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

Matthew Bolaños
Southern Methodist University

John Forrest
Microsoft

Michael Hahsler
Southern Methodist University

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 generation process. Common data mining tasks associated with data streams include clustering, classification and frequent pattern mining. New algorithms are proposed regularity and it is important to evaluate them thoroughly under standardized conditions.

In this paper we introduce **stream**, a general purpose tool that includes modeling and simulating data streams as well an extensible framework for implementing, interfacing and experimenting with algorithms for various data stream mining tasks. In this paper we describe the architecture of **stream** and and focus on its use for data stream clustering. **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 as well.

Keywords: data stream, data mining, clustering.

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 is not static, but are the result of a continuous data generation process which might even 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 a 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 started to see 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 can be 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 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 generation 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 streaming data. The following are some packages on CRAN related to streams:

Data sources: Random numbers are typically created as a stream (see e.g., **rstream** (Leydold 2012) and **rlecuyer** (Sevcikova and Rossini 2012)). Financial data can be obtained via packages like **quantmod** (Ryan 2013). Intra-day price and trading volume can be considered a data stream. As Twitter became popular, packages like **streamR** (Barbera 2014) and **twitteR** (Gentry 2013) provide interfaces to the Twitter web API to retrieve life Twitter feeds.

Statistical models: Several packages provide algorithms for iteratively updating statistical models to typically 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 still has some characteristics needed for data streams. In **rEMM** (Hahsler and Dunham 2014) we implemented a stand-alone version of a pure data stream clustering algorithm called tNN enhanced with a methodology to model a data stream's temporal structure. The clustering part is also available in the **stream** framework.

Distributed computing frameworks: With the development of Hadoop, distributed computing frameworks became very popular to solve large scale computational problems. In R **HadoopStreaming** (Rosenberg 2012) is available to use R script within the Hadoop framework. However, contrary to the word streaming in its name, **HadoopStreaming** does not support data streams. As Hadoop itself, **HadoopStreaming** is used for batch processing. Streaming in the name refers only to the internal usage of pipeslines for "streaming" the input and output between the Hadoop framework and the used R scripts. A distributed framework for realtime computation is Storm developed by the Apache Incubator (2014). Storm builds on the idea to put together a computing topology from spouts (data sources) and bolts (simple computational units). **RStorm** (Kaptein 2013) implements a simple, non-distributed version of Storm. At the time of writing this paper, the topology has a single spout which contains a static data.frame as input.

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** 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 **stream** framework including the design of the class hierarchy to represent different data streams and data stream clustering algorithms. Evaluation of data stream clustering algorithms is discussed in Section 4. In Section 5 we provide comprehensive examples. Extending the framework with new data stream sources and algorithms is briefely described in Section 6 and conclude with Section 7.

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 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). The rest of this section will give a brief introduction of these data stream mining tasks. We will focus on clustering, since this is also the current focus of **stream**.

2.1. Clustering

Clustering, the assignment of data points to (typically k) groups such that point 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-medians, 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 at a time and typically iterate over the data multiple times. This requirement makes these algorithms unsuitable for 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 use a two-stage online/offline approach:

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 intorduced by Aggarwal, Han, Wang, and Yu (2003) based on the idea of cluster features developed for BIRCH 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 typically represented by cluster centers and additional statistics such as weight (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 represent non-empty grid cells (e.g., Tu and Chen (2009); 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 ($k \ll k'$) 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 (typically k-means or reachability introduced by DBSCAN (Ester, Kriegel, Sander, and Xu 1996)) by regarding the micro-cluster centers as pseudo-points. The algorithms are often modified to take also the weight of micro-clusters into account.

2.2. Classification

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

Several classification methods suitable for data streams have been developed recently. 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 description of these and other methods we refer the reader to the survey by Gaber, Zaslavsky, and Krishnaswamy (2007).

2.3. Frequent Pattern Mining

The aim of frequent pattern mining is to discover frequently occurring patterns (e.g., itemsets, subsequences, subtrees, subgraphs) in large data sets. Patterns are then used to summarize the data set and can provide insights into the data. Although finding all frequent pattern is a computationally expensive task, many efficient algorithms have been developed for static data sets. Most notably 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 the data several times and thus makes them unusable for the streaming case. We refer the interested reader to the survey of frequent pattern mining in data streams by Jin and Agrawal (2007) which describe several algorithms for mining frequent itemsets.

2.4. Existing Solution: The MOA Framework

MOA (short for Massive Online Analysis) is a framework implemented in Java for stream classification, regression and clustering (Bifet, Holmes, Kirkby, and Pfahringer 2010). It is the first experimental framework to provide easy access to multiple data stream mining algorithms, as well as tools to generate 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 developed by the University of Waikato and its interface and workflow are similar to those of WEKA.

The workflow in MOA consists of three main steps:

1. Selection of the data stream model (also called data feeds or data generators).
2. Selection of the learning algorithm.
3. Apply selected evaluation methods on the results of the algorithm on the generated data stream.

MOA uses a very appealing graphical user interface. As the output MOA generates a report which contains the results from the data mining task as well as the performance evaluation. The learning algorithm and the evaluation differs depending in the data mining task (classification or clustering). 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.

The MOA framework 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 provides a clear Java interface to add new algorithms or use the existing algorithms in other applications.

3. The stream Framework

A drawback of MOA for R users is that for all but very simple experiments Java code has to be developed. 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 runing code and copying data manually.

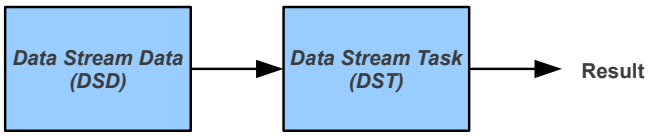


Figure 1: A high level view of the **stream** architecture.

The **stream** framework provides a R-based alternative to the MOA framework. It is based on several packages including **proxy** (Meyer and Buchta 2010), **MASS** (Venables and Ripley 2002), **clue** (Hornik 2013), **clusterGeneration** (Qiu and Joe. 2009), and others. The **stream** extension package **streamMOA** also interfaces the data stream clustering algorithms already available in MOA using the **rJava** package by Urbanek (2011). Furthermore, other than MOA, **stream** can incorporate any algorithm which is written in a language interfaceable by R.

The **stream** framework consists of two main components:

1. **Data Stream Data (DSD)** which manages or creates a data stream, and
2. **Data Stream Task (DST)** which performs a data stream mining task.

Figure 1 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 streaming mining task (e.g., classification or clustering). In the following we will concentrate on clustering since **stream** currently focuses on this type of task, but the framework is implemented such that classification, frequent pattern mining or any other task can be added easily in the future.

stream relies on object-oriented design using the S3 class system (Chambers and Hastie 1992) to provide for each of the two core components a lightweight interface (i.e., an abstract class) which can be easily implemented to create new data stream types or data stream mining algorithms. The detailed design of the DSD and DSC classes will be discussed in the following subsections.

3.1. Data Stream Data (DSD)

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 2 shows the relationship (inheritance) hierarchy of the DSD classes as a UML class diagram (Fowler 2003). All DSD classes extend the 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**. **stream** currently provides the following generators:

1. Streams with static structure
    - `DSD_BarsAndGaussians` generates two bars and two Gaussians clusters with different density.

- `DSD_Gaussians` generates static clusters with random Gaussian distribution.
- `DSD_mlbenchData` provides streaming access to machine learning benchmark data sets found within the `mlbench` package (Leisch and Dimitriadou 2010).
- `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.

2. Streams with concept drift

   - `DSD_Benchmark`, a collection of simple benchmark problems including spliting and joining clusters, changes in density and size. This collection is indented to grow into a 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 filmmaking) or mathematical functions.
   - `DSD_RandomRBFGeneratorEvents` (**streamMOA**) generates streams using radial base functions with noise. Clusters can merge and split.

For reading a saved data stream from a file (in csv format) or to connection to a real stream using a R connection **stream** provides:

- `DSD_ReadStream` is designed to read data from files or open connections.

A non-streaming data set (e.g., stored in a data.frame) can also be wrapped in a stream class to be replayed as a stream over and over again:

- `DSD_Wrapper` wraps static data (e.g., a data.frame, a matrix or a fixed portion of another data stream) as a data stream.

A DSD can also be scaled by wrapping it into an object of class:

- `DSD_ScaleStream` wraps a DSD and scales it using `scale` in **base**.

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

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 are `k`, number of clusters (i.e., areas with high densities), and `d`, number of dimensions. A full list of parameters can be obtained from the help page of each class. The result of this creator function is not a data set but an object representing the streams properties and its current state.

2. **A data generation function** `get_points(x, n=1, ...)`. This function is used to obtain the next data point (or next `n` data points) from the stream represented by object `x`. The data points are returned as a data.frame with each row representing a single data point.

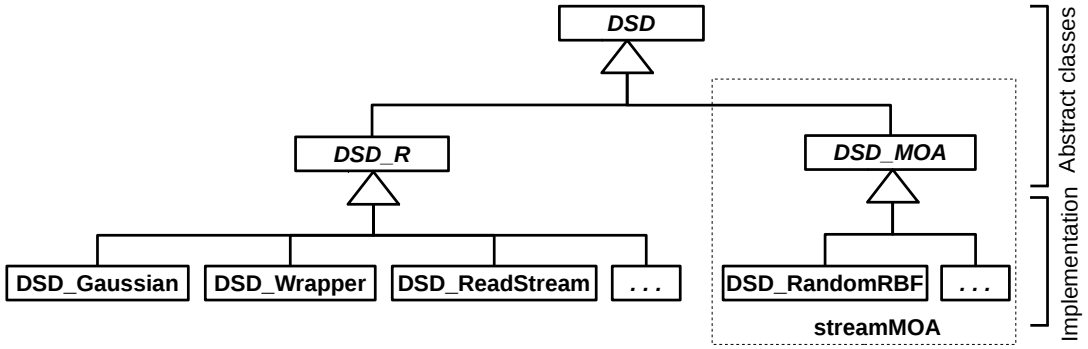


Figure 2: Overview of the Data Stream Data (DSD) class structure.

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_Wrapper` and `DSD_ReadStream` have `reset_stream()` implemented to reset the stream to its beginning.

Following this simple interface, other data stream implementations can be easily added in the future.

3.2. 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 to allow for future extensions and to add even currently unknown tasks. Figure 3 shows the class hierarchy for DST. It is important to note that the DST base class is shown merely for conceptual purposes and not directly visible in the code. The reason is that the actual implementation of clustering (DSC), classification (DSClassify) or frequent pattern mining (DSFP) are typically quite different and the benefit of sharing methods would be minimal. We will restrict the following discussion on data stream clustering (DSC) since **stream** currently focus on this task and has no implemented algorithms for the other tasks.

3.3. Data Stream Clustering (DSC)

Data stream clustering algorithms are implemented as subclasses of the DSC class (see Figure 3). DSCs implement the online process as subclasses of `DSC_Micro` (since it produces micro-clusters) and the offline process as subclasses of `DSC_Macro`.

The following function 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_`.
- `cluster(dsc, dsd, n=1)` which accepts a DSC object and a DSD object. It takes n data points out of the DSD and adds them to the clustering in the DSC object.

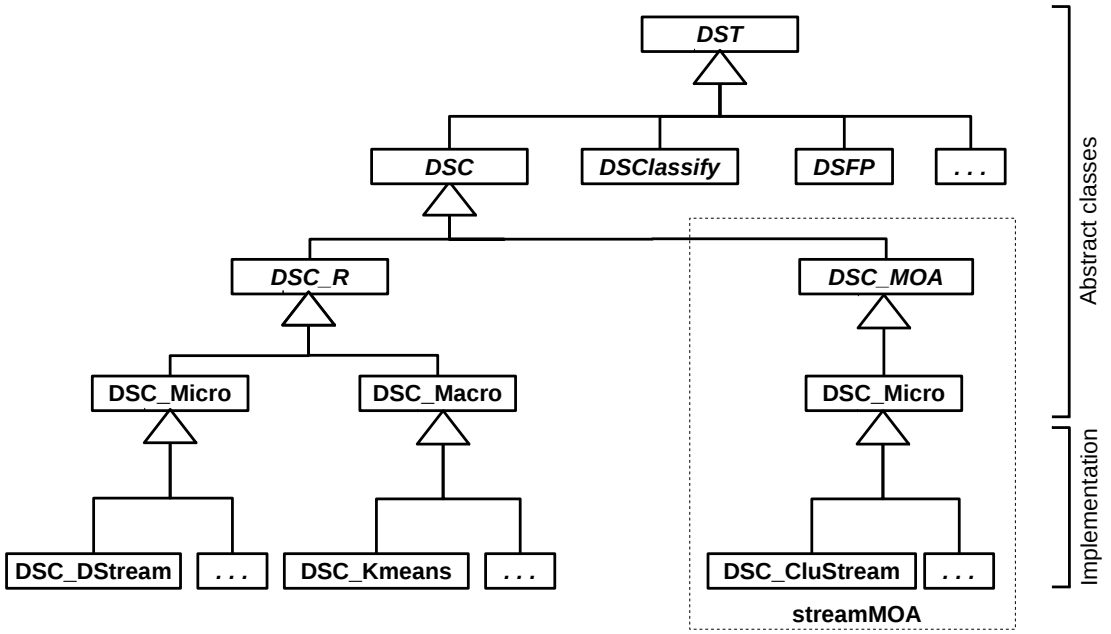


Figure 3: Overview of the Data Stream Task (DST) class structure with subclasses for clustering (DSC), classification (DSCClassify) and frequent pattern mining (DSFP).

- `nclusters(x)` which returns the number of clusters currently in the DSC object. This is important since the number of clusters in not fixed for most data stream clustering algorithms.

- `get_centers(x, type=c("auto", "micro", "macro"), ...)` returns the centers, either centroids or medoids, of the clusters of the DSC object. The default value for `type` is `"auto"` and results in `DSC_Micro` objects to return micro-cluster centers and `DSC_Macro` objects to return macro-cluster centers. Most `DSC_Macro` objects also store the micro-cluster centers and using `type` these centers can also be retrieved. Some `DSC_Micro` implementations also have a reclustering procedure implemented and `type` also allows the user to retrieve macro-clusters. Trying to access centers that are not available in the clustering results in an error.

- `get_weights(x, type=c("auto", "micro", "macro"), ...)` returns the weights of the clusters in the DSC object. How the weight is calculated depends on the clustering algorithm. Typically it depends on the number of points assigned to each cluster.

- `get_assignment(dsc, points, type=c("auto", "micro", "macro"), ...)` assigns each data point in `points` to its nearest cluster center using Euclidean distance and returns a cluster assignment vector.

- `get_copy(x)` creates a deep copy of a DSC object. This is necessary since most clusterings are represented by data structures in Java (for MOA-based algorithms) or by R-based reference classes. Calling this function results in an error if a mechanism for creating a deep copy is not implemented for the used DSC implementation.

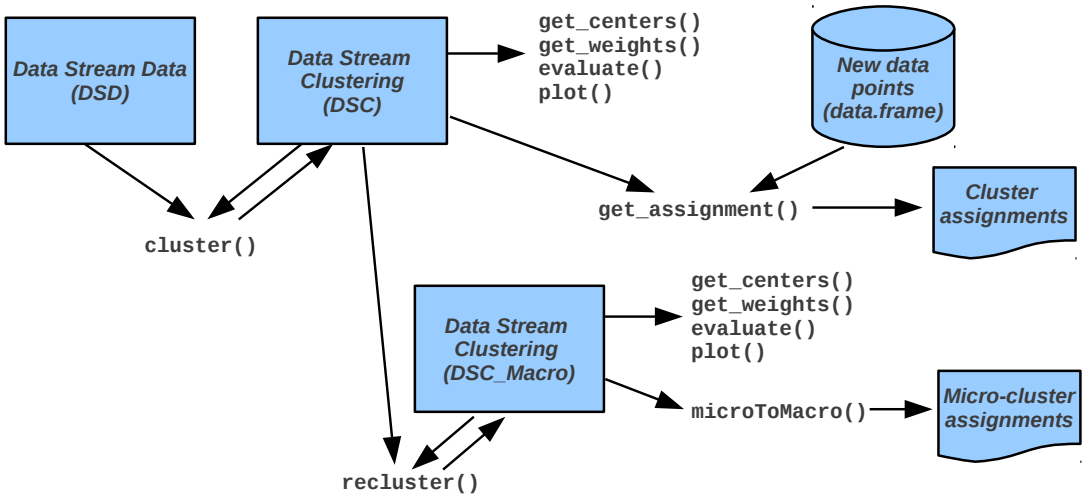


Figure 4: Interaction between the DSD and DSC classes

- `plot(x, dsd=NULL, ..., method="pairs", dim=NULL, type = c("auto", "micro", "macro", "both")` (see manual page for more available parameters) plots the centers of the clusters. There are 3 available plot methods: `"pairs"`, `"plot"`, `"pc"`. Method `"pairs"` is the default method that produces a matrix of scatter plots that plots the attributes against one another (this method is only available when `d > 2`). Method `"plot"` takes the attributes specified in `dim` (the first two if `dim` is unspecified) and plots them as `x` and `y` in a scatter plot. Lastly, method `"pc"` performs Principle Component Analysis (PCA) on the data and projects the data to a 2-dimensional plane and then plots the results. Parameter `type` controls if micro-, macro-clusters or both are plotted.
- `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.

Figure 4 shows the typical use of `cluster()` and other functions. Clustering on a data stream (DSD) is performed with `cluster()` 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 (via `get_centers()`, etc.) from the object after clustering. Note, that DSC classes are implemented as R5 reference classes (mutable objects) and thus the result of cluster does not need to be reassigned to the object. For evaluation, the clusters to which data points would be assigned can be obtained using `get_assignment` which results in a vector with cluster assignments.

Reclustering (the offline component of data stream clustering) is done with `recluster(macro, dsc, type="auto", ...)`. Here the centers in `dsc` 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 implementations for DSC are split into R-based (`DSC_R`) and MOA-based implementations (`DSC_MOA` from package `streamMOA`). (compare Figure 3). The following clustering algorithms are currently available:

- `DSC_CluStream` (**streamMOA**) implements the CluStream algorithm by Aggarwal *et al.* (2003). The algorithm creates a fixed number of micro-clusters and applies weighted k-means on the micro-clusters for reclustering.
- `DSC_ClusTree` (**streamMOA**) implements the ClusTree algorithm by Kranen, Assent, Baldauf, and Seidl (2009). The algorithm organizes the micro-clusters in a tree structure for faster access. Wither k-means or reachability from DBSCAN is used for reclustering.
- `DSC_DenStream` (**streamMOA**) is the DenStream algorithm by Cao, Ester, Qian, and Zhou (2006). DenStream is density-based and, organizes micro-clusters based on their weight as potential and outlier micro-clusters. Micro-clusters are reclustered using reachability from DBSCAN.
- `DSC_DStream` implements the D-Stream algorithm by Tu and Chen (2009). D-Stream uses a grid to estimate density in grid cells. For reclustering adjacent dense cells are merged to form macro-clusters.
- `DSC_Sample` selects representatives via Reservoir Sampling (Vitter 1985).
- `DSC_tNN` implements the simple data stream clustering algorithm called threshold nearest-neighbors (**?**Hahsler and Dunham 2010). Micro-clusters have a fixed radius. For reclustering reachability from DBSCAN is used. method.

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

- `DSC_DBSCAN` implements DBSCAN by Ester *et al.* (1996).
- `DSC_Hierarchical` interfaces R's `hclust` function.
- `DSC_Kmeans` interface R's k-means implementation or 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 distance ϵ from each other (they are within each other's ϵ-neighborhood). Two micro-clusters are reachable if they are connected by a chain of directly reachable micro-clusters. This is related to hierarchical clustering with single linkage.

Finally, clustering sometimes creates 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 with the number of clusters or with the sum of the weight of all clusters (`weight`). The resulting clustering is a static copy (`DSC_Static`). Further clustering cannot be performed by it, but it can be used as input for reclustering.

4. Evaluating Data Stream Clustering

Evaluation of data stream mining is an important issue. We will briefly introduce the evaluation of data stream clustering here and refer the interested reader to the books by Aggarwal (2007) and Gama (2010).

Evaluation of clustering and in particular data stream clustering is discussed in the literature extensively and there are many evaluation criteria available. For the evaluation of conventional clustering we refer the reader to the popular books by Jain and Dubes (1988) and Kaufman and Rousseeuw (1990). Evaluation of data stream clustering is treated in the book by Gama (2010).

Evaluation of data stream clustering is performed in **stream** via

```
evaluate(dsc, dsd, method, n = 1000, type=c("auto", "micro", "macro"),
                          assign="micro"), ...,
```

where `n` data points are taken from `dsd` and assigned to their closest cluster in the clustering in `dsc` using Euclidean distance. By default the points are assigned to micro-clusters, but it is also possible to assign them to macro-cluster centers instead (`assign="macro"`). Then 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. Then the evaluation measure specified in `method` is calculated.

A simple measure is to evaluate the compactness of the data points assigned to each cluster using the sum of squared distances between each data point and the center of its cluster (method `"SSQ"`). This is a measure of internal cluster validity which does not require any information about the ground truth (i.e., true partitioning of the data into classes).

Most evaluation measures perform external evaluation and require the ground truth (analog to the class label in classification) for the data (`dsd`). Then based on cluster membership of each new data point and the class label different measures can be computed. 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). We only list the measures currently available for `evaluate()` (method name are under quotation marks):

- `"precision"`, `"recall"`, F1 measure (`"F1"`),
- `"purity"`, false positive rate (`"fpr"`)
- Rand index (`"Rand"`), adjusted Rand index (`"cRand"`),
- Jaccard index (`"Jaccard"`),
- Euclidean dissimilarity of the memberships (`"Euclidean"`)
- Manhattan dissimilarity of the memberships (`"Manhattan"`),
- Normalized Mutual Information (`"NMI"`)
- Katz-Powell index (`"KP"`)
- Fowlkes and Mallows's index (`"FM"`)

- Maximal cosine of the angle between the agreements (`"angle"`),
- Maximal co-classification rate (`"diag"`),
- Prediction Strength (`"PS"`).

`evaluate()` evaluates the clustering at a certain point in time using explicitly specified test data. However, many data streams exhibit concept drift and evolve over time and it is important to evaluate how well the clustering algorithm is able to adapt to the changing cluster structure. Aggarwal *et al.* (2003) developed an evaluation scheme which was used later on by others (e.g., by Tu and Chen (2009) and Wan *et al.* (2009)). In this approach a horizon is defined as a number of data points which are first clustered and then the evaluation measure is calculated using the same data. Algorithms which can better adapt to the changing stream will achieve a better value. This evaluation strategy is implemented in **stream** as function `evaluate_cluster()`. It shares most parameters with `evaluate()` and can apply the same set of evaluation measures.

5. Examples

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 several increasingly complex examples of how to use **stream**. First, we start with creating a data stream using different implementations of the DSD class. The second example shows how to save and read stream data to and from disk. We then give examples in 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. Finally, the last example introduces the use of data stream clustering algorithms with a detailed comparison of two algorithms from start to finish by first running the online components, then using a weighted k-means algorithm to re-cluster the micro-clusters generated by each algorithm into final clusters.

5.1. Creating a data stream

In this example, we focus on the DSD class to model data streams.

```
> library("stream")
> dsd <- DSD_Gaussians(k=3, d=3, noise=0.05)
> dsd

Static Mixture of Gaussians Data Stream (DSD_Gaussians, DSD_R, DSD)
With 3 clusters in 3 dimensions
```

After loading the **stream** package (and setting a seed for the random number generator to make the experiments reproducible), we call the creator function for the class `DSD_Gaussians` specifying the number of clusters as $k = 4$ and the data dimensionality to $d = 2$ with an added noise of 5% of the data points. This data stream generator chooses for each cluster randomly a mean and a covariance matrix.

New data points are requested from the stream using `get_points(x, n=1, ...)`. When a new data point is requested from this generator, a cluster is chosen randomly and then a point is drawn from a multivariate normal distribution given by the mean and covariance matrix of the cluster. The following instruction requests $n = 5$ new data points.

```
> p <- get_points(dsd, n=5)
> p
```

```
         V1        V2        V3
1 0.6780328 0.5242068 0.5605932
2 0.4120231 0.6052871 0.6360699
3 0.2985138 0.2289293 0.3652078
4 0.7063846 0.4860862 0.4457926
5 0.4405654 0.4725780 0.5969839
```

The result is a data.frame containing the data points as rows. For evaluation it is often important to know the ground truth, in this case from which cluster each point was created. The generator also returns the ground truth if it is called with `assignment=TRUE`. The ground truth is returned as an attribute with the name `"assignment"` and can easily be accessed in the following way:

```
> p <- get_points(dsd, n=100, assignment=TRUE)
> attr(p, "assignment")
```

```
  [1]  2  2  2  2  2  2 NA  2  3  2  3  2  3  3  1  1  3  2  3  3  2  1  2  3  3
 [26]  3  2  1  3  1  2 NA  3  2  1  1  2  3  3  2  1  2  2 NA  1  2  3  3  1  1
 [51]  1  1  2  2  3  3  2  2  1  2  2  1  3  2 NA  3  1  3  3  3  1  3  3  1  2
 [76]  3  3  3  1  2  1  3  3  1  1  2  3  1  3  1  1  1  3  2  3  1  3  1  2  3
```

Note that we created a generator with 5% noise. Noise points do not belong to any cluster and thus have an assignment value of `NA`.

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

```
> plot(dsd, n=500)
```

The data can also be projected on its first two principal components

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

Figures 5 and 6 show the resulting plots. The assignment value is automatically used for color in the plot and noise points are plotted as gray crosses.

Stream also supports data streams which contain concept drift. Example for such a data stream generators are collected in `DSD_Benchmark` where clusters move over time.

```
> dsd <- DSD_Benchmark(1)
> dsd
```

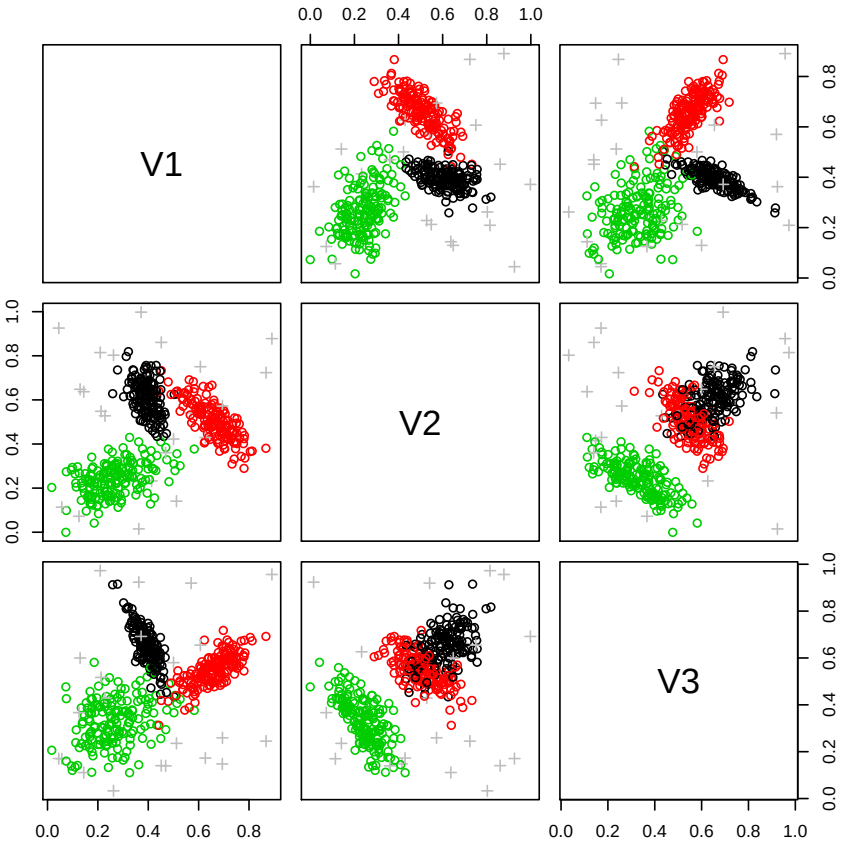


Figure 5: Plotting 500 data points from the data stream

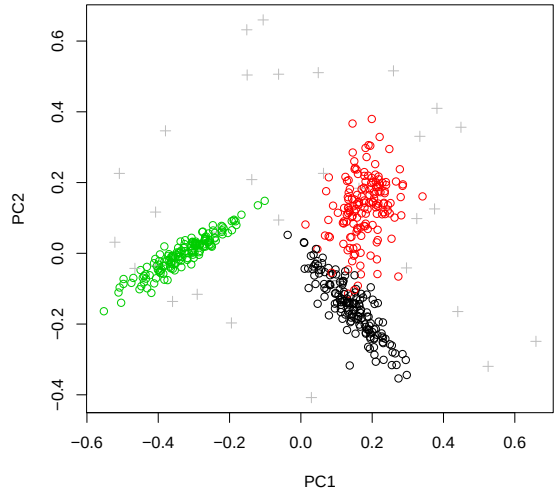


Figure 6: Plotting 500 data points from the data stream projected on its first two principal components

```
Moving Data Generator (DSD_MG, DSD_R, DSD)
With 3 clusters in 2 dimensions. Time is 1
```

k and d represent the number of clusters and the dimensionality of the data, respectively. To show concept drift, we request four times 200 data points from the stream and plot them. To fast-forward in the stream we request 1300 points in between the plots.

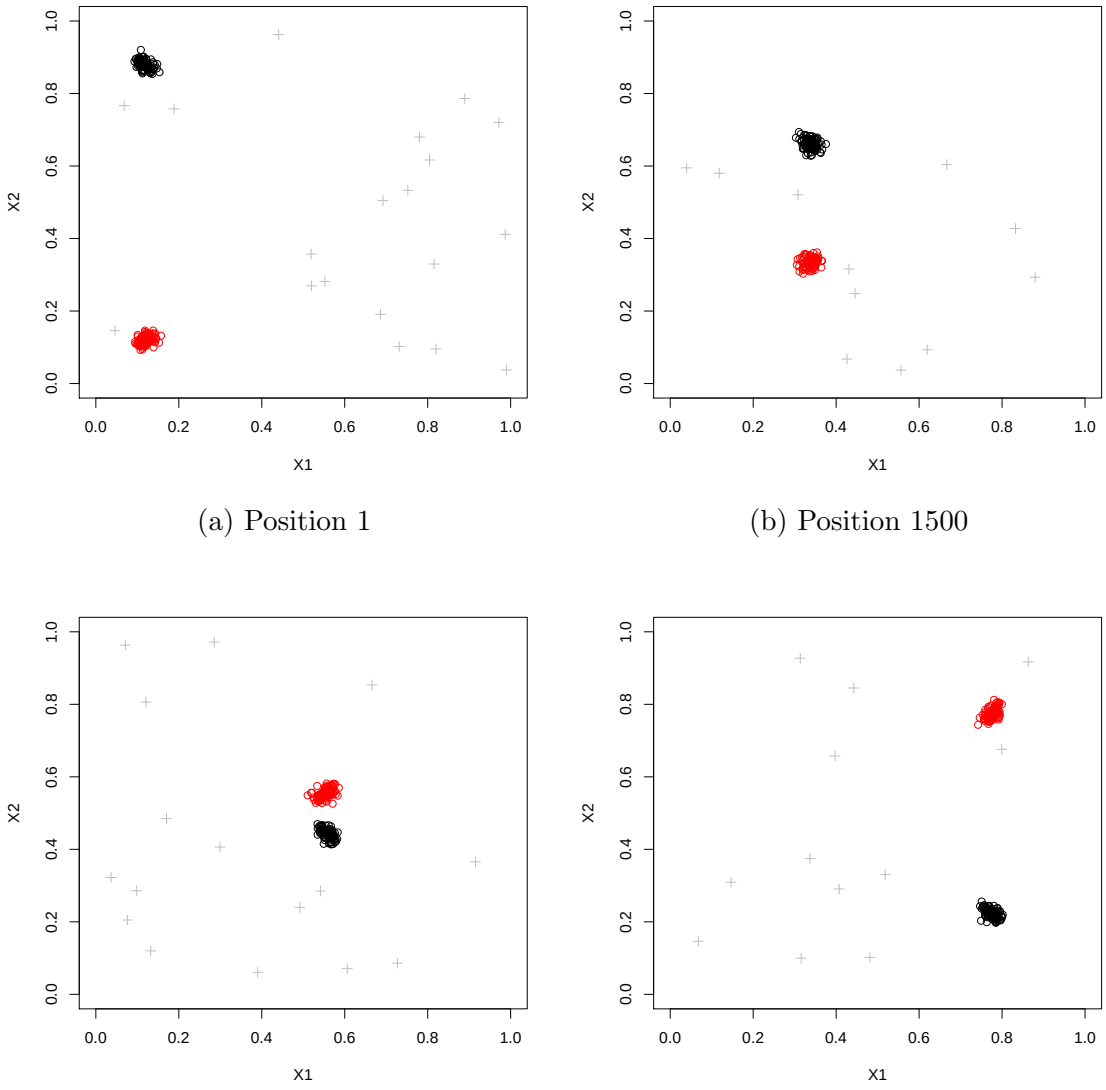
```
> plot(dsd, 200, xlim=c(0,1), ylim=c(0,1))
> tmp <- get_points(dsd, n=1300)
> plot(dsd, 200, xlim=c(0,1), ylim=c(0,1))
> tmp <- get_points(dsd, n=1300)
> plot(dsd, 200, xlim=c(0,1), ylim=c(0,1))
> tmp <- get_points(dsd, n=1300)
> plot(dsd, 200, xlim=c(0,1), ylim=c(0,1))
```

Figure 7 shows the four plots where clusters move over time. An animation of the data can also be generated using `animate_data()`.

```
> reset_stream(dsd)
> animate_data(dsd, n=10000, pointInterval=100, xlim=c(0,1), ylim=c(0,1))
```

The animation is recorded using package **animation** Xie (2013) and can be replayed and using `ani.replay()`, and saved as an animation embedded in a HTML document or an animated gif. More formats for saving the animation are available in **animation**.

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



(a) Position 1

(b) Position 1500

(c) Position 3000

(d) Position 4500

Figure 7: Data points from DSD_RandomRBFGeneratorEvents at different positions in the stream. Note that clusters change position over time.

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

To see a life animation, we refer the reader to the example code in the manual page for `animate_data`.

5.2. Reading and writing data streams

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

We start by creating a DSD object.

```
> dsd <- DSD_Gaussians(k=3, d=5)
> dsd

Static Mixture of Gaussians Data Stream (DSD_Gaussians, DSD_R, DSD)
With 3 clusters in 5 dimensions
```

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

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

`write_stream()` accepts a DSD object, and then either a connection directly, or the file name. The instruction above will create a new file called `dsd_data.cvs` (an existing file will be overwritten). The `sep` parameter defines how the dimensions in each data point (row) are separated. Here `","` is used to create a comma separated values file. The actual writing is done by the `write.table()` function and any additional parameters are passed directly to it. Data points are requested individually from the stream and then written to the connection. This way the only restriction for the size of the written stream are limitations (e.g., the available storage) at the receiving end.

The `DSD_ReadStream` object is used to read a stream from a connection or a file. It reads a single data point at a time with the `read.table()` function. Since, after the read data is processed, e.g., by a data stream clustering algorithm, it it removed from memory, we can efficiently process files larger than the available main memory in a streaming fashion. In the following example we read a data stream that is stored as a compressed csv-file in the package's examples directory.

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

```
File Data Stream (DSD_ReadStream, DSD_R, DSD)
With 7 clusters in 34 dimensions
```

Using `take` and `assignment` we define which columns should be used as data and which column contains the ground truth assignment. We also specify the true number of clusters k. Ground truth and number of clusters do not need to be specified if they are not available or no evaluation with external measures is planned.

`DSD_ReadStream` objects are just like any other DSD object in that you can call `get_points()` to retrieve data points from the data stream.

```
> get_points(dsd_file,5)
```

```
  V1  V5    V6 V8 V9 V10 V11 V13 V14 V15 V16 V17 V18 V19 V20 V23 V24 V25 V26
1  0 215 45076  0  0   0   0   0   0   0   0   0   0   0   0   1   1   0   0
2  0 162  4528  0  0   0   0   0   0   0   0   0   0   0   0   2   2   0   0
3  0 236  1228  0  0   0   0   0   0   0   0   0   0   0   0   1   1   0   0
4  0 233  2032  0  0   0   0   0   0   0   0   0   0   0   0   2   2   0   0
5  0 239   486  0  0   0   0   0   0   0   0   0   0   0   0   3   3   0   0
  V27 V28 V29 V30 V31 V32 V33 V34 V35  V36 V37 V38 V39 V40 V41
1   0   0   1   0   0   0   0   0   0 0.00   0   0   0   0   0
2   0   0   1   0   0   1   1   1   0 1.00   0   0   0   0   0
3   0   0   1   0   0   2   2   1   0 0.50   0   0   0   0   0
4   0   0   1   0   0   3   3   1   0 0.33   0   0   0   0   0
5   0   0   1   0   0   4   4   1   0 0.25   0   0   0   0   0
```

Looping over the data several times and resetting the position in the `DSD_ReadStream` to the file's beginning is possible with `reset_stream()` and will described in the next example.

5.3. 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. We can write a portion of a stream to disk with `write_stream()` and then use `DSD_ReadStream` 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_Wrapper` which provides a stream interface for a matrix/data.frame.

First we create some data and use `get_points()` to store 100 points as a data.frame in `points`.

```
> dsd <- DSD_Gaussians(k=3, d=2)
> p <- get_points(dsd, 100)
> head(p)
```

```
         V1        V2
1 0.7497399 0.4127833
2 0.4564119 0.1287219
3 0.1190636 0.1263880
4 0.6875141 0.5917221
5 0.3401115 0.3384190
6 0.6331463 0.4436735
```

Next, we create a `DSD_Wrapper` object which provides a data stream wrapper for points.

```
> replayer <- DSD_Wrapper(p, k=3)
> replayer

Data Frame/Matrix Stream Wrapper (DSD_Wrapper, DSD_R, DSD)
With 3 clusters in 2 dimensions
Contains 100 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.frame.

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

         V1        V2
1 0.7497399 0.4127833
2 0.4564119 0.1287219
3 0.1190636 0.1263880
4 0.6875141 0.5917221
5 0.3401115 0.3384190

> replayer

Data Frame/Matrix Stream Wrapper (DSD_Wrapper, DSD_R, DSD)
With 3 clusters in 2 dimensions
Contains 100 data points - currently at position 6 - loop is FALSE
```

Note that the stream is now at position 6. The stream only has 94 points left and the following request for more than the available data points will result in an error.

```
> get_points(replayer, n = 1000)

Error in get_points.DSD_Wrapper(replayer, n = 1000) :
  Not enought data points left in stream!
```

`DSD_Wrapper` and `DSD_ReadStream` can 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 via `reset_stream()`.

```
> reset_stream(replayer)
> replayer

Data Frame/Matrix Stream Wrapper (DSD_Wrapper, DSD_R, DSD)
With 3 clusters in 2 dimensions
Contains 100 data points - currently at position 1 - loop is FALSE
```

5.4. Clustering a data stream

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

```
> dsd <- DSD_Gaussians(k=3, d=2, noise=0.05)
> dsd

Static Mixture of Gaussians Data Stream (DSD_Gaussians, DSD_R, DSD)
With 3 clusters in 2 dimensions
```

Next, we prepare the clustering algorithm. We use here `DSC_DStream` and set the gridsize to 0.1.

```
> dstream <- DSC_DStream(gridsize=0.1)
> dstream

DStream (DSC_DStream, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 0
Number of macro-clusters: 0
```

Now we are ready to cluster data from the stream using the `cluster()` function. Note, that `cluster()` will implicitly alter the mutable `dsc` object so no reassignment is necessary.

```
> cluster(dstream, dsd, 500)
> dstream

DStream (DSC_DStream, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 11
Number of macro-clusters: 3
```

After clustering 500 data points, the clustering contains 11 micro-clusters. Note that our implementation of DStream has reclustering built in and therefore also shows macro-clusters. The micro-cluster centers are:

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

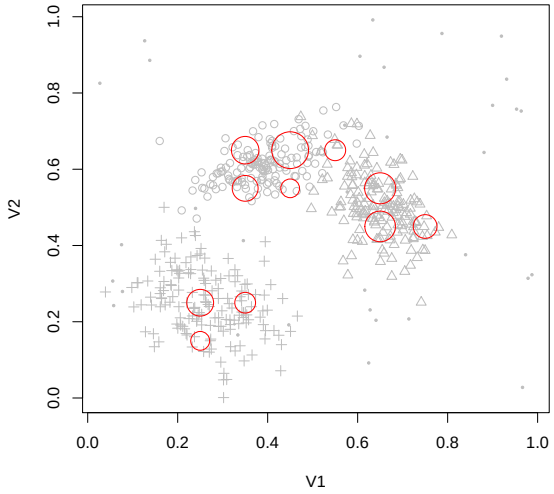


Figure 8: Plotting the micro-clusters produced by DenStream together with the original data points.

```
    V1   V2
1 0.25 0.15
2 0.25 0.25
3 0.35 0.25
4 0.35 0.55
5 0.35 0.65
6 0.45 0.55
```

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

```
> plot(dstream, dsd)
```

The resulting plot is shown in Figure 8. The micro-clusters are plotted in red on top of grey data points. The size of the micro-clusters indicates the weight, i.e., the number of data points represented by the micro-cluster. We see that DenStream places the micro-clusters in dense areas and ignores most of the noise.

5.5. Evaluating results

In this example we will show how to display evaluation measures after clustering data using a DSC object with the `evaluate()` function. The function takes a `DSC` object containing a clustering and a `DSD` with evaluation data to compute several quality measures for clustering. Here we use the data stream and the DenStream clustering objects created in the previous section.

```
> evaluate(dstream, dsd, n = 500)
```

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

numMicroClusters numMacroClusters       numClasses        precision
      11.0000000        3.0000000        3.0000000        0.9547414
          recall               F1           purity              fpr
       0.2592159        0.4077312        0.9547414        0.2592159
             SSQ        Euclidean        Manhattan             Rand
      28.2606866        0.2107948        0.3771552        0.7558185
           cRand              NMI               KP            angle
       0.3310043        0.6240380        0.4372053        0.3771552
            diag               FM          Jaccard               PS
       0.3771552        0.5159371        0.2751016        0.2123002
     classPurity
       0.2592159
```

The number of points taken from `dsd` and used for the evaluation are passed on as the parameter `n`. Individual measures can be calculated using the method argument.

```
> evaluate(dstream, dsd, method = c("purity", "crand"), n = 500)

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

   purity     cRand
0.9600000 0.3440005
```

Purity of the micro-clusters is high since each only covers points from the same true cluster, however, corrected Rand is low because several micro-clusters split the points from each true cluster.

To evaluate how well a clustering algorithm can adapt to an evolving data stream, we use `evaluate_cluster()`. Following the evaluation scheme developed by Aggarwal *et al.* (2003) we define an evaluation horizon as a number of data points which are first clustered and then the evaluation measure is calculated using the same data. The following examples evaluate DenStream on an evolving stream created with `DSD_RandomRBFGeneratorEvents`. We use a fixed seed to make the experiment repeatable.

```
> dsd <- DSD_Benchmark(1)
> micro <- DSC_DStream(gridsize=.05, lambda=.01)
> ev <- evaluate_cluster(micro, dsd, method=c("numMicroClusters","purity"),
+   n=5000, horizon=100)
> head(ev)

     points numMicroClusters purity
[1,]    100               13   0.99
[2,]    200                9   0.98
```

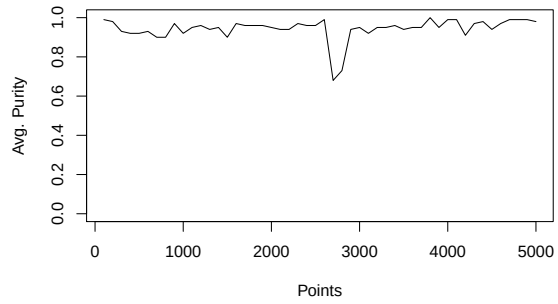


Figure 9: Micro-cluster purity over an evolving stream

```
[3,]    300              10   0.93
[4,]    400               8   0.92
[5,]    500               8   0.92
[6,]    600              12   0.93
```

```
> plot(ev[,"points"], ev[,"purity"], type="l",
+   ylim=c(0,1), ylab="Avg. Purity", xlab="Points")
```

Figure 9 shows the development of the average micro-cluster purity (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 because the two 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 use the same random number seed and initialize the DSD and DSC objects in the same way.

```
> reset_stream(dsd)
> micro <- DSC_DStream(gridsize=.05, lambda=.01)
> r <- animate_cluster(micro, dsd, evaluationMethod="purity", n=5000,
+     horizon=100, pointInterval=100,
+     xlim=c(0,1), ylim=c(0,1))
```

Figure 10 shows the result of the clustering animation with purity evaluation. The whole animation can be recreated by executing the code above. The animation can be again replayed and saved using the package **animation**.

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

5.6. Reclustering DSC objects with another DSC

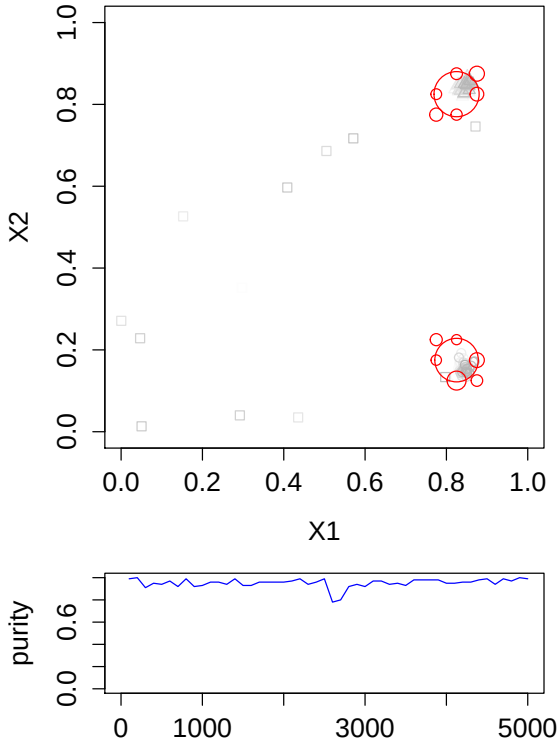
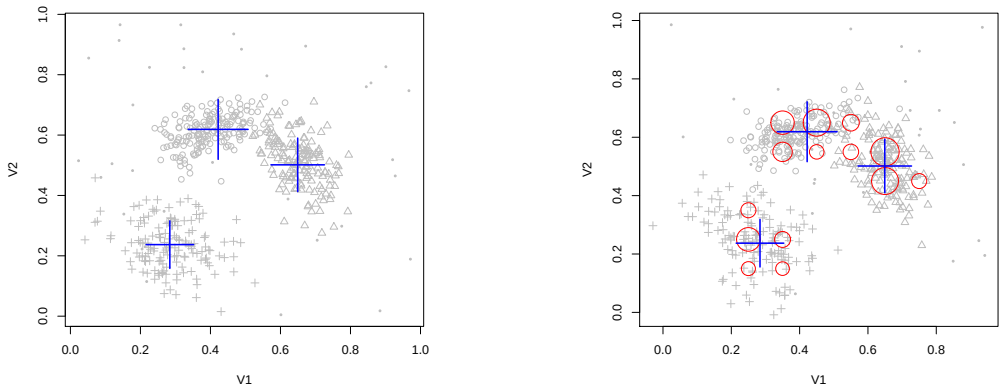


Figure 10: Animated clustering with evaluation.



(a) (b)

Figure 11: A data stream clustered with D-Stream and then reclustered with weighted k-means and $k = 3$. (a) shows macro-clusters and (b) shows both, micro and macro-clusters.

This examples show how to recluster a `DSC` object after creating it. To begin, first create a `DSC` object and run the clustering algorithm.

```
> dsd <- DSD_Gaussians(k=3, d=2, noise=0.05)
> dstream <- DSC_DStream(gridsize=.1)
> cluster(dstream, dsd, 1000)
> dstream

DStream (DSC_DStream, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 14
Number of macro-clusters: 2
```

Although the data contains three clusters, the built in reclustering of D-Stream (joining adjacent dense grids) only produces two macro-clusters. The produced micro-clusters can be reclustered by the `recluster()` method with any avaialble macro-clustering algorithm. Some supported macro-clustering models that are typically used for reclustering are k-means, hierarchical clustering, and reachability. Here we use weighted k-means.

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

weighted k-Means (DSC_Kmeans, DSC_Macro, DSC_R, DSC)
Number of micro-clusters: 14
Number of macro-clusters: 3

> plot(km, dsd)
```

The resulting plot is shown in Figure 11(a). Since `DSC_Kmeans` is a macro-clustering algorithm, the plot contains macro-clusters shown as large blue crosses. The large blue crosses. Micro and macro-clusters can be shown using `type="both"` (see Figure reffigure:recluster(b)).

```
> plot(km, dsd, type="both")
```

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. The assignment is evaluated using the ground truth provided by the data stream generator.

```
> evaluate(km, dsd, method=c("purity", "crand", "SSQ"), n=500)

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

    purity      cRand        SSQ
 0.9531915  0.8695939 46.6188222
```

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

```
> evaluate(km, dsd, c(method="purity", "crand", "SSQ"), n=500, assign="macro")

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

    purity      cRand        SSQ
 0.9666667  0.9056792 47.9831782
```

In this case the evaluation measures purity and corrected Rand sightly increase, since the there are two micro-clusters covering the area between the top two true clusters (see Figure 11(b)). These micro-clusters assigned all its points to one of the two clusterds. Assigning the points rather to the macro-cluster centers splits these points better and therefore decreases the number of incorrectly assigned points. The average within sum of square increases slightly since the number of macro-clusters is smaller than the number of micro-clusters.

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

6. Extending the stream Framework

Since stream mining is a relatively young field and many advances are expected in the near future, the object oriented framework in **stream** is developed with easy extensibility in mind. Implementations for data streams (DSD) and data stream tasks (DST) can be easily added by implementing a small number of core functions. The actual implementation can be written

in either R, Java, C/C++ or any other programming language which can be interfaced by R. In the following we discuss how to extend DSD and DST.

6.1. Implementing new Data Stream Data (DSD) Classes

The class hierarchy in Figure 2 (on page 8) is implemented in the S3 class system by using a vector of class names for the class attribute. For example, an object of class `DSD_Gaussians` will have the class attribute vector `c("DSD_Gaussians", "DSD_R", "DSD")` indicating that the object also is an R implementation of DSD. This allows the framework to implement all common functionality as functions at the level of `DSD` and `DSD_R` and only a minimal set of functions is required to implement a new data stream source.

For a new DSD implementation only a creator function and a `get_points()` method for the class needs to be implemented. The creator function creates an object of the appropriate `DSD` subclass. Typically this S3 object is a list of all parameters, an open R connection and/or an environment (or a reference class) for storing state information (e.g., the current position in the stream). Also an element called `"description"` should be provided. This element is used by `print()`. Note that the class attribute has to contain a vector of all parent classes in the class diagram in bottom-up order. The implemented `get_points()` needs to dispatch for the class and create as the output a data.frame containing the data points as rows. Also, if the ground truth (true cluster assignment) for the data is available, then this can be attached to the data.frame as an attribute called `"assignment"` as an integer vector (noise is represented by `NA`).

For a simple example, we show here the implementation of of `DSD_UniformNoise`.

```
> DSD_UniformNoise <- function(d=2)
+     structure(list(description = "Uniform Noise Data Stream", d = d),
+               class=c("DSD_UniformNoise","DSD_R","DSD"))
> get_points.DSD_UniformNoise <- function(x, n=1, assignment = FALSE, ...) {
+     data <- as.data.frame(t(replicate(n, runif(x$d))))
+     if(assignment) attr(data, "assignment") <- rep(NA, n)
+     data
+ }
```

The constructor only stores the description and the dimensionality of the data. Since all data is random, there is no need to store a state. The `get_points()` implementation creates n random points and if assignments are needed attaches a vector with the appropriate number of `NA`s.

6.2. Implementing new Data Stream Task (DST) Classes

We concentrate again on data stream clustering. However, to add new data stream mining tasks, a subclass hierarchy similar to the hierarchy in Figure 3 (on page 9) for data stream clustering (DSC) can be easily added.

To implement a new clustering algorithm, a creator function (typically named after the algorithm) and a `cluster()` function is needed. The clustering algorithm itself is part of the object created by the creator. To understand this slightly complicated approach consider again Figure 4 (on page 10). The framework provides the function `cluster(dsc, dsd, n=1)`

which contains a loop to go through `n` new data points. In the loop a block of data points is obtained from `dsd` using its `get_point()` function and then the data points are passed on to an internal generic clustering function which has implementations for `DSC_MOA` and `DSC_R`. The implementation for `DSC_MOA` takes care of all MOA-based clustering algorithms. For R-based implementation, the `DSC_R` version looks in the list of the `dsc` object for an element called `"RObj"`, which needs to be a reference class object. Reference classes have been recently introduced with R-2.12 in core package **methods** as a construct for mutable objects. Mutability means that the object can be changed without creating a copy and assigning it back to itself as would be necessary in a purely functional programming language. The RObj in DSC is expected to be a reference class with a `cluster` method. Note at this point that methods of reference classes are called in a very different way from normal R function calls. For example, the cluster method of Robj is invoked by `RObj$cluster()`. However, this is not important for the end user since the cluster method is only used internally and never called directly by the user.

To obtain the clustering result, a methods called `get_microclusters` and `get_microweights` which dispatched for the new class need to be implemented. These methods extract the centers/weights of the clusters from the reference class object in `dsc` and return them as a data.frame (centers) or a vector (weights). These methods are also not exposed to the user and are called internally from `get_centers` and `get_weights`.

For a macro-clustering algorithm, the `cluster` method performs reclustering and `get_macroclusters` and `get_macroweights` need to be implemented. In addition `microToMacro`, a method which does micro- to macro-cluster matching, has to be provided.

For a complete example of a clustering algorithm implemented in R, look at `DSC_DStream` (in file `DSC_DStream.R`) in the package's `R` directory.

7. Conclusion and Future Work

stream is a data stream modeling framework in R that has both a variety of data stream generation tools as well as a component for performing data stream mining tasks. The flexibility offered by our framework allows the user to create a multitude of easily reproducible experiments to compare the performance of these tasks.

Furthermore, the presented infrastructure can be easily extended by adding new data sources and algorithms. 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 design components in Java, Python or C/C++, and then write a small R wrapper as we did for some MOA algorithms in **streamMOA**. This allows experimenting with a multitude of algorithms in a consistent way.

Currently, **stream** focuses on the data stream clustering task. In the future we plan to also incorporate classification and frequent pattern mining algorithms as an extension of the base DST class.

Acknowledgments

This work is supported in part by the U.S. National Science Foundation as a research experience for undergraduates (REU) under contract number IIS-0948893 and by the National

Human Genome Research Institute under contract number R21HG005912.

References

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

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.

Apache Incubator (2014). *Storm: Distributed and Fault-tolerant Realtime Computation*. URL http://storm.incubator.apache.org/.

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

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

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

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 http://CRAN.R-project.org/package=birch.

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.

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.

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.

Gentry J (2013). *twitteR: R Based Twitter Client*. R package version 1.1.7, URL http://CRAN.R-project.org/package=twitteR.

Hahsler M, Dunham MH (2010). "rEMM: Extensible Markov Model for Data Stream Clustering in R." *Journal of Statistical Software*, **35**(5), 1–31. URL http://www.jstatsoft.org/v35/i05/.

Hahsler M, Dunham MH (2014). *rEMM: Extensible Markov Model for Data Stream Clustering in R*. R package version 1.0-9., URL http://CRAN.R-project.org/.

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

Hornik K (2013). *clue: Cluster Ensembles*. R package version 0.3-47., URL http://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 Compututer 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.

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

Kaufman L, Rousseeuw PJ (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and 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.

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.

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

Leisch F, Dimitriadou E (2010). *mlbench: Machine Learning Benchmark Problems*. R package version 2.1-0.

Leydold J (2012). *rstream: Streams of Random Numbers*. R package version 1.3.2, URL http://CRAN.R-project.org/package=rstream.

Meyer D, Buchta C (2010). *proxy: Distance and Similarity Measures*. R package version 0.4-6, URL http://CRAN.R-project.org/package=proxy.

Qiu W, Joe H (2009). *clusterGeneration: Random Cluster Generation*. R package version 1.2.7.

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

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

Ryan JA (2013). *quantmod: Quantitative Financial Modelling Framework*. R package version 0.4-0, URL http://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 http://CRAN.R-project.org/package=rlecuyer.

Silva JA, Faria ER, Barros RC, Hruschka ER, Carvalho ACPLFd, Gama Ja (2013). "Data Stream Clustering: A Survey." *ACM Comput. Surv.*, **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 (2011). *rJava: Low-level R to Java interface*. R package version 0.9-6, URL http://CRAN.R-project.org/package=rJava.

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

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.

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 (2013). *animation: A Gallery of Animations in Statistics and Utilities to Create Animations*. R package version 2.2, URL http://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.

Affiliation:

Michael Hahsler
Engineering Management, Information, and Systems
Lyle School of Engineering
Southern Methodist University
P.O. Box 750122
Dallas, TX 75275-0122
E-mail: mhahsler@lyle.smu.edu
URL: http://lyle.smu.edu/~mhahsler

Matthew Bolaños
Computer Science and Engineering
Lyle School of Engineering
Southern Methodist University
E-mail: mbolanos@smu.edu

John Forrest
Microsoft Corporation
E-mail: jforrest@microsoft.com