-) cluster the dominant true cluster label is determined and the proportion of points with the dominant label is averaged over all clusters. This type of evaluation strategy is called prequential since new data is always used for evaluation and and afterwards to update the model. Recent detailed analysis of prequential error estimation for classification can be found in the work by Gama et al. (2013) and Bifet, de Francisci Morales,

Read, Holmes, and Pfahringer (2015). Obviously, algorithms which can better adapt to the changing stream will achieve better evaluation values. However, it is important to mention that choosing the horizon inappropriately for the stream may impact the evaluation. Consider, for example, a fast changing stream and a very long horizon. In this case the evaluation data might have not much similarity to the data used for clustering and thus the evaluation will produce meaningless results. For fast evolving streams a shorter horizon, or even a horizon of length one, needs to be used. Longer horizons have the advantage that evaluation can be usually performed more efficiently for larger batches of points.

This prequential evaluation strategy is implemented in **stream** as function `evaluate_stream()`. It shares most parameters with `evaluate_static()` and all evaluation measures for `evaluate_static()` described above can be used.

6.4. Evaluation of clustering done by single-pass clusterers

The single-pass clusterers and outlier detectors are doing classification and model update for every data point retrieved from the input data stream. This makes then more fine-grained. Evaluation of single-pass clusterers is a special case of evaluation for dynamic data streams having $horizon = 1$. For `evaluate_stream(..., horizon=1000, ...)` this means that a single-pass clusterer processes the batch of 1000 data points each data point individually, which includes both the model update and returning the assignment. From the caller point of view, such evaluation is the same as for all other clustering algorithms in the **stream** package.

6.5. Example: Evaluating clustering results

In this example we will show how to calculate evaluation measures, first on a stream without concept drift and then on an evolving stream. First, we prepare a data stream and create a clustering.

```
R> library("stream")
R> stream <- DSD_Gaussians(k = 3, d = 2, noise = .05)
R> dstream <- DSC_DStream(gridsize = .1)
R> update(dstream, stream, n = 2000)
```

The `evaluate_static()` function takes a DSC object containing a clustering and a DSD object with evaluation data to compute several quality measures for clustering.

```
R> evaluate_static(dstream, stream, n = 100)

Evaluation results for micro-clusters.
Points were assigned to micro-clusters.

            numPoints      numMicroClusters      numMacroClusters
            100.00000              13.00000               3.00000
       noisePredicted                   SSQ           silhouette
              6.00000              0.17100               0.27823
     average.between       average.within          max.diameter
              0.40415              0.05829               0.98497
```

```
       min.separation ave.within.cluster.ss                  g2
              0.00180               0.00724             0.90388
         pearsongamma                  dunn               dunn2
              0.48611               0.00182             0.06461
              entropy              wb.ratio          numClasses
              2.27295               0.14423             4.00000
           noiseActual       noisePrecision       outlierJaccard
              3.00000               0.50000             0.50000
            precision                recall                  F1
              0.98120               0.37689             0.54459
               purity             Euclidean           Manhattan
              0.96429               0.33668             0.56000
                 Rand                 cRand                 NMI
              0.80606               0.45081             0.68995
                   KP                 angle                diag
              0.54874               0.56000             0.56000
                   FM               Jaccard                  PS
              0.60811               0.37419             0.33712
                   vi
              1.19403
attr(,"type")
[1] "micro"
attr(,"assign")
[1] "micro"
```

The number of points taken from `dsd` and used for the evaluation are passed on as the parameter `n`. Since no evaluation measure is specified, all available measures are calculated. We use only a small number of points for evaluation since calculating some measures is computational quite expensive. Individual measures can be calculated using the measure argument.

```
R> evaluate_static(dstream, stream, measure = c("purity", "crand"), n = 500)

Evaluation results for micro-clusters.
Points were assigned to micro-clusters.

purity  cRand
 0.974  0.440
attr(,"type")
[1] "micro"
attr(,"assign")
[1] "micro"
```

Note that this second call of `evaluate_static()` uses a new and larger set of 500 evaluation data points from the stream and thus the results may vary slightly from the first call. Purity of the micro-clusters is high since each micro-cluster only covers points from the same true cluster, however, the corrected Rand index is low because several micro-clusters split the

points from each true cluster. We will see in one of the following examples that reclustering will improve the corrected Rand index.

To evaluate how well a clustering algorithm can adapt to an evolving data stream, **stream** provides `evaluate_stream()` to perform prequential evaluation with a given horizon. Each data point in the horizon is assigned to clusters to evaluate how well it fits into the clustering (internal evaluation) or its assignment agrees with the known true cluster labels (external evaluation). Average evaluation measures for each horizon are returned. Afterwards, the clustering is updated with the points in the horizon.

The following examples evaluate D-Stream on an evolving stream created with `DSD_Benchmark`. This data stream was introduced in Figure 6 on page 15 and contains two Gaussian clusters moving from left to right with their paths crossing in the middle. We modify the default decay parameter `lambda` of D-Stream since the data stream evolves relatively quickly and then perform the evaluation over 5000 data points with a horizon of 100.

```
R> set.seed(1000)
R> stream <- DSD_Benchmark(1)
R> dstream <- DSC_DStream(gridsize = .05, lambda = .01)
R> ev <- evaluate_stream(dstream, stream,
+   measure = c("numMicroClusters", "purity"), n = 5000, horizon = 100)
R> head(ev)
```

```
  points numMicroClusters purity
1      0                0  0.470
2    100                6  0.929
3    200                8  0.873
4    300               14  1.000
5    400               10  0.929
6    500                9  0.861
```

Note that the first row in the results contains `NA` for the purity measure. This is the case since we started evaluation with a new, empty clustering and for evaluating the first horizon no prior clusters were available.

```
R> plot(ev[ , "points"], ev[ , "purity"], type = "l",
+   ylab = "Avg. Purity", xlab = "Points")
```

Figure 12 shows the development of the average micro-cluster purity (how well each micro-cluster only represents points of a single group in the ground truth) over 5000 data points in the data stream. Purity drops before point 3000 significantly, because the two true clusters overlap for a short period of time.

To analyze the clustering process, we can visualize the clustering using `animate_cluster()`. To recreate the previous experiment, we reset the data stream and create a new empty clustering.

```
R> set.seed(1000)
R> stream <- DSD_Benchmark(1)
```

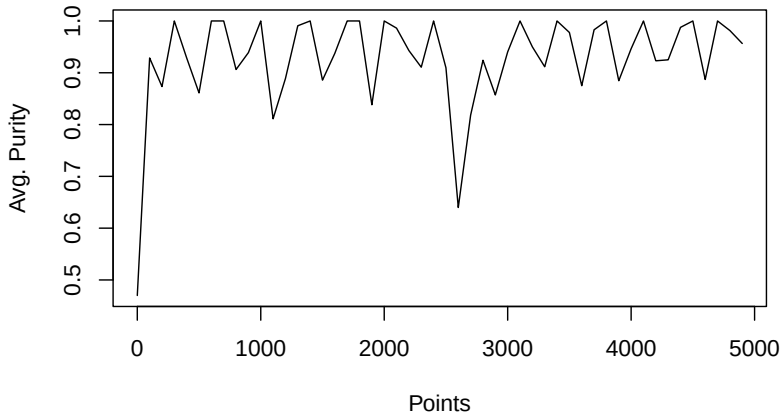


Figure 12: Micro-cluster purity of D-Stream over an evolving stream.

```
R> dstream <- DSC_DStream(gridsize = .05, lambda = .01)
R> r <- animate_cluster(dstream, stream, horizon = 100, n = 5000,
+      measure = "purity", plot.args = list(xlim = c(0, 1), ylim = c(0, 1)))
```

Figure 13 shows the result of the clustering animation with purity evaluation. The whole animation can be recreated by executing the code above. The animation can also be replayed and saved using package **animation**.

6.6. Example: Evaluating reclustered DSC objects

This example shows how to recluster a DSC object after creating it and performing evaluation on the macro clusters. First we create data, a DSC micro-clustering object and cluster 1000 points.

```
R> library("stream")
R> set.seed(1000)
R> stream <- DSD_Gaussians(k = 3, d = 2, noise = .05)
R> dstream <- DSC_DStream(gridsize = .05, Cm = 1.5)
R> update(dstream, stream, n = 1000)
R> dstream

D-Stream
Class: DSC_DStream, DSC_Micro, DSC_R, DSC
Number of micro-clusters: 36
Number of macro-clusters: 3
```

Although the data contains three clusters, the built-in reclustering of D-Stream (joining adjacent dense grids) only produces two macro-clusters. The reason for this can be found by visualizing the clustering.

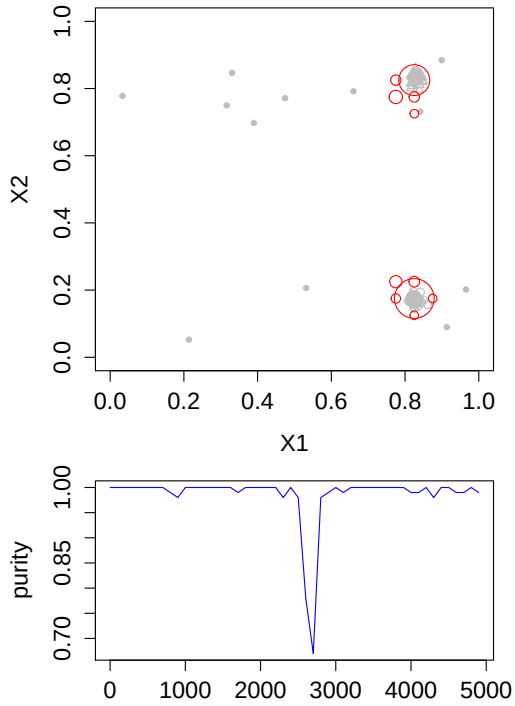


Figure 13: Result of animated clustering with evaluation.

```
R> plot(dstream, stream, type = "both")
```

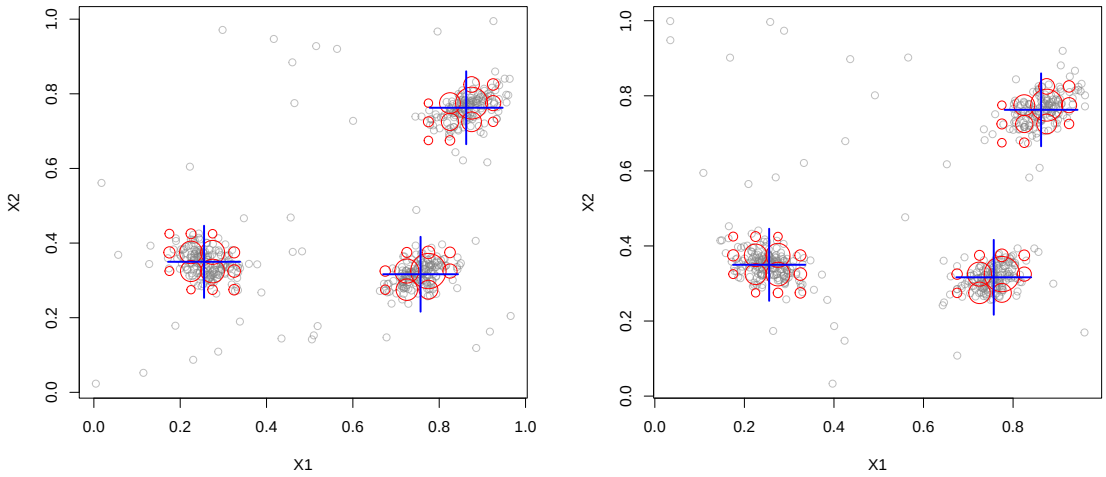
Figure 14(a) shows micro- and macro-clusters produced by D-Stream. Micro-clusters are shown as red circles while macro-clusters are represented by large blue crosses. Cluster symbol sizes are proportional to the cluster weights. We see that D-Stream's reclustering strategy which joins adjacent dense grid cells is not able to separate the two overlapping clusters in the top part of the plot.

Micro-clusters produced with any clustering algorithm can be reclustered by the `recluster()` method with any available macro-clustering algorithm (sub-classes of `DSD_Macro`) available in **stream**. Some supported macro-clustering models that are typically used for reclustering are k-means, hierarchical clustering, and reachability. We use weighted k-means since we want to separate overlapping Gaussian clusters.

```
R> km <- DSC_Kmeans(k = 3, weighted = TRUE)
R> recluster(km, dstream)
R> km

k-Means (weighted)
Class: DSC_Kmeans, DSC_Macro, DSC_R, DSC
Number of micro-clusters: 36
Number of macro-clusters: 3

R> plot(km, stream, type = "both")
```



(a) (b)

Figure 14: A data stream clustered with D-Stream using the (a) built-in reclustering strategy, and (b) reclustered with weighted k-means and $k = 3$.

Figure 14(b) shows that weighted k-means on the micro-clusters produces by D-Stream separated the three clusters correctly.

Evaluation on a macro-clustering model automatically uses the macro-clusters. For evaluation, `n` new data points are requested from the data stream and each is assigned to its nearest micro-cluster. This assignment is translated into macro-cluster assignments and evaluated using the ground truth provided by the data stream generator.

```
R> evaluate_static(km, stream, measure = c("purity", "crand", "SSQ"), n = 1000)

Evaluation results for macro-clusters.
Points were assigned to micro-clusters.

purity  cRand    SSQ
 0.958  0.934  2.751
attr(,"type")
[1] "macro"
attr(,"assign")
[1] "micro"
```

Alternatively, the new data points can also be directly assigned to the closest macro-cluster.

```
R> evaluate_static(km, stream, c(measure = "purity", "crand", "SSQ"), n = 1000,
+   assign = "macro")

Evaluation results for macro-clusters.
Points were assigned to macro-clusters.
```

```
purity  cRand    SSQ
  0.96   0.94   2.72
attr(,"type")
[1] "macro"
attr(,"assign")
[1] "macro"
```

In this case the evaluation measures purity and corrected Rand slightly increase, since D-Stream produces several micro-clusters covering the area between the top two true clusters (see micro-clusters in Figure 14). Each of these micro-clusters contains a mixture of points from the two clusters but has to assign all its points to only one resulting in some error. Assigning the points rather to the macro-cluster centers splits these points better and therefore decreases the number of incorrectly assigned points. The sum of squares decreases because the data points are now directly assigned to minimize this type of error.

Other evaluation methods can also be used with a clustering in **stream**. For example we can calculate and plot silhouette information Kaufman and Rousseeuw (1990) using the functions available in **cluster**. We take 100 data points and find the assignment to macro clusters in the data stream clustering. For a `DSC_Micro` implementation like D-Stream, the data points are assigned by default to micro clusters and then this assignment is translated to macro-cluster assignments.

```
R> points <- get_points(stream, n = 100)
R> assignment <- get_assignment(dstream, points, type = "macro")
R> assignment
```

```
  22 <NA>   18   18   14    3    6   19   18    5 <NA>   29   21    6   15
   2   NA    2    2    1    1    1    2    2    1   NA    3    2    1    2
  13   35   13   21   30   18   32   10    9   32   26   20   13    6   10
   1    3    1    2    3    2    3    1    1    3    2    2    1    1    1
  21    6   22   18    2   32   32    7   31   10   35   20    2   21   20
   2    1    2    2    1    3    3    1    3    1    3    2    1    2    2
  36    6   33   32   17   36   21    5   26   30    6    5    5   17   29
   3    1    3    3    2    3    2    1    2    3    1    1    1    2    3
  14   30   32 <NA> <NA>   17   13    5   21    9   35    6   29   18   13
   1    3    3   NA   NA    2    1    1    2    1    3    1    3    2    1
  22 <NA>    9    6 <NA>   29   28 <NA> <NA> <NA>    3   29   17   22    8
   2   NA    1    1   NA    3    3   NA   NA   NA    1    3    2    2    1
  36   32   18    5    9   10   31 <NA>   21   18
   3    3    2    1    1    1    3   NA    2    2
attr(,"method")
[1] "model"
```

Note that D-Stream uses a grid for assignment and that points which do not fall inside a dense (or connected transitional) cell are not assigned to a cluster represented by a value of `NA`. For the following silhouette calculation we replace the `NA`s with 0 to make the unassigned

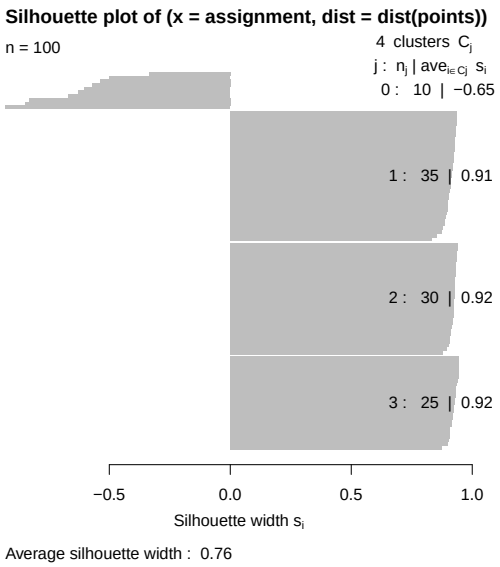


Figure 15: Silhouette plot for D-Stream clustering with two macro-clusters and a cluster ($j = 0$) representing the unassigned points.

(noise) points its own cluster. Note also that the silhouette is only calculated for a small number of points and not the whole stream.

```
R> assignment[is.na(assignment)] <- 0L
R> library("cluster")
R> plot(silhouette(assignment, dist = dist(points)))
```

Figure 15 shows the silhouette plot for the macro-clusters produced by D-Stream. The top cluster ($j = 0$) represents the points not assigned to any cluster by the algorithm (predicted noise) and thus is expected to have a large negative silhouette. Cluster $j = 2$ comprises the two overlapping real clusters and thus has lower silhouette values than cluster $j = 1$. Other visual evaluation methods can be used in a similar way.

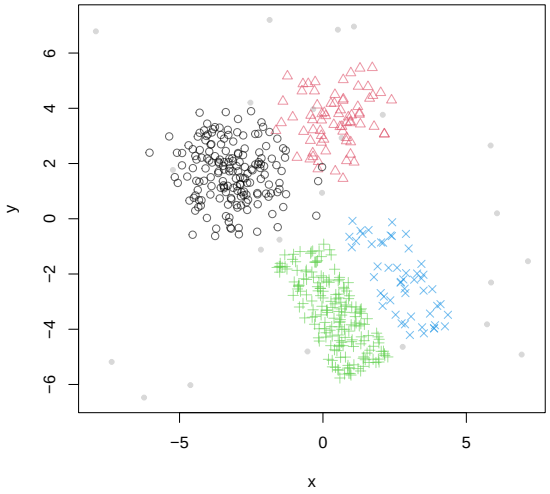


Figure 16: Bars and Gaussians data set.

7. Example applications

7.1. Experimental comparison of different algorithms

Providing a framework for rapid prototyping new data stream mining algorithms and comparing them experimentally is the main purpose of **stream**. In this section we give a more elaborate example of how to perform a comparison between several algorithms.

First, we set up a static data set. We extract 1500 data points from the Bars and Gaussians data stream generator with 5% noise and put them into a `DSD_Memory`. This object is used to replay the same part of the data stream for each algorithm. We will use the first 1000 points to learn the clustering and the remaining 500 points for evaluation.

```
R> set.seed(1000)
R> library("stream")
R> stream <- DSD_BarsAndGaussians(noise = .05) %>% DSD_Memory(n = 1500)
R> stream

Memorized Stream for Bars and Gaussians (d = 2, k = 4)
Class: DSD_Memory, DSD_R, DSD
Contains 1500 data points - currently at position 1 - loop is FALSE

R> plot(stream)
```

Figure 16 shows the structure of the data set. It consists of four clusters, two Gaussians and two uniformly filled, slightly rotated rectangular clusters. The Gaussian and the bar to the right have 1/3 the density of the other two clusters.

We initialize four algorithms from **stream**. We choose the parameters experimentally so that the algorithms produce each approximately 100 micro-clusters.

```
R> algorithms <- list(
+   'Sample' = DSC_TwoStage(micro = DSC_Sample(k = 100),
+     macro = DSC_Kmeans(k = 4)),
+   'Window' = DSC_TwoStage(micro = DSC_Window(horizon = 100),
+     macro = DSC_Kmeans(k = 4)),
+   'D-Stream' = DSC_DStream(gridsize = .7, Cm = 1.5),
+   'DBSTREAM' = DSC_DBSTREAM(r = .45)
+ )
```

The algorithms are reservoir sampling reclustered with weighted k-means, sliding window reclustered with weighted k-means, D-Stream and DBSTREAM with their built-in reclustering strategies. We store the algorithms in a list for easier handling and then cluster the same 1000 data points with each algorithm. Note that we have to reset the stream each time before we cluster with a new algorithm.

```
R> for(a in algorithms) {
+   reset_stream(stream)
+   update(a, stream, n = 1000)
+ }
```

We use `nclusters()` with `type="micro"` to inspect the number of micro-clusters.

```
R> sapply(algorithms, nclusters, type = "micro")
```

```
  Sample   Window D-Stream DBSTREAM
     100      100       84       99
```

To inspect micro-cluster placement, we plot the calculated micro-clusters on a sample of the original data.

```
R> op <- par(no.readonly = TRUE)
R> layout(mat = matrix(1:length(algorithms), ncol = 2))
R> for (a in algorithms) {
+   reset_stream(stream)
+   plot(a, stream, main = description(a), type = "micro")
+ }
R> par(op)
```

Figure 17 shows the micro-cluster placement by the different algorithms. Micro-clusters are shown as red circles and the size is proportional to each cluster's weight. Reservoir sampling and the sliding window select some data points as micro-clusters and also include a few noise points. D-Stream and DBSTREAM suppress noise well and concentrate the micro-clusters on the real clusters. D-Stream is grid-based and thus the micro-clusters are regularly spaced. DBSTREAM produces a slightly less regular pattern.

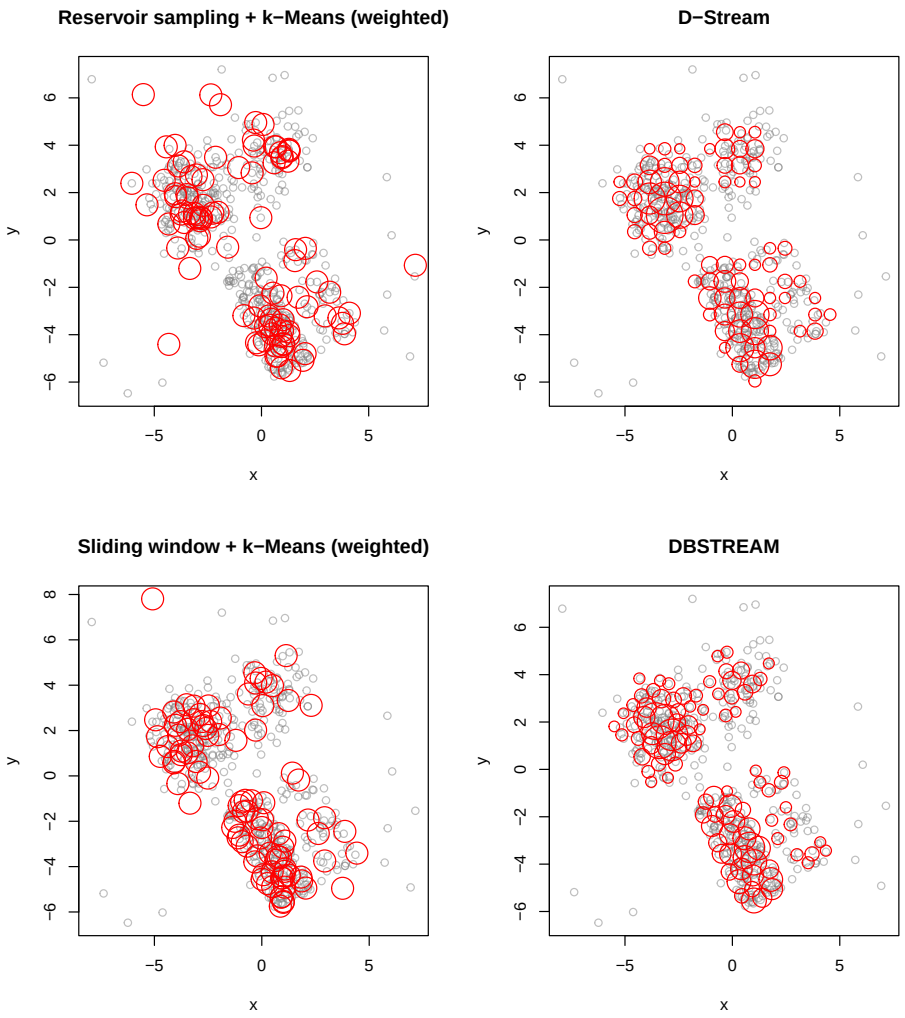


Figure 17: Micro-cluster placement for different data stream clustering algorithms.

It is also interesting to compare the assignment areas for micro-clusters created by different algorithms. The assignment area is the area around the center of a micro-cluster in which points are considered to belong to the micro-cluster. The specific clustering algorithm decides how points which fall inside the assignment area of several micro-clusters are assigned (e.g., assign the point to the closest center). To show the assignment area we add `assignment = TRUE` to plot. We also disable showing micro-cluster weights to make the plot less cluttered.

```
R> op <- par(no.readonly = TRUE)
R> layout(mat = matrix(1:length(algorithms), ncol = 2))
R> for (a in algorithms) {
+   reset_stream(stream)
+   plot(
+     a,
+     stream,
```

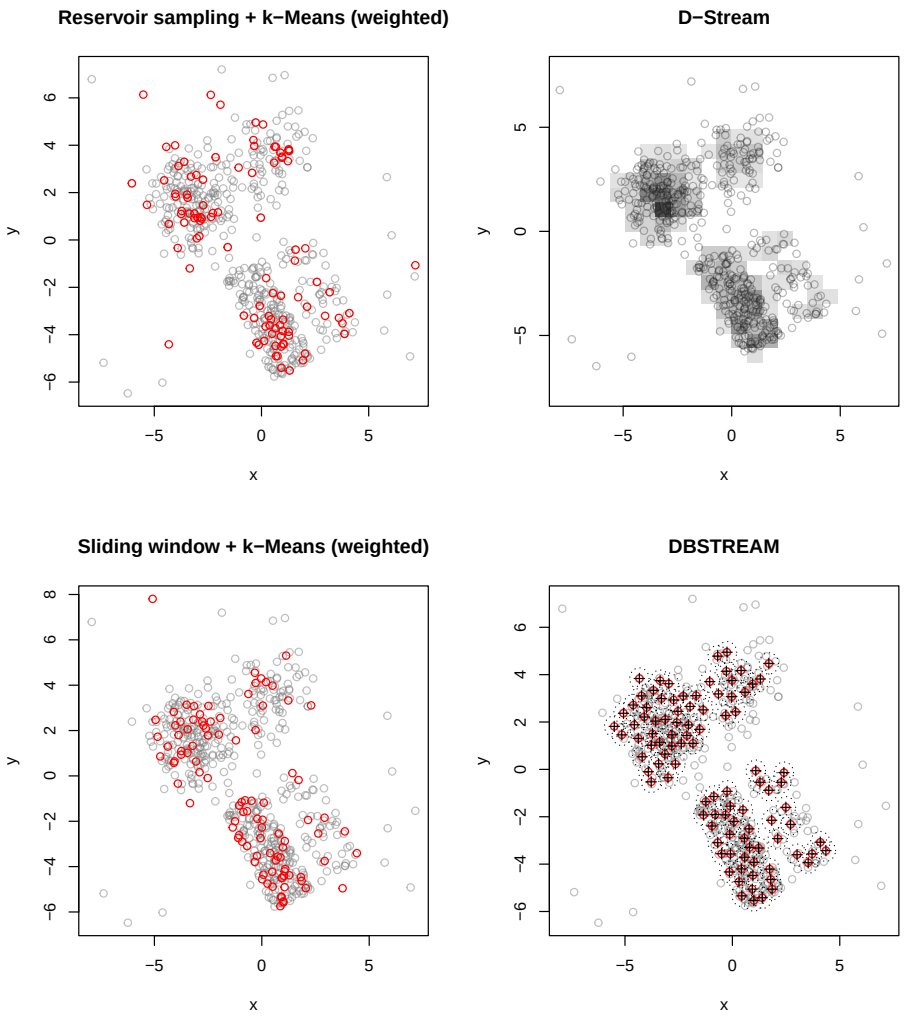


Figure 18: Micro-cluster assignment areas for different data stream clustering algorithms.

```
+     main = description(a),
+     assignment = TRUE,
+     weight = FALSE,
+     type = "micro"
+   )
+ }
R> par(op)
```

Figure 18 shows the assignment areas. For regular micro-cluster-based algorithms the assignment areas are shown as dotted circles around micro-cluster centers. For example for DBSTREAM the assignment area for all micro-clusters has exactly the same radius. D-Stream uses a grid for assignment and thus shows the grid. Reservoir sampling and sliding window does not have assignment areas and data points are always assigned to the nearest micro-cluster.

To compare the cluster quality, we can check for example the micro-cluster purity. Note that we set the stream to position 1001 since we have used the first 1000 points for learning and we want to use data points not seen by the algorithms for evaluation.

```
R> sapply(
+   algorithms,
+   FUN = function(a) {
+     reset_stream(stream, pos = 1001)
+     evaluate_static(
+       a,
+       stream,
+       measure = c("numMicroClusters", "purity"),
+       type = "micro",
+       n = 500
+     )
+   }
+ )
```

```
                  Sample  Window D-Stream DBSTREAM
numMicroClusters 100.000 100.000   84.000   99.000
purity             0.944   0.947    0.966    0.971
```

We need to be careful with the comparison of these numbers, since they depend heavily on the number of micro-clusters with more clusters leading to a better value. We can compare purity here since we have set the clustering parameters such that the number of micro-clusters is very close. All algorithms produce very good values for purity for this data set with reasonably well separated clusters.

Next, we compare macro-cluster placement. D-Stream and DBSTREAM have built-in reclustering strategies. D-Stream joins adjacent dense grid cells to form macro-clusters and DBSTREAM joins micro-clusters reachable by overlapping assignment areas. For sampling and sliding window we already have created a two-stage process together with weighted k-means ($k = 4$).

```
R> op <- par(no.readonly = TRUE)
R> layout(mat = matrix(1:length(algorithms), ncol = 2))
R> for (a in algorithms) {
+   reset_stream(stream)
+   plot(a, stream, main = description(a))
+ }
R> par(op)
```

Figure 19 shows the macro-cluster placement. Sampling and the sliding window use k-means reclustering and therefore produce exactly four clusters. However, the placement is off, splitting a true cluster and missing one of the less dense clusters. D-Stream and DBSTREAM identify the two denser clusters correctly, but split the lower density clusters into multiple pieces.

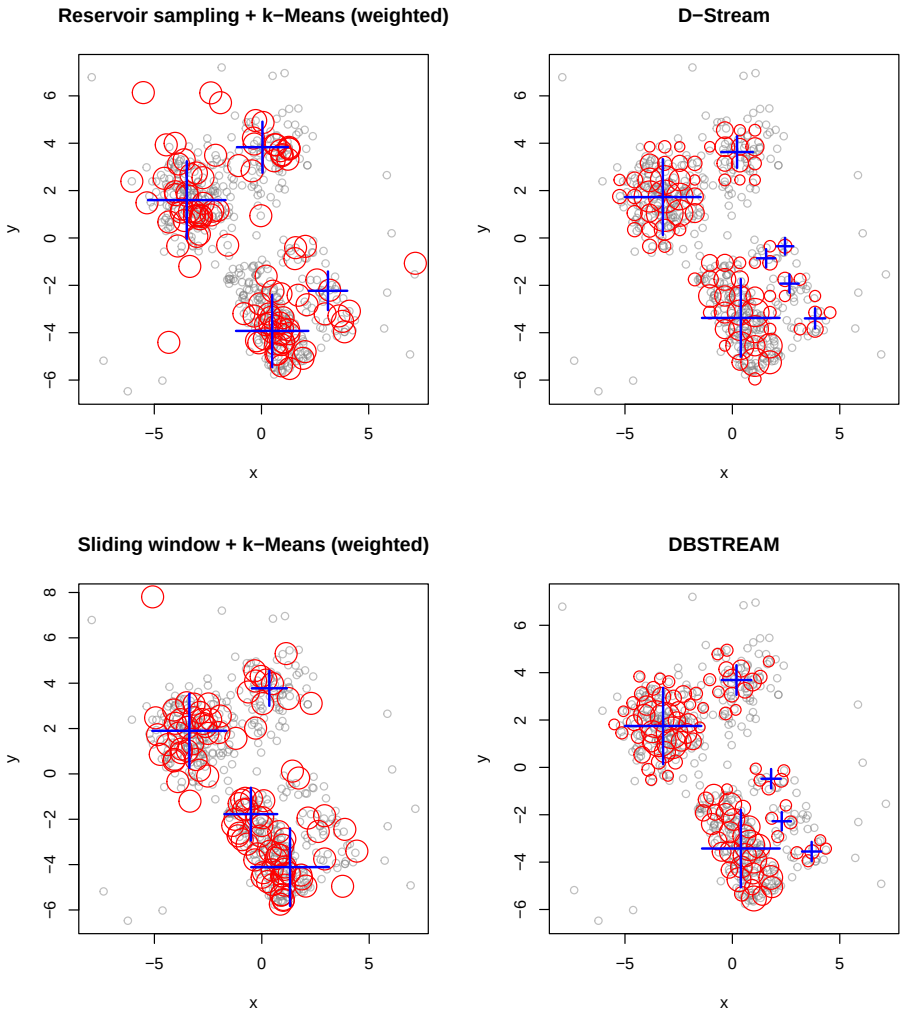


Figure 19: Macro-cluster placement for different data stream clustering algorithms.

```
R> sapply(algorithms, FUN = function(a) {
+   reset_stream(stream, pos = 1001)
+   evaluate_static(a, stream, measure = c("numMacroClusters", "purity",
+       "SSQ", "cRand", "silhouette"),
+     n = 500, assign = "micro", type = "macro")
+ })
```

```
                    Sample  Window D-Stream DBSTREAM
numMacroClusters     4.000   4.000    7.000    6.000
purity               0.898   0.823    0.893    0.876
SSQ               1111.975 976.861  866.966  885.504
cRand                0.846   0.673    0.782    0.795
silhouette           0.456   0.466    0.305    0.294
```

The evaluation measures at the macro-cluster level reflect the findings from the visual analysis

of the clustering with D-Stream and DBSTREAM producing the best results. Note that D-Stream and DBSTREAM do not assign some points which are not noise points which has a negative effect on the average silhouette width.

Comparing algorithms on evolving streams is similarly easy in **stream**. For the following example we use again `DSD_Benchmark` with two moving clusters crossing each other's path (see Section 4.2). First we create a fixed stream with 5000 data points.

```
R> set.seed(0)
R> stream <- DSD_Memory(DSD_Benchmark(1), n = 5000)
```

Next we initialize again a list of clustering algorithms. Note that this time we use a k of two for reclustering sampling and the sliding window. We also use a sample biased to newer data points (Aggarwal 2006) since otherwise outdated data points would result in creating outdated clusters. For the sliding window, D-Stream and DBSTREAM we use faster decay (`lambda=.01`) since the clusters in the data stream move very quickly.

```
R> algorithms <- list(
+   'Sample + k-means' = DSC_TwoStage(micro = DSC_Sample(k = 100, biased = TRUE),
+     macro = DSC_Kmeans(k = 2)),
+   'Window + k-means' = DSC_TwoStage(micro = DSC_Window(horizon = 100, lambda = .01),
+     macro = DSC_Kmeans(k = 2)),
+   'D-Stream' = DSC_DStream(gridsize = .1, lambda = .01),
+   'DBSTREAM' = DSC_DBSTREAM(r = .05, lambda = .01)
+ )
```

We apply `evaluate_stream()` to each of the clustering algorithms, and we evaluate and cluster 5000 data points using the prequential evaluation method with a horizon of 250 points. The chosen evaluation measure is the corrected Rand index. This produces a list with $5000/250 = 20$ evaluations for each algorithm.

```
R> evaluation <- lapply(algorithms, FUN = function(a) {
+   reset_stream(stream)
+   evaluate_stream(a, stream, horizon = 100, n = 5000, measure = "cRand",
+     type = "macro", assign = "micro")
+ })
```

To plot the results we first get the positions at which the evaluation measure was calculated from the first element in the evaluation list and then extract a matrix with the corrected Rand index values. Note that the first evaluation values are again `NA` since we start with empty clusterings.

```
R> cRand <- sapply(evaluation, FUN = function(x) x[ , "cRand"])
R> head(cRand)
```

```
     Sample + k-means Window + k-means D-Stream DBSTREAM
[1,]            0.000             0.000    0.000    0.000
```

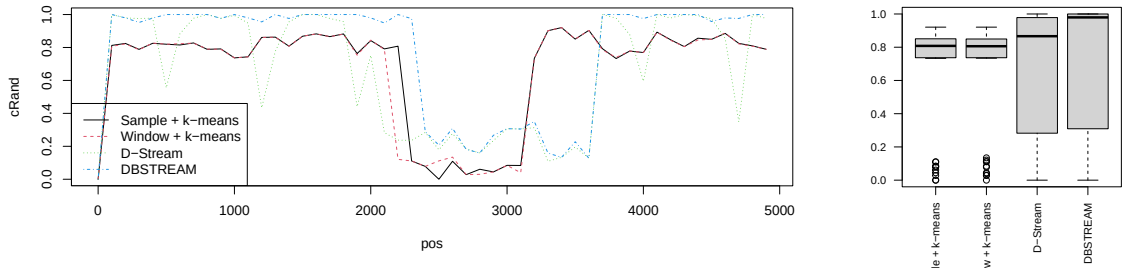


Figure 20: Evaluation of data stream clustering of an evolving stream.

```
[2,]          0.813          0.807    1.000    1.000
[3,]          0.825          0.822    0.980    0.980
[4,]          0.789          0.787    0.976    0.952
[5,]          0.825          0.827    0.978    0.978
[6,]          0.820          0.820    0.555    1.000
```

We visualize the development of the evaluation measure over the stream as a line plot and we add a boxplot comparing the distributions.

```
R> pos <- evaluation[[1]][ , "points"]
R> matplot(pos, cRand, type = "l", lwd = 1)
R> legend("bottomleft",  legend = names(evaluation),
+   col = 1:6, lty = 1:6, lwd = 1)

R> boxplot(cRand, las = 2, cex.axis = .8)
```

Figure 20 shows the corrected Rand index for the four data stream clustering algorithms over the evolving data stream. All algorithms show that separating the two clusters is impossible around position 3000 when the two clusters overlap. D-Stream and DBSTREAM perform equally well while biased sampling and the sliding window achieve only a lower corrected Rand index. This is easily explained by the fact that these two algorithms cannot detect noise and thus have to assign noise points to one of the clusters resulting in the lower Rand index. The behavior of the individual clustering algorithms can be visually analyzed using `animate_cluster()`.

The **stream** framework allows us to easily create many experiments by using different data and by matching different clustering and reclustering algorithms. An example of a study for clustering large data sets using an earlier version of **stream** can be found in Bolaños, Forrest, and Hahsler (2014).

7.2. Clustering a real data set

In this section we show how to cluster the well-known and widely used KDD Cup'99 data set. The data set was created for the Third International Knowledge Discovery and Data Mining Tools Competition and contains simulated network traffic with a wide variety of intrusions.

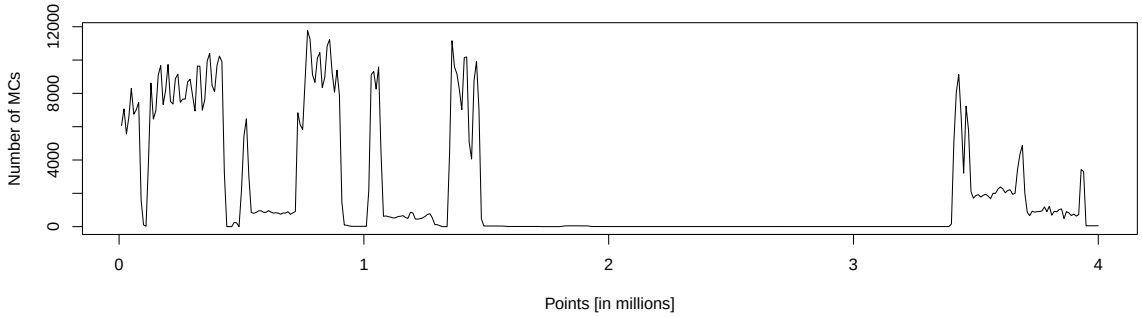
The data set contains 4,898,431 data points and we use the 34 numeric features for clustering. The data set is available from the UCI Machine Learning Repository (Bache and Lichman 2013) and we directly stream the data from there. We use the first 1000 data points to center and scale the observations in the data stream in flight.

```
R> library("stream")
R> con <- gzcon(
+   url(paste0("http://archive.ics.uci.edu/ml/machine-learning-databases/",
+     "kddcup99-mld/kddcup.data.gz")))
R> stream <- DSD_ReadCSV(con, take=c(1, 5, 6, 8:11, 13:20, 23:42),
+     class = 42, k = 7)
R> stream2 <- DSD_ScaleStream(stream, n = 1000)
```

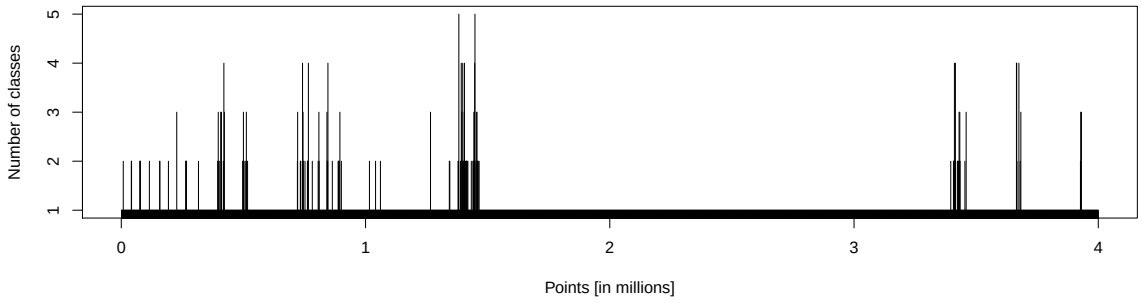
Next, we set up D-Stream with slightly modified values for gaptime (increased number of points after which obsolete micro-clusters are removed) and lambda (faster fading), and cluster the next 4 million data points.

```
R> dstream <- DSC_DStream(gridsize = .5, gaptime = 10000L, lambda = .01)
R> update(dstream, stream2, n = 4000000, verbose = TRUE)
```

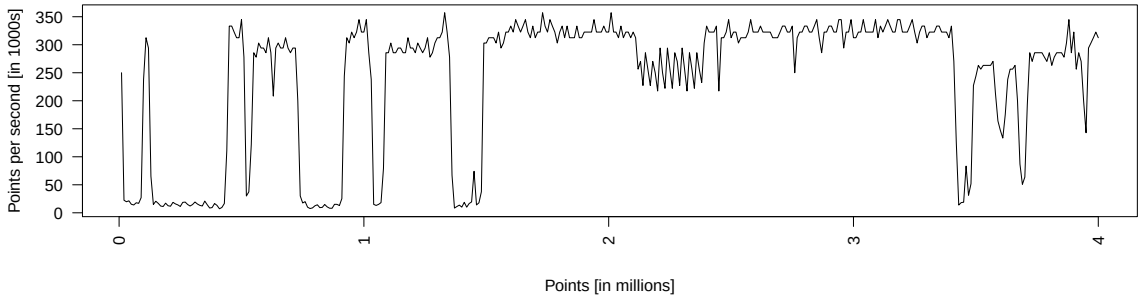
In stream clustering, each data point is processed individually and we have recorded some key statistics averaged over 1000 point intervals. Figure 21(a) shows the number of micro-clusters used by the algorithm. This number is directly related to the memory used by the algorithm. For the used 34 dimensional data set, each micro-cluster occupies 416 bytes of storage leading to a maximal memory requirement of less than 5MB (a maximum of 12,039 micro-clusters are used at the end of the first quarter of the stream) for this clustering. The number of micro-clusters varies significantly over the stream. This behavior can be explained by the changes in the distribution of the data. Figure 21(b) shows the number of different classes (normal and different types of intrusions) in each 1000 point segment. It is easy to see that the number of micro-clusters is directly related to the number of different classes in the data. Figure 21(c) reports the clustering speed in number of points per second. We use here R 3.1.2 on Linux 3.16.0-28 with an Intel i5 processor at 1.9GHz and 8GB of memory, and the algorithm is implemented as a mixture of R and C++ code using the **Rcpp** interface package (Eddelbuettel and François 2011; Eddelbuettel 2013). The speed varies significantly between 7,559 and 384,600 points per second with an average throughput of 280,200 points per second (this measure excludes delays caused by the network connection). The throughput remains very high for a long stretch between point 1.5 and 3.5 million. It is easy to see that the performance is inversely related to the number of micro-clusters since more micro-clusters increase the search time for updates. Clustering the 4 million data points took a total of 65 seconds. In comparison, k-means clustering using `kmeans` (in package **stats**) with eight clusters (number of classes) took 186 seconds and used at its peek 80% of 8GB of the available main memory (the whole dataset is stored in memory).



(a)



(b)



(c)

Figure 21: Clustering 4 million data points of the KDD Cup'99 data set with D-Stream.

8. Conclusion and future work

Package **stream** is a data stream modeling framework for R that provides both, a variety of data stream generation tools as well as a component for performing data stream mining tasks. The flexibility offered by the framework allows the user to create a multitude of easily reproducible experiments to compare the performance of these tasks. While R is not an ideal environment to process high-throughput streams in real-time, **stream** provides an infrastructure to develop and test these algorithms. **stream** can be directly used for applications where new points are produced at slower speeds (less than 100,000 points per second depending on the algorithm). Another important application of **stream** is for processing data point by point which otherwise would not fit into main memory.

The presented infrastructure can be extended by adding new data sources and algorithms, or by defining whole new data stream mining tasks. We have abstracted each component to only require a small set of functions that are defined in each base class. Writing the framework in R means that developers have the ability to design components either directly in R, or implement components in Java, Python or C/C++, and then write a small R wrapper as we did for some MOA algorithms in **streamMOA**. This approach makes it easy to experiment with a multitude of algorithms in a consistent way.

Currently, **stream** focuses on the data stream clustering and outlier detection tasks, but we are working on incorporating classification (incorporating the algorithms interfaced by **RMOA** (Wijffels 2014)) and frequent pattern mining algorithms as an extension of the base DST class.

Acknowledgments

Matthew Bolaños and John Forrest worked on **stream** when they were undergraduate students at the Lyle School of Engineering at SMU. Both were supported in part by the U.S. National Science Foundation as a research experience for undergraduates (REU) under contract number IIS-0948893. Part of this work was also supported by the National Human Genome Research Institute under contract number R21HG005912.

References

Adler D, Gläser C, Nenadic O, Oehlschlägel J, Zucchini W (2014). ff: Memory-efficient Storage of Large Data on Disk and Fast Access Functions. R package version 2.2-13, URL https://CRAN.R-project.org/package=ff.

Aggarwal C (ed.) (2007). Data Streams – Models and Algorithms. Springer-Verlag.

Aggarwal CC (2006). "On Biased Reservoir Sampling in the Presence of Stream Evolution." In Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, pp. 607–618. VLDB Endowment.

Aggarwal CC, Han J, Wang J, Yu PS (2003). "A Framework for Clustering Evolving Data Streams." In Proceedings of the International Conference on Very Large Data Bases (VLDB '03), pp. 81–92.

Aggarwal CC, Han J, Wang J, Yu PS (2004). "On Demand Classification of Data Streams." In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '04, pp. 503–508. ACM, New York, NY, USA.

Agrawal R, Imielinski T, Swami A (1993). "Mining Association Rules between Sets of Items in Large Databases." In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 207–216. Washington D.C.

Babcock B, Babu S, Datar M, Motwani R, Widom J (2002). "Models and Issues in Data Stream Systems." In Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, pp. 1–16. ACM, New York, NY, USA.

Bache K, Lichman M (2013). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml.

Bar R (2014). factas: Data Mining Methods for Data Streams. R package version 2.3, URL https://CRAN.R-project.org/package=factas.

Barbera P (2014). streamR: Access to Twitter Streaming API via R. R package version 0.2.1, URL https://CRAN.R-project.org/package=streamR.

Bifet A, de Francisci Morales G, Read J, Holmes G, Pfahringer B (2015). "Efficient Online Evaluation of Big Data Stream Classifiers." In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '15, pp. 59–68. ACM. ISBN 978-1-4503-3664-2.

Bifet A, Holmes G, Kirkby R, Pfahringer B (2010). "MOA: Massive Online Analysis." Journal of Machine Learning Research, **99**, 1601–1604. ISSN 1532-4435.

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

Cao F, Ester 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, pp. 328–339. SIAM.

Chambers JM, Hastie TJ (1992). Statistical Models in S. Chapman & Hall. ISBN 9780412830402.

Charest L, Harrington J, Salibian-Barrera M (2012). birch: Dealing With Very Large Datasets Using BIRCH. R package version 1.2-3, URL https://CRAN.R-project.org/package=birch.

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, KDD '07, pp. 133–142. ACM, New York, NY, USA. doi:10.1145/1281192.1281210.

Cheng J, Ke Y, Ng W (2008). "A Survey on Algorithms for Mining Frequent Itemsets Over Data Streams." Knowledge and Information Systems, **16**(1), 1–27. doi:10.1007/s10115-007-0092-4.

Domingos P, Hulten G (2000). "Mining High-speed Data Streams." In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '00, pp. 71–80. ACM, New York, NY, USA.

Eddelbuettel D (2013). Seamless R and C++ Integration with Rcpp. Springer, New York. ISBN 978-1-4614-6867-7.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." Journal of Statistical Software, **40**(8), 1–18. URL https://doi.org/10.18637/jss.v040.i08.

Ester M, Kriegel HP, Sander J, Xu X (1996). "A Density-based Algorithm for Discovering Clusters in Large Spatial Databases With Noise." In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'1996), pp. 226–231.

Fowler M (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3 edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0321193687.

Gaber M, Zaslavsky A, Krishnaswamy S (2007). "A Survey of Classification Methods in Data Streams." In C Aggarwal (ed.), Data Streams – Models and Algorithms. Springer-Verlag.

Gaber MM, Zaslavsky A, Krishnaswamy S (2005). "Mining Data Streams: A Review." SIGMOD Rec., **34**, 18–26.

Gama J (2010). Knowledge Discovery from Data Streams. 1st edition. Chapman & Hall/CRC, Boca Raton, FL. ISBN 1439826110, 9781439826119.

Gama J, Sebastião R, Rodrigues PP (2013). "On evaluating stream learning algorithms." Machine Learning, pp. 317–346.

Gentry J (2015). twitteR: R Based Twitter Client. R package version 1.1.9, URL https://CRAN.R-project.org/package=twitteR.

Hahsler M (2017). dbscan: Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms. R package version 1.0-0, URL https://CRAN.R-project.org/package=dbscan.

Hahsler M, Bolanos M (2015). streamMOA: Interface for MOA Stream Clustering Algorithms. R package version 1.1-2, URL https://CRAN.R-project.org/package=streamMOA.

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. ISSN 1041-4347. doi:10.1109/TKDE.2016.2522412.

Hahsler M, Bolaños M, Forrest J (2017a). "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R." Journal of Statistical Software, **76**(14), 1–50. doi:10.18637/jss.v076.i14.

Hahsler M, Bolaños M, Forrest J (2017b). stream: Infrastructure for Data Stream Mining. R package version 1.2-4, URL https://CRAN.R-project.org/package=stream.

Hahsler M, Dunham MH (2010). "rEMM: Extensible Markov Model for Data Stream Clustering in R." Journal of Statistical Software, **35**(5), 1–31. URL https://doi.org/10.18637/jss.v035.i05.

Hahsler M, Dunham MH (2015). rEMM: Extensible Markov Model for Data Stream Clustering in R. R package version 1.0-11., URL https://CRAN.R-project.org/.

Hastie T, Tibshirani R, Friedman J (2001). The Elements of Statistical Learning (Data Mining, Inference and Prediction). Springer-Verlag.

Hennig C (2014). fpc: Flexible procedures for clustering. R package version 2.1-7, URL https://CRAN.R-project.org/package=fpc.

Hornik K (2017). clue: Cluster Ensembles. R package version 0.3-53., URL https://CRAN.R-project.org/package=clue.

Jain AK, Dubes RC (1988). Algorithms for Clustering Data. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-022278-X.

Jain AK, Murty MN, Flynn PJ (1999). "Data Clustering: A Review." ACM Computer Surveys, **31**(3), 264–323.

Jin R, Agrawal G (2007). "Frequent Pattern Mining in Data Streams." In C Aggarwal (ed.), Data Streams – Models and Algorithms. Springer-Verlag.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." Journal of Statistical Software, **55**(14), 1–19. URL https://doi.org/10.18637/jss.v055.i14.

Kaptein M (2013). RStorm: Simulate and Develop Streaming Processing in R. R package version 0.902, URL https://CRAN.R-project.org/package=RStorm.

Kaptein M (2014). "RStorm: Developing and Testing Streaming Algorithms in R." The R Journal, **6**(1), 123–132. URL https://doi.org/10.32614/RJ-2014-012.

Kaufman L, Rousseeuw PJ (1990). Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley & Sons, New York.

Keller-McNulty S (ed.) (2004). Statistical Analysis of Massive Data Streams: Proceedings of a Workshop. Committee on Applied and Theoretical Statistics, National Research Council, National Academies Press, Washington, DC.

Kontaki M, Gounaris A, Papadopoulos AN, 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 (2009). "Self-Adaptive Anytime Stream Clustering." In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM '09, pp. 249–258. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3895-2.

Kranen P, Kremer H, Jansen T, Seidl T, Bifet A, Holmes G, Pfahringer B (2010). "Benchmarking Stream Clustering Algorithms within the MOA Framework." In 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2010), Washington, DC, USA.

Kremer H, Kranen P, Jansen T, Seidl T, Bifet A, Holmes G, Pfahringer B (2011). "An Effective Evaluation Measure for Clustering on Evolving Data Streams." In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '11, pp. 868–876. ACM, New York, NY, USA. ISBN 978-1-4503-0813-7. doi:10.1145/2020408.2020555.

Krleža D, Vrdoljak B, Brčić M (2020). "Statistical hierarchical clustering algorithm for outlier detection in evolving data streams." Machine Learning. doi:10.1007/s10994-020-05905-4. URL https://doi.org/10.1007/s10994-020-05905-4.

Last M (2002). "Online Classification of Nonstationary Data Streams." Intelligent Data Analysis, **6**, 129–147. ISSN 1088-467X.

Leisch F, Dimitriadou E (2012). mlbench: Machine Learning Benchmark Problems. R package version 2.1-1, URL https://CRAN.R-project.org/package=mlbench.

Leydold J (2015). rstream: Streams of Random Numbers. R package version 1.3.4, URL https://CRAN.R-project.org/package=rstream.

Maechler M, Rousseeuw P, Struyf A, Hubert M, Hornik K (2014). cluster: Cluster Analysis Basics and Extensions. R package version 1.15.2, URL https://CRAN.R-project.org/package=cluster.

McLeod A, Bellhouse D (1983). "A Convenient Algorithm for Drawing a Simple Random Sample." Applied Statistics, **32**(2), 182–184.

Meyer D, Buchta C (2017). proxy: Distance and Similarity Measures. R package version 0.4-17, URL https://CRAN.R-project.org/package=proxy.

Qiu W, Joe H (2015). clusterGeneration: Random Cluster Generation. R package version 1.3.4, URL https://CRAN.R-project.org/package=clusterGeneration.

R Core Team (2014). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

R Foundation (2011). R Data Import/Export. Version 2.13.1 (2011-07-08), URL https://CRAN.R-project.org/doc/manuals/R-data.html.

R Special Interest Group on Databases (2014). DBI: R Database Interface. R package version 0.3.1, URL https://CRAN.R-project.org/package=DBI.

Rosenberg DS (2012). HadoopStreaming: Utilities for Using R Scripts in Hadoop Streaming. R package version 0.2, URL https://CRAN.R-project.org/package=HadoopStreaming.

Ryan JA (2016). quantmod: Quantitative Financial Modelling Framework. R package version 0.4-7, URL https://CRAN.R-project.org/package=quantmod.

Sevcikova H, Rossini T (2012). rlecuyer: R Interface to RNG With Multiple Streams. R package version 0.3-3, URL https://CRAN.R-project.org/package=rlecuyer.

Silva JA, Faria ER, Barros RC, Hruschka ER, Carvalho A, Gama J (2013). "Data Stream Clustering: A Survey." ACM Computer Surveys, **46**(1), 13:1–13:31. ISSN 0360-0300. doi:10.1145/2522968.2522981.

Tu L, Chen Y (2009). "Stream Data Clustering Based on Grid Density and Attraction." ACM Transactions on Knowledge Discovery from Data, **3**(3), 12:1–12:27. ISSN 1556-4681.

Urbanek S (2016). rJava: Low-level R to Java interface. R package version 0.9-8, URL https://CRAN.R-project.org/package=rJava.

Venables WN, Ripley BD (2002). Modern Applied Statistics with S. Fourth edition. Springer-Verlag, New York. ISBN 0-387-95457-0, URL http://www.stats.ox.ac.uk/pub/MASS4.

Vijayarani S, Sathya P (2012). "A Survey on Frequent Pattern Mining Over Data Streams." International Journal of Computer Science and Information Technology & Security, **2**(5), 1046–1050. ISSN 2249-9555.

Vitter JS (1985). "Random Sampling With a Reservoir." ACM Transactions on Mathematical Software, **11**(1), 37–57. ISSN 0098-3500. doi:10.1145/3147.3165.

Wan L, Ng WK, Dang XH, Yu PS, Zhang K (2009). "Density-based Clustering of Data Streams at Multiple Resolutions." ACM Transactions on Knowledge Discovery from Data, **3**, 14:1–14:28. ISSN 1556-4681.

Wijffels J (2014). RMOA: Connect R with MOA to perform streaming classifications. R package version 1.0, URL https://CRAN.R-project.org/package=RMOA.

Witten IH, Frank E (2005). Data Mining: Practical Machine Learning Tools and Techniques. The Morgan Kaufmann Series in Data Management Systems, 2nd edition. Morgan Kaufmann Publishers. ISBN 0-12-088407-0.

Xie Y (2015). animation: A Gallery of Animations in Statistics and Utilities to Create Animations. R package version 2.4, URL https://CRAN.R-project.org/package=animation.

Zhang T, Ramakrishnan R, Livny M (1996). "BIRCH: An Efficient Data Clustering Method for Very Large Databases." SIGMOD Rec., **25**(2), 103–114. ISSN 0163-5808. doi:10.1145/235968.233324.

Zhu Y, Shasha D (2002). "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time." In Proceedings of the 28th International Conference on Very Large Data Bases, VLDB '02, pp. 358–369. VLDB Endowment.

Affiliation:

Michael Hahsler
Computer Science
Lyle School of Engineering
Southern Methodist University
P.O. Box 750122
Dallas, TX 75275-0122
E-mail: mhahsler@lyle.smu.edu
URL: https://michael.hahsler.net

Matthew Bolaños
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-7329
E-mail: mbolanos@curiouscrane.com

John Forrest
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-7329
E-mail: jforrest@microsoft.com