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

Michael Hahsler
Southern Methodist University

Matthew Bolaños
Southern Methodist University

John Forrest
Microsoft

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 as an extensible framework for implementing, interfacing and experimenting with algorithms for various data stream mining tasks. The advantages of **stream** are that it seamlessly integrates with existing infrastructure in R (data handling, plotting, existing algorithms, etc.) and supports the use of recently introduced out-of-memory methods (e.g., in **ff** and **bigmemory**). 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.

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 are not static, but are the result of a continuous data generation 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 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 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 on CRAN[1] 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. For Twitter, a popular micro-blogging service, packages like **streamR** (Barbera 2014) and **twitteR** (Gentry 2013) provide interfaces to retrieve life Twitter feeds.

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,

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

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. **rEMM** (Hahsler and Dunham 2014) implemented a stand-alone version of a pure data stream clustering algorithm enhanced with a methodology to model a data stream's temporal structure.

Distributed computing frameworks: With the development of Hadoop[2], distributed computing frameworks to solve large scale computational problems became very popular. **HadoopStreaming** (Rosenberg 2012) is available to use R map and reduce scripts 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 pipelines for "streaming" the input and output between the Hadoop framework and the used R scripts. A distributed framework for realtime computation is Storm[3]. Storm builds on the idea of constructing a computing topology from spouts (data sources) and bolts (computational units). **RStorm** (Kaptein 2013) implements a simple, non-distributed version of Storm.

Even in the stream-related packages discussed above, data is still represented by static 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 several comprehensive examples. Extending the framework with new data stream sources and algorithms is briefly described in Section 6 and Section 7 concludes the paper.

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). In this section we 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. Data Stream Clustering

[2] http://hadoop.apache.org/
[3] http://storm.incubator.apache.org/

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-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 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 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 typically 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 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 generation 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. New data points have more impact on the clustering and the effect 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. 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.

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).

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 pattern is a computationally expensive task, many efficient algorithms have been developed for static data sets. An 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 streaming case. We refer the interested reader to the surveys of frequent pattern mining in data streams by Jin and Agrawal (2007), Cheng, Ke, and Ng (2008) and Vijayarani and Sathya (2012) which describe several algorithms for mining frequent patterns in streams.

2.3. Existing Tools

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 was 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.

Similar to WEKA, MOA uses a very appealing graphical user interface. 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.

SAMOA[4] (Scalable Advanced Massive Online Analysis) is a recently introduced tool for dis-

[4] http://yahoo.github.io/samoa/

tributed stream mining based on 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 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).

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 (or C++ for Jabatus) 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.

3. The stream Framework

The **stream** framework provides a R-based alternative to MOA which seamlessly integrates with the extensive existing R infrastructure. Since R can interface code written in a whole set of 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 2013), **cluster** (Maechler, Rousseeuw, Struyf, Hubert, and Hornik 2014), **clusterGeneration** (Qiu and Joe. 2009), **MASS** (Venables and Ripley 2002), **proxy** (Meyer and Buchta 2010), 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).

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 stream 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.

[5] http://jubat.us/en/

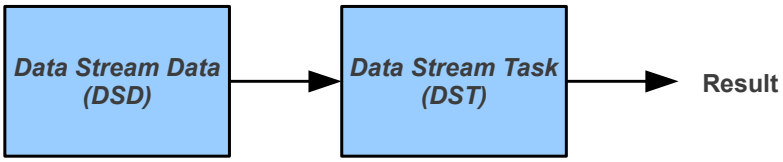


Figure 1: A high level view of the **stream** architecture.

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 definition (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 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.

stream provides a set of DSD implementations. The following generators are currently available:

1. Streams with static structure

   - `DSD_BarsAndGaussians` generates two bars and two Gaussians clusters with different density.
   - `DSD_Gaussians` generates static clusters with random multivariate Gaussian distributions.
   - `DSD_mlbenchData` provides streaming access to machine learning benchmark data sets found in 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 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 mathematical functions.
- `DSD_RandomRBFGeneratorEvents` (**streamMOA**) generates streams using radial base functions with noise. Clusters move, merge and split.

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

- `DSD_ReadStream`, which reads data from files or open connections and makes it available in a streaming fashion.

A non-streaming data set stored in a matrix-like object (e.g., data.frame) can also be wrapped in a stream class to be replayed as a stream.

- `DSD_Wrapper` wraps static matrix-like data (e.g., a data.frame, a matrix) which represent a fixed portion of a data stream and provides a streaming interface. Matrix-like objects also includes large, out-of-memory objects like `ffdf` from package **ff** (Adler, Gläser, Nenadic, Oehlschlägel, and Zucchini 2014) or `big.matrix` from package **bigmemory** (Kane, Emerson, and Weston 2013). Using these, stream mining algorithms (e.g., clustering) can be performed on data that does not fit into main memory. In addition, `DSD_Wrapper` can directly create a static copy of a portion of another DSD object to be replayed several times.

Data in a DSD can also be standardized (centering and scaling) in flight by wrapping it into an object of class:

- `DSD_ScaleStream` estimates scaling parameters from a sample of the stream and then standardizes each new data point as it arrives using R's `scale()` function.

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`, the number of clusters (i.e., areas of density, and `d`, the 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 stream's 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.

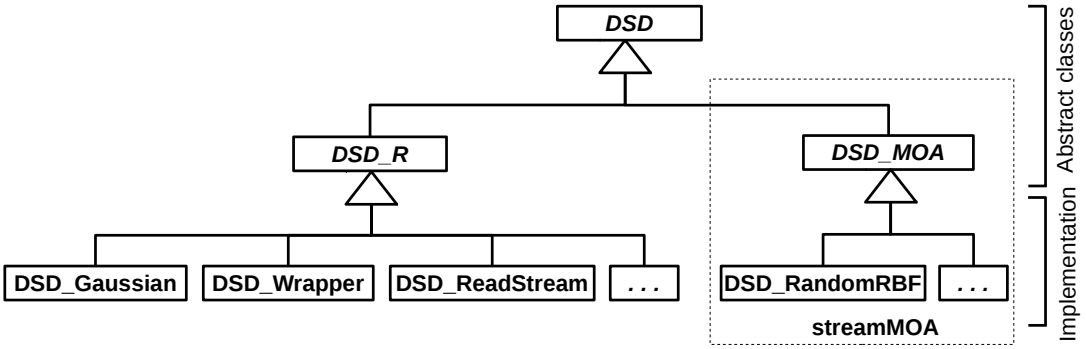


Figure 2: Overview of the Data Stream Data (DSD) class structure.

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. This will be discussed in Section 6.

3.2. Data Stream Task (DST) and Data Stream Clustering (DSC)

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 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 purpose and is not directly visible in the code. The reason is that the actual implementations of clustering (DSC), classification (DSClassify) or frequent pattern mining (DSFP) are typically quite different and the benefit of sharing methods would be minimal.

DST classes implement in **stream** mutable objects. Mutable objects can be changed without creating a copy. This is more efficient, since otherwise for processing each data point a new copy of all used data structures used by the algorithm would be created. 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, external pointers to 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 and algorithms for the other tasks are currently under development.

Data stream clustering algorithms are implemented as subclasses of the class `DSC` (see Figure 3). 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`.

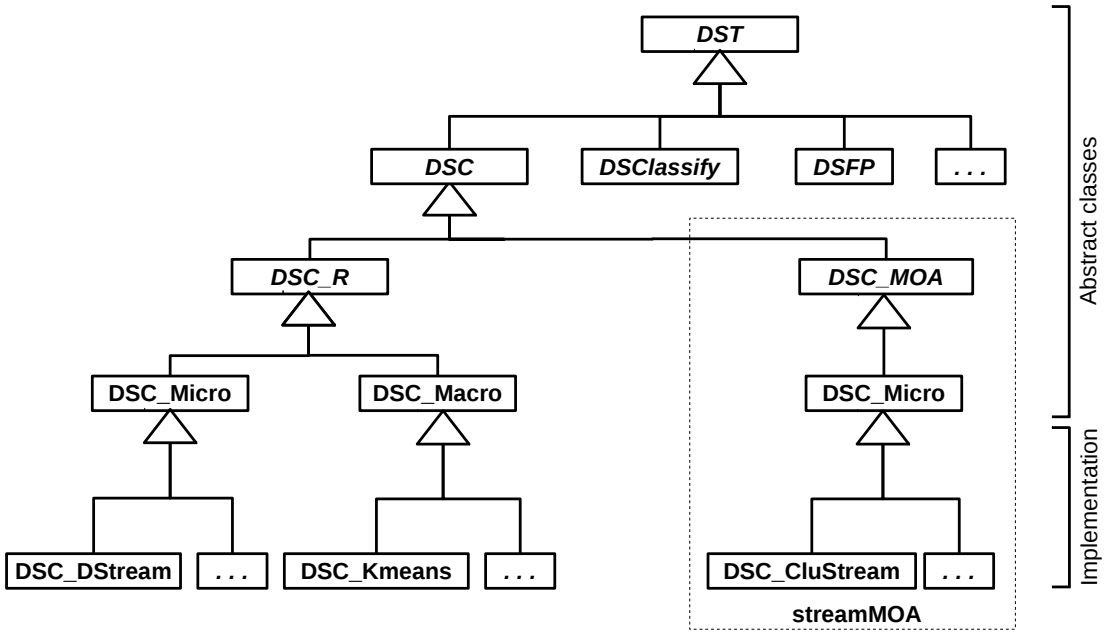


Figure 3: Overview of the Data Stream Task (DST) class structure with subclasses for clustering (DSC), classification (DSCClassify) and frequent pattern mining (DSFP).

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 `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 in 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 in 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

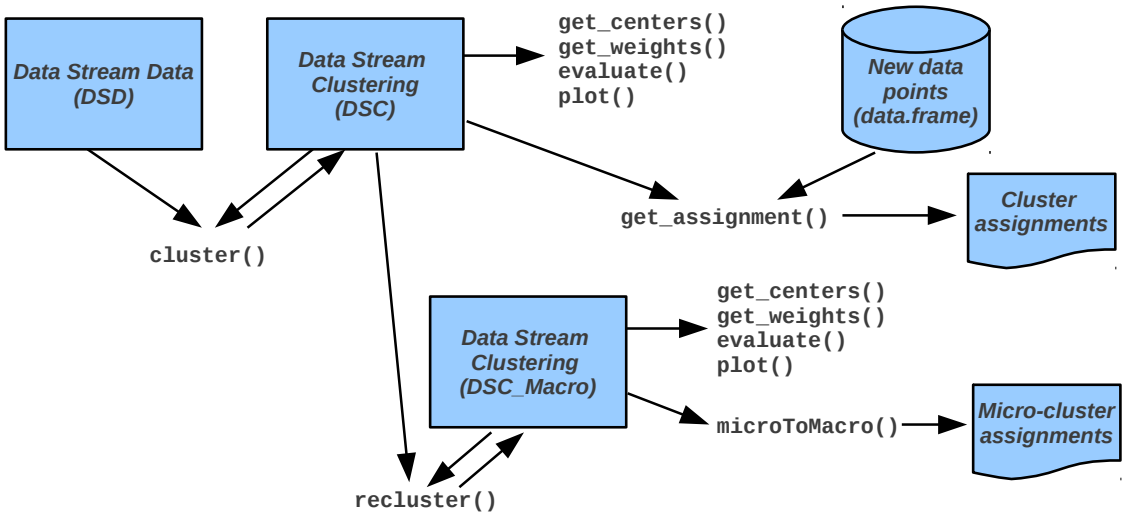


Figure 4: Interaction between the DSD and DSC classes

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="Euclidean", ...)` returns a cluster assignment vector indicating to which cluster each data point in `points` would be assigned. For assignment, the nearest cluster using the distance measure specified in `method` is used.

- `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 implemented for the used DSC implementation.

- `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"`, `"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 if micro-, macro-clusters or both are plotted. If a DSD object is provides as `dsd`, then some example data points are plotted in the background in light grey.

- `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 from the object after clustering (via `get_centers()`, etc.). Note, that DSC classes implement mutable objects and thus the result of `cluster()` does not need to be reassigned to its name.

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 following data stream clustering algorithms are currently available:

- `DSC_CluStream` (**streamMOA**) implements 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**) implements 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**) is 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` selects a user-specified number of representative points from the stream via *Reservoir Sampling* (Vitter 1985). Sampling can keep 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, Algorithm 2.1 by Aggarwal (2006) is implemented.
- `DSC_tNN` implements the simple data stream clustering algorithm called *tNN threshold nearest-neighbors (tNN)* which was developed for package **rEMM** by Hahsler and Dunham (2014, 2010). Micro-clusters are defined by a fixed radius (threshold) around their center. Reachability from DBSCAN is used for reclustering.
- `DSC_Window` 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 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 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 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.

Finally, some 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.

4. Evaluating Data Stream Clustering

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, the evaluation of data stream clustering is still in its infancy. We will only briefly introduce the current state of the evaluation of data stream clustering here and refer the interested reader to the books by Aggarwal (2007) and Gama (2010), and the paper by Kremer, Kranen, Jansen, Seidl, Bifet, Holmes, and Pfahringer (2011).

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

```
evaluate(dsc, dsd, measure, n = 1000, type=c("auto", "micro", "macro"),
   assign="micro"), assignmentMethod=c("auto", "model", "nn"), ...),
```

where `n` data points are taken from `dsd` and assigned to their closest cluster 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 using 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. 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. Finally, the evaluation measure specified in `measure` is calculated. Several measures can be specified as a vector of character strings.

Clustering evaluation measures can be categorized into internal and external cluster validity measures. Internal measures evaluate properties of the clusters. A simple measure to evaluate the compactness of (spherical) clusters is 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 the 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 2013) and **cluster** (Maechler *et al.* 2014). Measures currently available for `evaluate()` (method name are under quotation marks) include:

- Information items
  - `"numMicroClusters"` number of micro-clusters,
  - `"numMacroClusters"` number of macro-clusters,
  - `"numClasses"` number of classes (i.e., groups in the ground truth).
- Internal evaluation measures
  - `"SSQ"` sum of squares (actual noise points are excluded),
  - `"silhouette"` average silhouette width (points that are actual noise and predicted to be noise are excluded) (**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 / maximum diameter) (**fpc**),
  - `"dunn2"` minimum average dissimilarity between two cluster / maximum average within cluster dissimilarity (**fpc**),
  - `"entropy"` entropy of the distribution of cluster memberships (**fpc**),
  - `"wb.ratio"` average.within/average.between (**fpc**)

- External evaluation measures
  - `"precision"`,
  - `"recall"`,
  - `"F1"` F1 measure,
  - `"purity"` (average purity of found clusters),
  - `"fpr"` false positive rate,
  - `"Euclidean"` Euclidean dissimilarity of the memberships (**clue**),
  - `"Manhattan"` Manhattan dissimilarity of the memberships (**clue**),
  - `"Rand"` Rand index (**clue**),
  - `"cRand"` Corrected Rand index (**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**)

Noise data points need special attention. For external validity measures, noise data points just form a special group in the partition. However, for internal 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 internal measures, noise points are excluded.

`evaluate()` 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. However, since data streams typically exhibit concept drift and evolve over time this approach might not be ideal. Also, for data streams 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 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 horizons and after clustering all the data in a horizon the average 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)). Wan

et al. (2009) also use the scheme for the external measure of average purity in 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. Algorithms which can better adapt to the changing stream will achieve better evaluation values. This evaluation strategy is implemented in **stream** as function `evaluate_cluster()`. It shares most parameters with `evaluate()` and all evaluation measures for `evaluate()` described above can be used.

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, in Section 5.1 we start with creating a data stream using different implementations of the DSD class. The second example in Section 5.2 shows how to save and read stream data to and from disk. Section 5.3 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. We show how to cluster data streams in Section 5.4 and to evaluate cluster algorithms in Section 5.5. Finally, reclustering examples are given in Section 5.6. All presented examples contain the complete code necessary to replicate the examples.

5.1. Creating a Data Stream

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

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

Mixture of Gaussians
Class: DSD_Gaussians, DSD_R, DSD
With 3 clusters in 3 dimensions
```

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 and then a point is drawn from a 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 $n = 5$ new data points.

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

     V1    V2    V3
1 0.678 0.524 0.561
```

```
2 0.412 0.605 0.636
3 0.299 0.229 0.365
4 0.706 0.486 0.446
5 0.441 0.473 0.597
```

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. 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 be accessed in the following way:

```
R> p <- get_points(dsd, n=100, assignment=TRUE)
R> 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
 [24]  3  3  3  2  1  3  1  2 NA  3  2  1  1  2  3  3  2  1  2  2 NA  1  2
 [47]  3  3  1  1  1  1  2  2  3  3  2  2  1  2  2  1  3  2 NA  3  1  3  3
 [70]  3  1  3  3  1  2  3  3  3  1  2  1  3  3  1  1  2  3  1  3  1  1  1
 [93]  3  2  3  1  3  1  2  3
```

Note that the data was created by 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.

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

The data can also be projected on its first two principal components using `method="pc"`.

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

Figures 5 and 6 show the resulting plots. The assignment values are automatically used to distinguish between clusters using color and different plotting symbols. Noise points are plotted as gray dots.

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

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

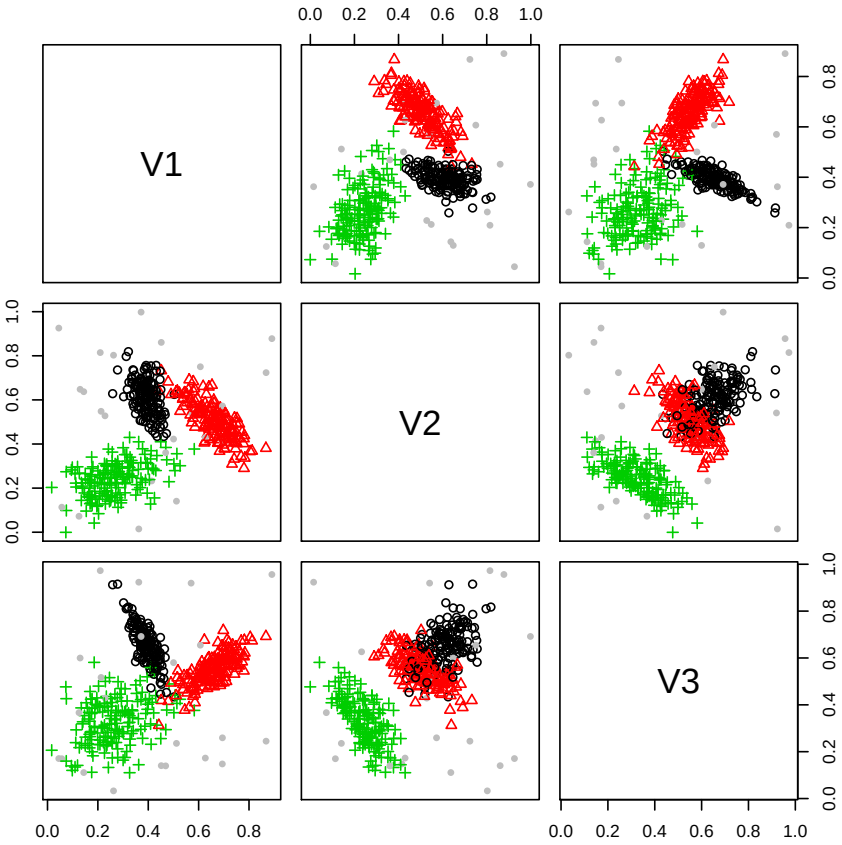


Figure 5: Plotting 500 data points from the data stream

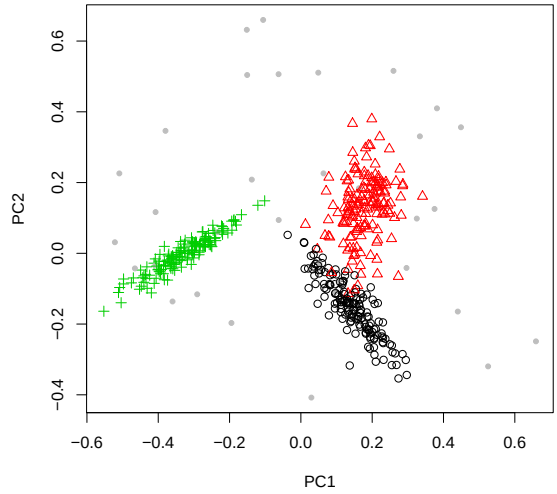


Figure 6: Plotting 500 data points from the data stream projected onto its first two principal components

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.

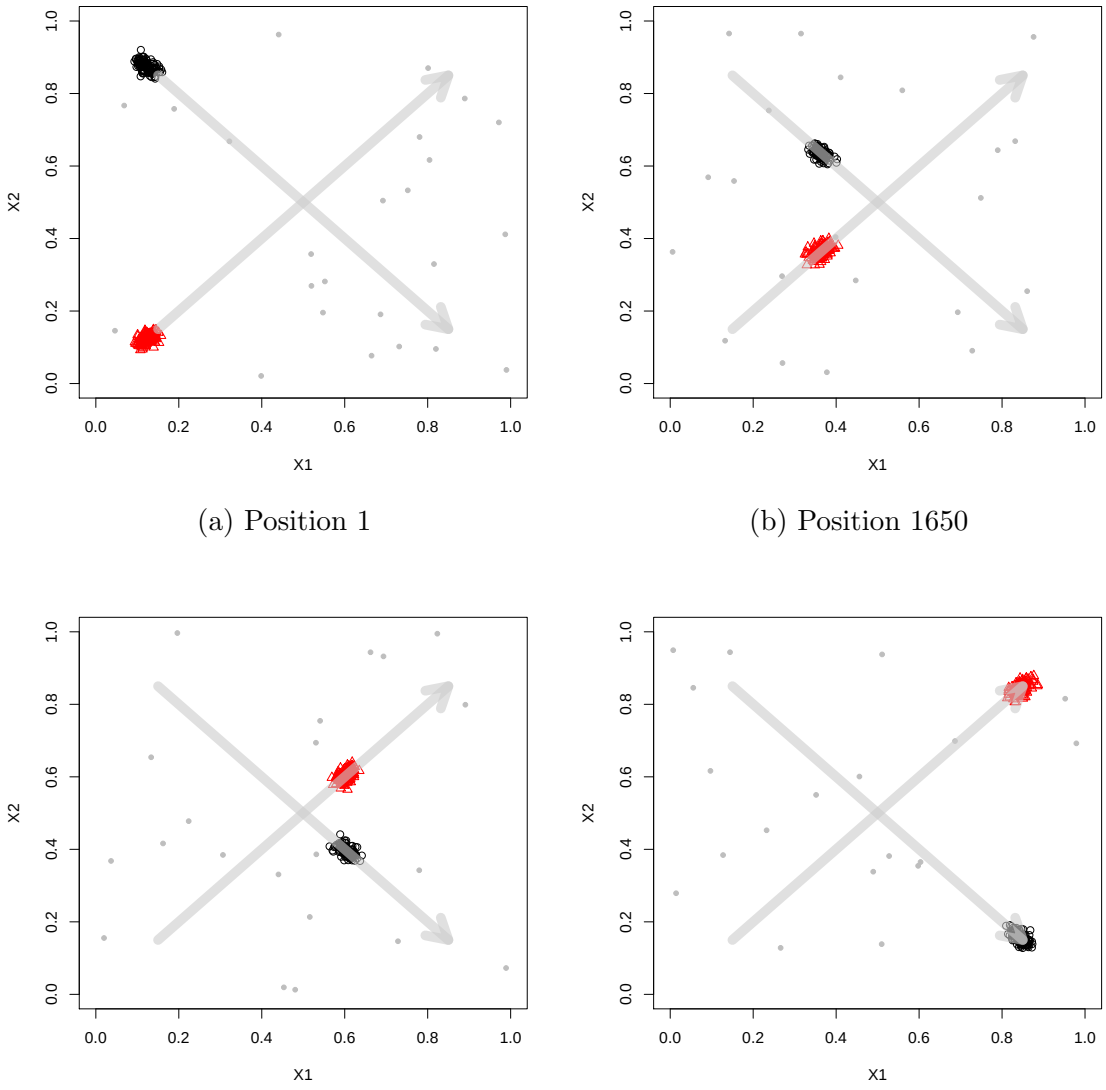
```
R> for(i in 1:4) {
+   plot(dsd, 250, xlim=c(0,1), ylim=c(0,1))
+   tmp <- get_points(dsd, n=1400)
+ }
```

Figure 7 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(dsd)
R> animate_data(dsd, n=10000, horizon=100, xlim=c(0,1), ylim=c(0,1))
```

Animations are recorded using package **animation** (Xie 2013) and can be replayed using `ani.replay()`, and saved as an animation embedded in a HTML document or an animated image in the Graphics Interchange Format (GIF).

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



(a) Position 1

(b) Position 1650

(c) Position 3300

(d) Position 4950

Figure 7: Data points from `DSD_Benchmark(1)` at different positions in the stream. The two arrows are added to highlight the direction of movement.

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

5.2. 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 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.

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

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

```
R> write_stream(dsd, "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.cvs` (an existing file will be overwritten). 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 the `write.table()` function and additional parameters are passed on. 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 at the receiving end (e.g., the available storage).

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 using 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 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> dsd_file <- DSD_ReadStream(gzfile(file),take=c(1, 5, 6, 8:11, 13:20, 23:41),
+ assignment=42, k=7)
R> dsd_file

File Data Stream (kddcup10000.data.gz)
Class: 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. Note that at this point no data has been read in. Reading only occurs when `get_points` is called.

```
R> get_points(dsd_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
1   0   0   0   1   0   0   0   0   0   0 0.00   0   0   0   0   0
2   0   0   0   1   0   0   1   1   1   0 1.00   0   0   0   0   0
3   0   0   0   1   0   0   2   2   1   0 0.50   0   0   0   0   0
4   0   0   0   1   0   0   3   3   1   0 0.33   0   0   0   0   0
5   0   0   0   1   0   0   4   4   1   0 0.25   0   0   0   0   0
```

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

```
       V1     V5       V6 V8 V9     V10 V11 V13 V14 V15 V16 V17     V18
1 -0.0507 -0.645  3.22349  0  0 -0.0447   0   0   0   0   0   0 -0.0316
2 -0.0507 -1.806  0.00853  0  0 -0.0447   0   0   0   0   0   0 -0.0316
3 -0.0507 -0.185 -0.25312  0  0 -0.0447   0   0   0   0   0   0 -0.0316
4 -0.0507 -0.250 -0.18938  0  0 -0.0447   0   0   0   0   0   0 -0.0316
5 -0.0507 -0.119 -0.31195  0  0 -0.0447   0   0   0   0   0   0 -0.0316
      V19 V20    V23    V24 V25 V26 V27 V28 V29 V30    V31    V32   V33
1 -0.0316   0 -1.030 -1.037   0   0   0   0   0   0 -0.428 -0.893 -3.43
2 -0.0316   0 -0.869 -0.917   0   0   0   0   0   0 -0.428 -0.881 -3.41
3 -0.0316   0 -1.030 -1.037   0   0   0   0   0   0 -0.428 -0.870 -3.40
4 -0.0316   0 -0.869 -0.917   0   0   0   0   0   0 -0.428 -0.858 -3.38
5 -0.0316   0 -0.708 -0.798   0   0   0   0   0   0 -0.428 -0.847 -3.37
  V34 V35    V36   V37 V38 V39 V40 V41
1  -1   0 -0.486 -1.64   0   0   0   0
2   0   0  4.906 -1.64   0   0   0   0
3   0   0  2.210 -1.64   0   0   0   0
4   0   0  1.293 -1.64   0   0   0   0
5   0   0  0.862 -1.64   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. As described in the previous section, 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 and 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-like objects.

For illustration purposes, we use data for four major European stock market indices.

```
R> data(EuStockMarkets)
R> head(EuStockMarkets)

      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
[6,] 1611 1672 1714 2467
```

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

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

Matrix Stream Wrapper
Class: DSD_Wrapper, DSD_R, DSD
With NA clusters in 4 dimensions
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
```

```
Matrix Stream Wrapper
Class: DSD_Wrapper, DSD_R, DSD
With NA clusters in 4 dimensions
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 an error.

```
R> get_points(replayer, n = 2000)
```

```
Error in get_points.DSD_Wrapper(replayer, n = 2000) :
  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()` or to an arbitrary position like 100.

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

```
Matrix Stream Wrapper
Class: DSD_Wrapper, DSD_R, DSD
With NA clusters in 4 dimensions
Contains 1860 data points - currently at position 100 - loop is FALSE
```

`DSD_Wrapper` also accepts other matrix-like objects. This includes 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).

5.4. Clustering a Data Stream

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

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

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. We use here a gridsize of 0.1.

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

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

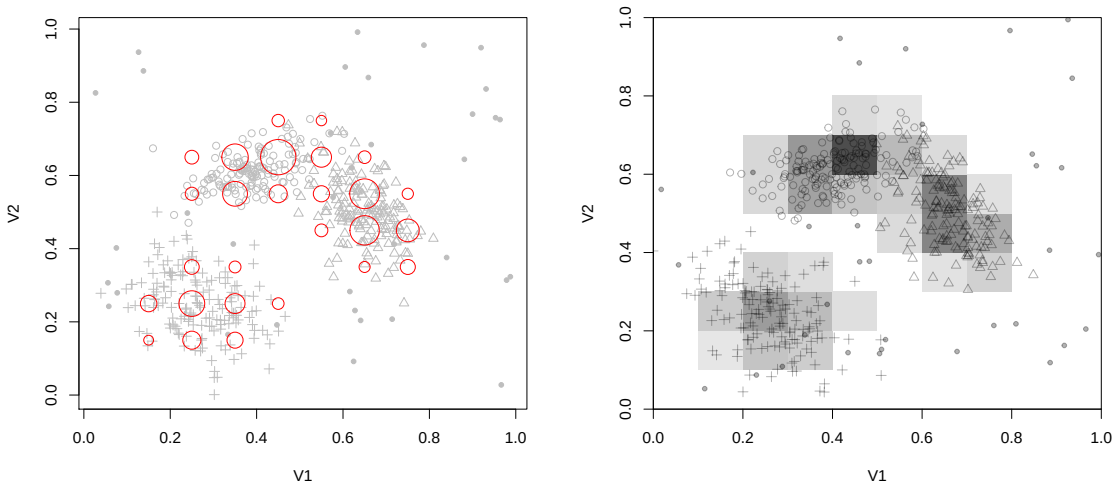
Now we are ready to cluster data from the stream using the `cluster()` function. Note, that `cluster()` will implicitly alter the mutable DSC object so no reassignment is necessary.

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

D-Stream
Class: DSC_DStream, DSC_Micro, DSC_R, DSC
Number of micro-clusters: 27
Number of macro-clusters: 3
```

After clustering 500 data points, the clustering contains 27 micro-clusters. Note that our implementation of D-Stream has built-in reclustering and therefore also shows macro-clusters. The micro-cluster centers are:

```
R> get_centers(dstream)

     V1   V2
1  0.15 0.15
2  0.15 0.25
3  0.25 0.15
4  0.25 0.25
5  0.25 0.35
6  0.25 0.55
7  0.25 0.65
8  0.35 0.15
9  0.35 0.25
10 0.35 0.35
11 0.35 0.55
12 0.35 0.65
13 0.45 0.25
14 0.45 0.55
15 0.45 0.65
16 0.45 0.75
17 0.55 0.45
18 0.55 0.55
19 0.55 0.65
20 0.55 0.75
21 0.65 0.35
22 0.65 0.45
23 0.65 0.55
24 0.65 0.65
25 0.75 0.35
26 0.75 0.45
27 0.75 0.55
```

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



(a) (b)

Figure 8: 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.

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

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

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

The resulting plots are shown in Figure 8. In Figure 8(a) 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 each micro-cluster. In Figure 8(b) the micro-clusters are shown as dense grid cells (density is coded with grey values).

5.5. 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. The `evaluate()` function takes a DSC object containing a clustering and a DSD object with evaluation data to compute several quality measures for clustering. Here we use the data stream and the D-Stream clustering objects created in the previous section.

```
R> evaluate(dstream, dsd, n = 100)

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

     numMicroClusters      numMacroClusters             numClasses
```

```
            27.00000             3.00000             4.00000
                 SSQ          silhouette           precision
             0.86333             0.15073             0.77551
              recall                  F1              purity
             0.11592             0.20170             0.96821
           Euclidean           Manhattan                Rand
             0.12250             0.23000             0.69616
               cRand                 NMI                  KP
             0.12647             0.52981             0.25188
               angle                diag                  FM
             0.23000             0.23000             0.29983
             Jaccard                  PS      average.between
             0.11216             0.10128             0.31180
      average.within        max.diameter        min.separation
             0.04742             0.13234             0.00906
ave.within.cluster.ss                  g2        pearsongamma
             0.00105             0.95825             0.34207
                dunn               dunn2             entropy
             0.06846             0.65551             3.01822
            wb.ratio                  vi
             0.15207             2.19576
```

The number of points taken from `dsd` and used for the evaluation are passed on as the parameter `n`. If no evaluation measure is specified, then all available measures are calculated. Individual measures can be calculated using the measure argument. Note that this uses a new set of 500 evaluation data points from the stream and this the results vary slightly from above.

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

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

purity  cRand
 0.935  0.168
```

Purity of the micro-clusters is high since each micro-cluster 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, **stream** provides `evaluate_cluster()`. Following the evaluation scheme developed by Aggarwal *et al.* (2003), we define an evaluation horizon as a number of data points. Each data point in the horizon is used for clustering and then it is evaluated how well the point's cluster assignment fits into the clustering (internal evaluation) or agrees with the known true clustering (external evaluation). Average evaluation measures for each horizon are returned.

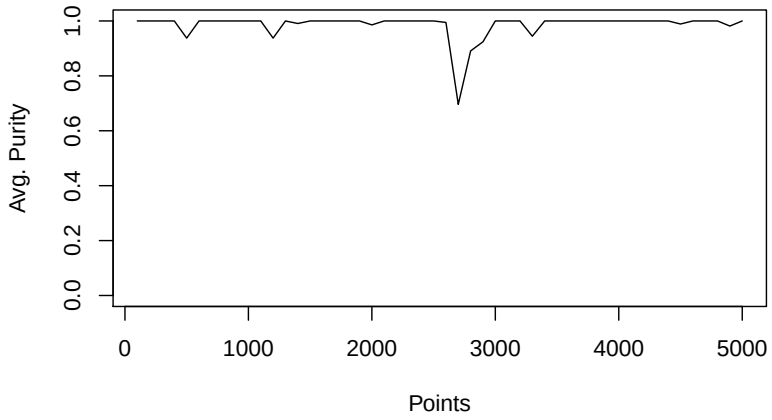


Figure 9: Micro-cluster purity of D-Stream over an evolving stream.

The following examples evaluate D-Stream on an evolving stream created with `DSD_Benchmark`. This data stream was shown in Figure 7 on page 20 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.

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

     points numMicroClusters purity
[1,]    100               13  1.000
[2,]    200               10  1.000
[3,]    300               14  1.000
[4,]    400               10  1.000
[5,]    500                9  0.938
[6,]    600               14  1.000

R> 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 (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 an new empty clustering.

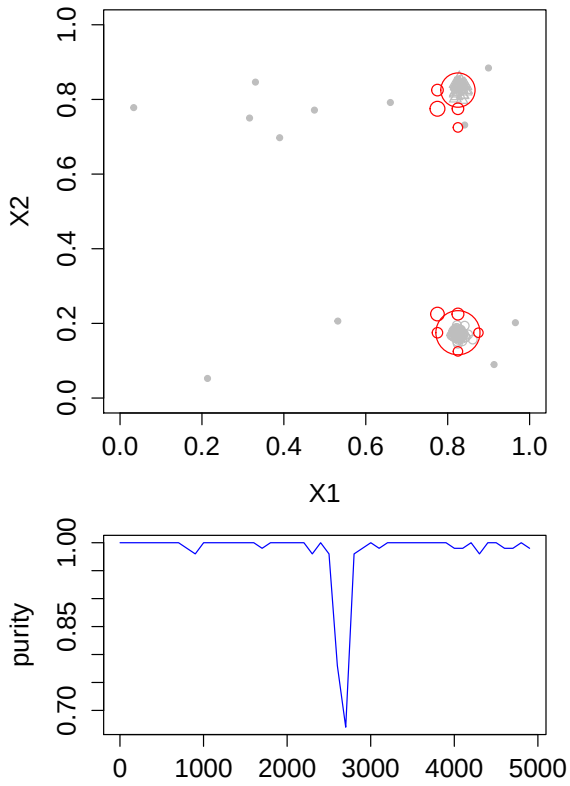


Figure 10: Result of animated clustering with evaluation.

```
R> reset_stream(dsd)
R> micro <- DSC_DStream(gridsize=.05, lambda=.01)
R> r <- animate_cluster(micro, dsd, n=5000,
+     horizon=100, evaluationMeasure="purity", 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 package **animation**.

5.6. Reclustering DSC Objects

This examples show how to recluster a DSC object after creating it. First we create data, a DSC micro-clustering object and run the clustering algorithm.

```
R> dsd <- DSD_Gaussians(k=3, d=2, noise=0.05)
R> dstream <- DSC_DStream(gridsize=.05)
R> cluster(dstream, dsd, 1000)
R> dstream

D-Stream
Class: DSC_DStream, DSC_Micro, DSC_R, DSC
```

```
Number of micro-clusters: 84
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 reason for this can be found by visualizing the clustering.

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

Figure 11(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 that joining adjacent dense grids is not able to separate the two overlapping clusters in the top part of the plot.

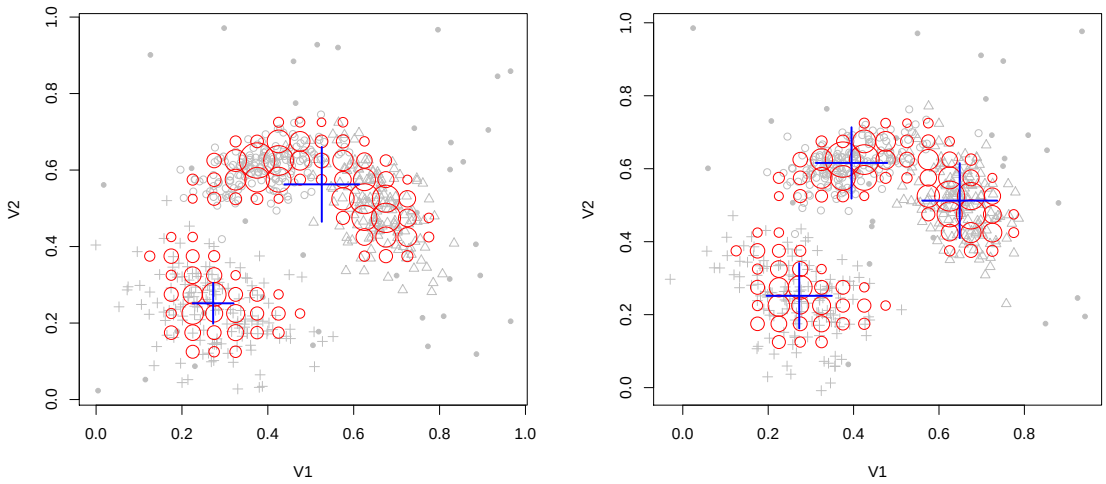
Micro-clusters produced with any clustering algorithm can be reclustered by the `recluster()` method with any available macro-clustering algorithm (sub-classes of `DSD_Macro`) available in **stream**. Some supported macro-clustering models that are typically used for reclustering are k-means, hierarchical 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: 84
Number of macro-clusters: 3

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

Figure 11(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(km, dsd, measure=c("purity", "crand", "SSQ"), n=1000)

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

purity  cRand    SSQ
 0.932  0.859 15.592
```



(a) (b)

Figure 11: A data stream clustered with D-Stream using the (a) built-in reclustering strategy, and (b) reclustered with weighted k-means and $k = 3$.

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

```
R> evaluate(km, dsd, c(measure="purity", "crand", "SSQ"), n=1000, assign="macro")

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

purity  cRand    SSQ
 0.942  0.892 14.180
```

In this case the evaluation measures purity and corrected Rand sightly increase, since D-Stream produces several micro-clusters covering the area between the top two true clusters (see micro-clusters in Figure 11). 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 average within sum of squares decreases because the data points are now directly assigned to minimize this type of error.

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 mining 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 **stream** with new DSD and DST implementations.

6.1. Implementing a New Data Stream Source (DSD)

The class hierarchy in Figure 2 (on page 9) is implemented using the S3 class system (Chambers and Hastie 1992). Class membership and the inheritance hierarchy is represented by a vector of class names stored as the object's 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 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. Note that the class attribute has to contain a vector of all parent classes in the class diagram in bottom-up order.

For a new DSD implementation only the following two functions need to be implemented:

1. A creator function (with a name starting with the prefix `DSD_`) and
2. the `get_points()` method.

The creator function creates an object of the appropriate `DSD` subclass. Typically this S3 object contains 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). Standard parameters are `d` and `k` for the number of dimensions of the created data and the true number of clusters, respectively. In addition an element called `"description"` should be provided. This element is used by `print()`.

The implemented `get_points()` needs to dispatch for the class and create as the output a data.frame containing the new data points as rows. Also, if the ground truth (true cluster assignment as an integer vector; noise is represented by `NA`) is available, then this can be attached to the data.frame as an attribute called `"assignment"`.

For a very simple example, we show here the implementation of `DSD_UniformNoise` available in the package's source code in file `DSD_UniformNoise.R`. This generator creates noise points uniformly distributed in a d-dimensional hypercube with a given range.

```
R> DSD_UniformNoise <- function(d=2, range=NULL) {
+   if(is.null(range)) range <- matrix(c(0,1), ncol=2, nrow=d, byrow=TRUE)
+   structure(list(description = "Uniform Noise Data Stream", d = d,
+     k=NA_integer_, range=range),
+         class=c("DSD_UniformNoise","DSD_R","DSD"))
+   }
R> get_points.DSD_UniformNoise <- function(x, n=1, assignment = FALSE, ...) {
+     data <- as.data.frame(t(replicate(n,
+       runif(x$d, min=x$range[,1], max=x$range[,2]))))
+     if(assignment) attr(data, "assignment") <- rep(NA_integer_, n)
```

```
+     data
+ }
```

The constructor only stores the description, the dimensionality and the range of the data. For this data generator `k`, the number of true clusters, is not applicable. Since all data is random, there is also 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 indicating that the data points are all noise. Several more complicated examples are available in the package's source code directory in files starting with `DSD_`.

6.2. Implementing new Data Stream Tasks (DST)

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 10) for data stream clustering (DSC) can be easily added.

To implement a new clustering algorithm,

1. a creator function (typically named after the algorithm and starting with `DSC_`) which created the clustering object,
2. an implementation of the actual cluster algorithm, and
3. accessors for the clustering

are needed. The implementation depends on the interface that is used. Currently an `R` interface is available as `DSC_R` and a MOA interface is implemented in `DSC_MOA` (in **streamMOA**). The implementation for `DSC_MOA` takes care of all MOA-based clustering algorithms and we will concentrate here on the R interface.

For the R interface, the clustering class needs to contain the elements `"description"` and `"RObj"`. The description needs to contain a character string describing the algorithm. RObj is expected to be a reference class object and contain the following methods:

1. `cluster(newdata, ...)`, where `newdata` is a data.frame with new data points.
2. For micro-clusters: `get_microclusters(...)` and `get_microweights(...)`
3. For macro-clusters: `get_macroclusters(...)`, `get_macroweights` and `microToMacro(micro, ...)` which does micro- to macro-cluster matching.

Note that these are methods for reference classes and do not contain the called object in the parameter list. Neither of these methods are called directly by the user. Figure 4 (on page 11) shows that the function `cluster()` is used to cluster data points, and `get_centers()` and `get_weights()` are used to obtain the clustering. These user facing functions call internally the methods in RObj via the R interface in class `DSC_R`.

For a comprehensive example of a clustering algorithm implemented in R, we refer the reader to `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 the 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 is provided 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 task, but we are working on incorporating 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

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

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

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.

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.

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.

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.

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

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 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.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. URL http://www.jstatsoft.org/v55/i14/.

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.

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.

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.

Maechler M, Rousseeuw P, Struyf A, Hubert M, Hornik K (2014). *cluster: Cluster Analysis Basics and Extensions*. R package version 1.15.2 — For new features, see the 'Changelog' file (in the package source).

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

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

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.

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.

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.

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
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