Introduction to stream: A Framework for Data Stream Mining Research

John Forrest
Microsoft

Matthew Bolaños
Southern Methodist University

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 data mining 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**, an R package that provides an intuitive interface for experimenting with data streams and data stream mining algorithms. **stream** is 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.

Keywords: data stream, data mining, clustering.

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.

1. Introduction

Typical statistical and data mining methods (e.g., parameter estimation, statistical tests, clustering, 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, 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 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. This type of data is called a data stream 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 statistical 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 arriving data.
- **Concept drift:** The algorithm has be able to deal with a data generation process which evolves over time (e.g., distributions change or new structure in the data appears).

Obviously, most existing algorithms designed for static data are not able to satisfy 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 is an ideal platform to develop and test prototypes for data stream algorithms, R currently does not have an infrastructure to support data streams.

1. Data sets are typically represented by data.frames or matrices which is suitable for static data but not to represent streams.
2. Algorithms for data streams are not available in R.

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 algorithms (focusing on clustering) 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 algorithms and how to extend the framework with new data stream sources and algorithms. We provide comprehensive examples in Section 5 and conclude with Section 6.

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 introduce 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 groups (typically k) such that point within each group are more similar than 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). However, the standard algorithms for these methods 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.

In the last 10 years many algorithms for clustering data streams have been proposed Guha, Meyerson, Mishra, Motwani, and O'Callaghan (2003); Aggarwal, Han, Wang, and Yu (2003a, 2004a); Cao, Ester, Qian, and Zhou (2006a); Tasoulis, Adams, and Hand (2006); Tasoulis, Ross, and Adams (2007); Udommanetanakit, Rakthanmanon, and Waiyamai (2007); Tu and Chen (2009); Wan, Ng, Dang, Yu, and Zhang (2009); Kranen, Assent, Baldauf, and Seidl (2011). 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 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 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 *et al.* (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 only state

that they use a conventional clustering algorithm (typically k-means or 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 2004b) based on micro-clusters found with the data-stream clustering algorithm CluStream. 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 both stream classification and stream 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 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 clustering (for two-dimensional data) and the change in 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 only partially possible or very difficult.

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), **clusterGeneration** (Qiu and Joe. 2009), and several others. **stream** also interfaces the data stream clustering algorithms already available in MOA using the **rJava** package by Urbanek (2010). Furthermore, **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 easily be added.

For **stream** rely 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)

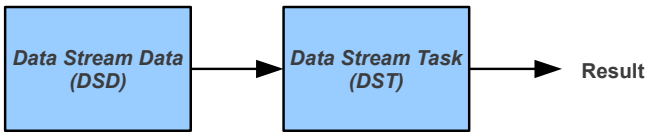


Figure 1: A high level view of the **stream** architecture.

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 on disk or a generator which simulates a data stream with know properties for controlled experiments. Figure 2 shows relationship 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`). **stream** currently provides the following generators:

- `DSD_GaussianStatic`, a DSD that generates static cluster data with a random Gaussian distribution.
- `DSD_GaussianMoving`, a DSD that generates moving cluster data with a Gaussian distribution.
- `DSD_UniformNoise`, generates uniform noise in a d-dimensional (hyper) cube.
- `DSD_mlbenchGenerator`, a class uses the generators for artificial data sets defined in the `mlbench` package.
- `DSD_mlbenchData`, a DSD that wraps data sets found within the `mlbench` package.
- `DSD_Target`, a DSD that generates a ball in circle data set.
- `DSD_BarsAndGaussians`, a DSD that generates two bars and two Gaussians clusters with different density.
- `DSD_RandomRBFGeneratorEvents` (from MOA), a data generator for moving Gaussian clusters which noise which 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`, a class designed to read data from files or open connections. This object also provides support to scale the data points using `scale()`.

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

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

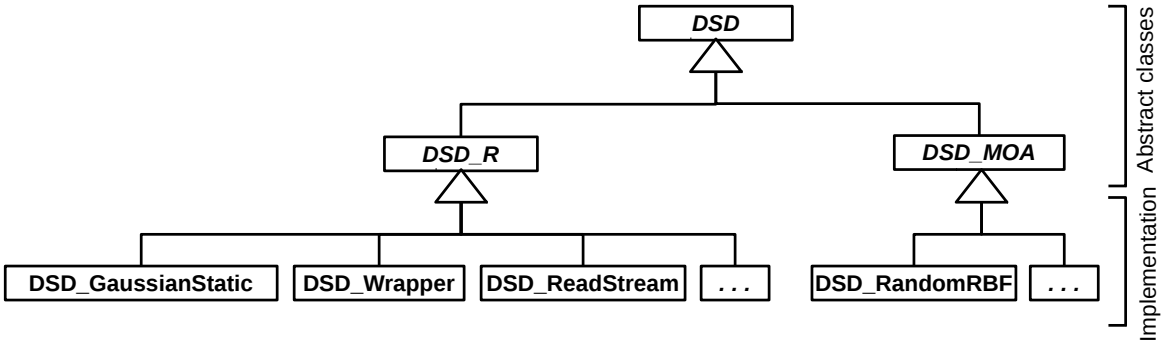


Figure 2: Overview of the DSD class structure.

A DSD can also be scaled by wrapping it in a stream class using:

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

As depicted in the class diagram, other data steam implementations can be easily added in the future.

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. 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. The 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 point(s) are returned as a data.frame with each row representing a single data point.

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 automatically by **stream**. 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.

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 with even currently unknown tasks. Figure 3 shows the class hierarchy for DST. It is important to note that the concept of the DST class is merely for conceptual purposes, the actual implementation of clustering, classification or frequent pattern mining are typically quite different and share

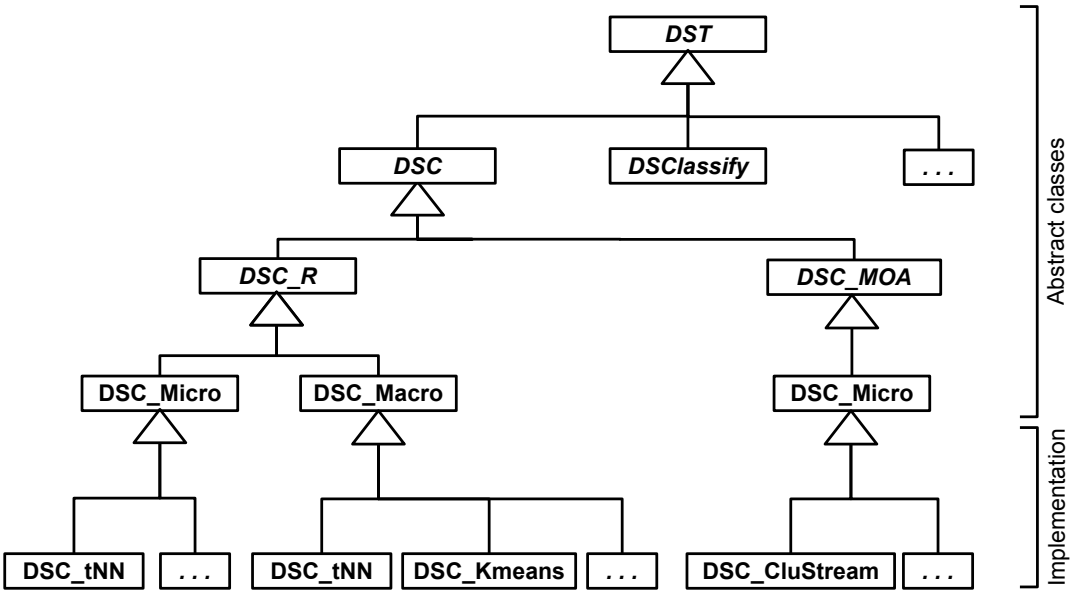


Figure 3: Overview of the DST class structure.

only basic management functionality. We will restrict the following discussion on data stream clustering (DSC) since **stream** currently focus on this task.

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.
- `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.
- `nclusters(x)` which returns the number of clusters currently in the DSC object.
- `get_centers(x, type=c("auto", "micro", "macro"), ...)` returns the centers, either the centroids or the 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. 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 (typically the number of points assigned to the cluster after fading) in the DSC object.

- `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. This function is currently not available for all DSC implementations.
- `plot(x, dsd=NULL, ..., method="pairs", type = c("auto", "micro", "macro")` (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"` simply takes the first two attributes of the matrix and plots it as `x` and `y` on 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- or macro-clusters are plotted.
- `print(x, ...)` prints common attributes of the DSC object (a small 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 will be available (via `get_centers()`, etc.) from the object after clustering. Note that DSC classes are implemented as reference classes and thus the result of cluster does not need to be reassigned to the object. New data points can be assigned to the clusters in the clustering using `get_assignment` resulting in the 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 again into R-based implementations (`DSC_R`) and MOA-based implementations. The following clustering algorithms are available:

- `DSC_Sample`, selects representatives via Reservoir Sampling (Vitter 1985).
- `DSC_BIRCH`, the first pass of the BIRCH (balanced iterative reducing and clustering using hierarchies) algorithm (Zhang, Ramakrishnan, and Livny 1996). used to generate a CF tree where the leave notes are used as micro-clusters.
- `DSC_CluStream`, the CluStream algorithm from MOA (Aggarwal, Han, Wang, and Yu 2003b).
- `DSC_DenStream`, the DenStream algorithm from MOA (Cao, Ester, Qian, and Zhou 2006b).
- `DSC_ClusTree`, the ClusTree algorithm from MOA (Kranen, Assent, Baldauf, and Seidl 2009).

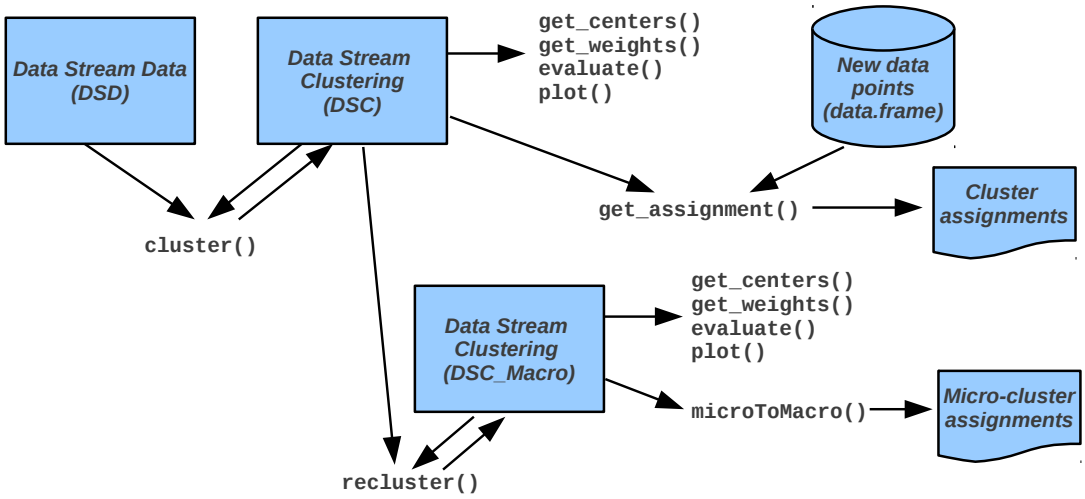


Figure 4: Interaction between the DSD and DSC classes

- `DSC_DStream`, the D-Stream algorithm (Tu and Chen 2009).
- `DSC_tNN`, a simple data stream clustering algorithm called threshold nearest-neighbors (Hahsler and Dunham 2010a,b) extended by a new shared-density graph reclustering method which makes `DSC_tNN` a combined micro- and macro-clustering algorithm.

For reclustering, the following traditional clustering algorithms are available as objects of class `DSC_Macro`:

- `DSC_Kmeans`, interface to R's k-means implementation or a 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_DBSCAN`, DBSCAN (Ester *et al.* 1996).
- `DSC_Hierarchical`, interface to R's `hclust` function.

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 Mining

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

4.1. Evaluating Data Stream Clustering

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 (`assign`), but it is also possible to direct assign them to macro-cluster centers. 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 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 (class label) 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"`).

4.2. 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.

4.3. Implementing new Data Stream Data (DSD) Classes

The class hierarchy in Figure 2 (on page 7) 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_GaussianStatic` will have the class attribute vector `c("DSD_GaussianStatic", "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 has to be implemented in order to add a new data stream implementation.

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 typically is represented by `NA`).

For a complete example, look at the implementation of `DSD_UniformNoise` in the package's source code.

4.4. 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 8) 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 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, look at the implementation of `DSC_tNN` in the package's source code.

5. Examples

Developing 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")
> set.seed(1000)
> dsd <- DSD_GaussianStatic(k=3, d=2, noise=0.05)
> dsd

Static Mixture of Gaussians Data Stream (DSD_GaussianStatic, DSD_R, DSD)
With 3 clusters in 2 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_GaussianStatic` specifying the number of clusters as $k = 4$, the data dimensionality to $d = 2$ and a noise of 5%. 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 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
1 0.7254021 0.44002678
2 0.2642523 0.25574463
3 0.3684851 0.13816977
4 0.6470814 0.52239215
5 0.4239730 0.08627291
```

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

Figure 5 shows the resulting plot. The assignment value is automatically used to change the color in the plot. Noise points are plotted as gray crosses.

Stream also supports data streams which contain concept drift. An example for such a data stream generator is MOA's `DSD_RandomRBFGeneratorEvents` where clusters move over time.

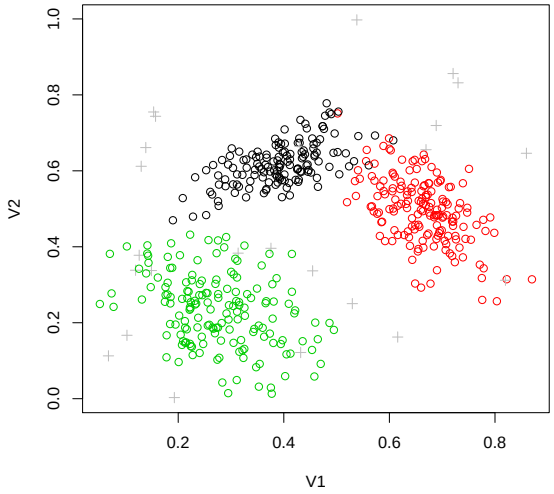


Figure 5: Plotting 1000 data points from the data stream

```
> set.seed(1000)
> dsd <- DSD_RandomRBFGeneratorEvents(k=3, d=2)
> dsd

Random RBF Generator Events (MOA) (DSD_RandomRBFGeneratorEvents, DSD_MOA, DSD)
With 3 clusters in 2 dimensions
```

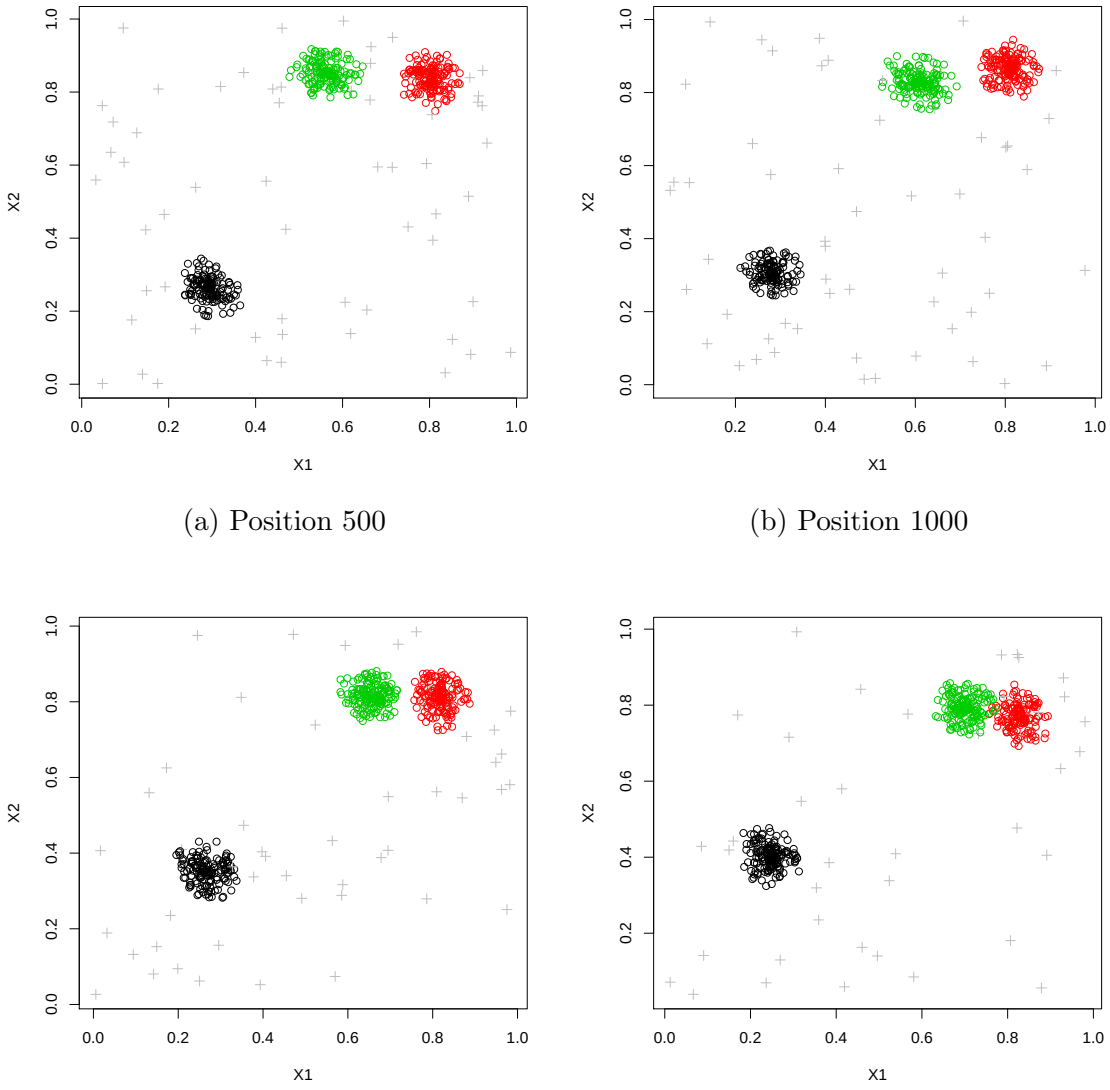
k and d represent the number of clusters and the dimensionality of the data, respectively. To show concept drift, we request four times 500 data points from the stream and plot them.

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

Figure 6 shows the four plots where clusters move over time. An animation of the data can also be generated using `animate_data()`. The animation is recoded using package **animation** and can be replayed and saved using `ani.replay()`.

```
> animate_data(dsd, n=2000, pointInterval=100, xlim=c(0,1), ylim=c(0,1))
> library(animation)
> ani.replay()
> saveHTML(ani.replay())
```

5.2. Reading and writing data streams



(a) Position 500

(b) Position 1000

(c) Position 1500

(d) Position 2000

Figure 6: Data points from DSD_RandomRBFGeneratorEvents at different positions in the stream. Note that clusters change position over time.

Although data streams by definition are unbounded and thus storing them long term is typically 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.

```
Static Mixture of Gaussians Data Stream (DSD_GaussianStatic, 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 is 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.

```
> 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
```

`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 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 see 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`.

```
> library("stream")
> set.seed(1000)
> dsd <- DSD_GaussianStatic(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)
> replayer
```

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

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)
```

`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 NA 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).

```
> library("stream")
> set.seed(1000)
> dsd <- DSD_GaussianStatic(k=3, d=2, noise=0.05)
> dsd
```

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

Next, we prepare the clustering algorithm. We use here `DSC_BIRCH` and set the radius (BIRCH's closeness criterion) to 0.01.

```
> birch <- DSC_BIRCH(radius=0.01)
> birch

BIRCH (DSC_BIRCH, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 0
```

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

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

BIRCH (DSC_BIRCH, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 35
```

After clustering 500 data points data the clustering contains 35 micro clusters. The micro cluster centers are:

```
> head(get_centers(birch))

         V1        V2
1 0.2097448 0.2734255
2 0.1976754 0.1597070
3 0.2840328 0.1478495
4 0.6910668 0.4507183
5 0.6335987 0.5383234
6 0.2950907 0.5313310
```

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

```
> plot(birch, dsd)
```

The resulting plot is shown in Figure 7. The micro-clusters are plotted in red on top of grey data points.

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 BIRCH clustering objects created in the previous section.

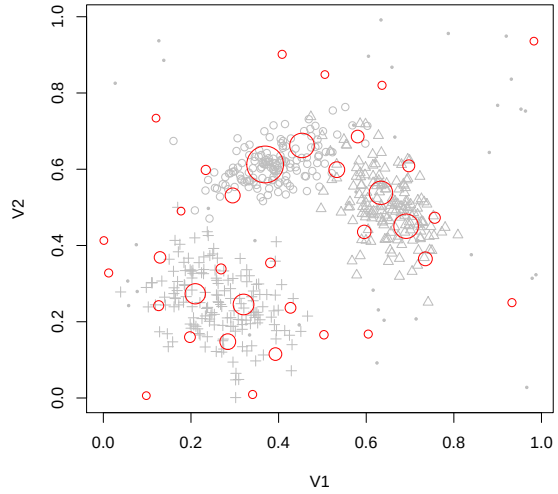


Figure 7: Plotting the micro-clusters produced by BIRCH together with the original data points.

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

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

numMicroClusters numMacroClusters       numClasses        precision
      35.0000000                NA        3.0000000        0.9590517
          recall                F1           purity              fpr
       0.1009528         0.1826765        0.9590517        0.1009528
             SSQ         Euclidean        Manhattan             Rand
      20.2788437         0.1553924        0.2866379        0.7237935
           cRand               NMI               KP            angle
       0.2177155         0.5660987        0.3452754        0.2866379
            diag                FM          Jaccard               PS
       0.2866379         0.4118375        0.1756779        0.1109869
     classPurity
       0.1009528
```

The number of points used for the evaluation are passed on as the argument n. Note that the clustering only contains micro-clusters. Therefore `numMacroClusters` shows `NA`.

Individual measures can be calculated using the method argument.

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

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

```
   purity     cRand
0.9600000 0.2324877
```

For evolving streams it is important to evaluate how well the clustering algorithm is able to adapt to the changing cluster structure over time. Several algorithms were evaluated with the scheme presented by Aggarwal *et al.* (2003b), 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()`. The following examples evaluate DenStream and DenStream plus weighted k-means reclustering on an evolving stream.

```
> set.seed(1000)
> dsd <- DSD_RandomRBFGeneratorEvents(k=3, d=2)
> micro <- DSC_CluStream(k=30)
> evaluate_cluster(micro, dsd, method=c("purity","crand"), n=500, horizon= 100)

     points purity     cRand
[1,]    100      1 0.2360001
[2,]    200      1 0.5252657
[3,]    300      1 0.6218078
[4,]    400      1 0.7431882
[5,]    500      1 0.7995586
```

The dynamic clustering can also be visualized using `animate_cluster()`. Animations can be replayed and saved using the package **animation**.

```
> animate_cluster(micro, dsd, evaluationMethod="crand", n=2000, horizon=100,
+   xlim=c(0,1), ylim=c(0,1))
> library(animation)
> ani.replay()
> saveHTML(ani.replay())
```

5.6. Reclustering DSC objects with another DSC

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

```
> library("stream")
> set.seed(1000)
> dsd <- DSD_GaussianStatic(k=3, d=2, noise=0.05)
> clustream <- DSC_CluStream(k=30)
> cluster(clustream, dsd, 1000)
```

This will produce micro-clusters which can then be reclustered. To achieve this, simply use the `recluster()` method with a macro cluster. The supported macro clustering models that are typically used for reclustering are k-means, weighted k-means, hierarchical clustering, and DBSCAN. Here we use weighted k-means.

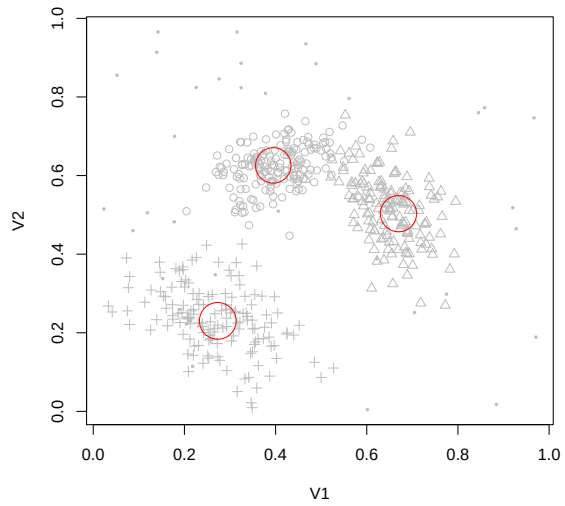


Figure 8: A data stream clustered with Clustream and then reclustered with weighted k-means and $k = 3$.

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

weighted k-Means (DSC_Kmeans, DSC_Macro, DSC_R, DSC)
Number of micro-clusters: 30
Number of macro-clusters: 3
```

We can plot the resulting clustering which is shown in Figure 8.

```
> plot(km, dsd)
```

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

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

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

   purity     cRand
0.9619450 0.8871358
```

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

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

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

   purity     cRand
0.9597458 0.8847793
```

5.7. Full experimental comparison

This example shows the **stream** framework being used from start to finish. It encompasses the creation of data streams, data clusterers, the online clustering of data points as micro-clusters, and then the comparison of offline reclustering.

First, we set up the data. We extract 1000 data points and put them in a `DSD_Wrapper` to make sure that we provide both algorithms with exactly the same data.

```
> library("stream")
> set.seed(1000)
> d <- get_points(DSD_GaussianStatic(k=3, d=2, noise=0.01), n=1000,
+ assignment=TRUE)
> head(d)

          V1        V2
1 0.64111587 0.4704272
2 0.17924327 0.2921518
3 0.08830749 0.2186050
4 0.81291725 0.5317657
5 0.31715474 0.2536838
6 0.62280912 0.6041445

> dsd <- DSD_Wrapper(d, k=3)
> dsd

Data Frame/Matrix Stream Wrapper (DSD_Wrapper, DSD_R, DSD)
With 3 clusters in 2 dimensions
Contains 1000 data points - currently at position 1 - loop is FALSE
```

Next, we create the thre clustering algorithms and cluster the same 1000 data points with all of them. Note that we have to reset the stream before we cluster the data points for the next clustering algorithm.

```
> denstream <- DSC_DenStream(epsilon=0.1)
> clustream <- DSC_CluStream(k=100)
> dstream <- DSC_DStream(gridsize=0.05)
```

```
> cluster(denstream, dsd, 1000)
> reset_stream(dsd)
> cluster(clustream, dsd, 1000)
> reset_stream(dsd)
> cluster(dstream, dsd, 1000)

> denstream

DenStream (DSC_DenStream, DSC_Micro, DSC_MOA, DSC)
Number of micro-clusters: 33
Number of macro-clusters: 2

> clustream

CluStream (DSC_CluStream, DSC_Micro, DSC_MOA, DSC)
Number of micro-clusters: 100

> dstream

DStream (DSC_DStream, DSC_Micro, DSC_R, DSC)
Number of micro-clusters: 52
Number of macro-clusters: 2
```

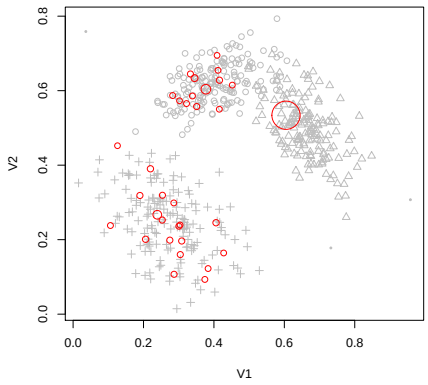
After the clustering operations, we plot the calculated micro-clusters and the original data.

```
> reset_stream(dsd)
> plot(denstream, dsd)

> reset_stream(dsd)
> plot(clustream, dsd)

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

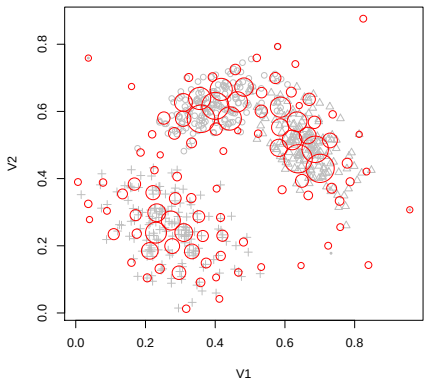
For D-Stream there is an alternative visualization which shows the density of the used grid.

```
> plot(dstream, image=TRUE)
```

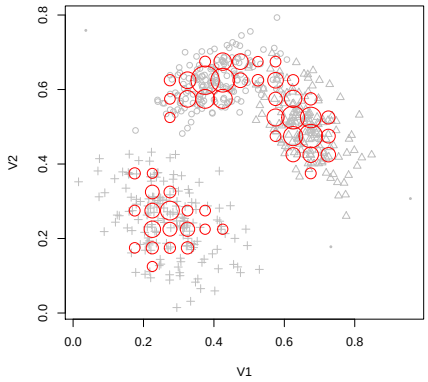
Next, we will recluster the micro-clusters generated by DenStream, CluStream and D-Stream. Different reclustering algorithms were suggested by the creators of each of the algorithm. Althoug, any off-line reclustering strategy can be applied to any on-line clustering algorithm, we will apply here the suggested reclustering strategies. For DenStream weighted DBSCAN is used. Thew parameter eps (reachability distance) is choosen the epsilon-neighborhood parameter of DenStrean. For CluStream we use weighted k-means clustering with k as the actual number of clusters in the data set. For D-Stream finding dense grid clusters is used. Note that for D-Stream we have we have to add a small amount to to the gridsize so that the single-link hierarchical clustering joins adjacent dense grids.



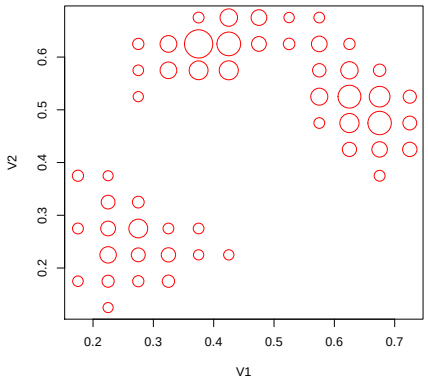
(a) DenStream



(b) CluStream



(c) D-Stream



(d) D-Stream grid

Figure 9: Plotting three sets of different micro-clusters against the generated data

```
> denstream_dbscan <- DSC_DBSCAN(eps=0.1, weighted=TRUE)
> recluster(denstream_dbscan, denstream)
> clustream_kmeans <- DSC_Kmeans(k=3, weighted=TRUE)
> recluster(clustream_kmeans, clustream)
> dstream_grids <- DSC_Hierarchical(h=.05+1e-3, method="single")
> recluster(dstream_grids, dstream)

> reset_stream(dsd)
> plot(denstream_dbscan, dsd)

> reset_stream(dsd)
> plot(clustream_kmeans, dsd)

> reset_stream(dsd)
> plot(dstream_grids, dsd)
```

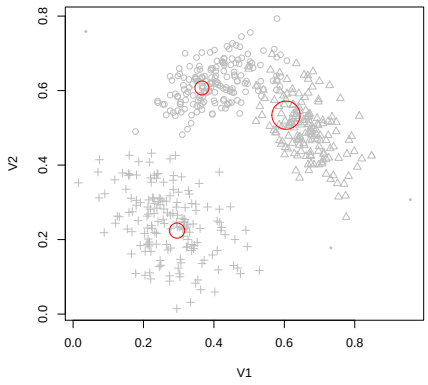
The final clusters are shown in Figure 10. Finally, we can compare the clusterings using `evaluate`.

```
> sapply(list(DenStream=denstream_dbscan,
+   CluStream=clustream_kmeans,
+   DStream=dstream_grids),
+ FUN=function(x){
+     reset_stream(dsd)
+     evaluate(x, dsd, c("purity","rand", "crand"))
+ })

       DenStream CluStream   DStream
purity 0.9757085 0.9757085 0.6690283
Rand   0.9682984 0.9683415 0.7719749
cRand  0.9286157 0.9287125 0.5617859
```

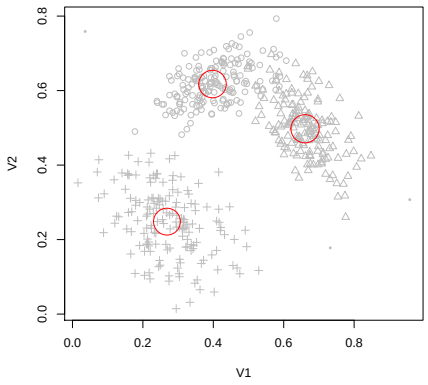
6. Conclusion and Future Work

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.

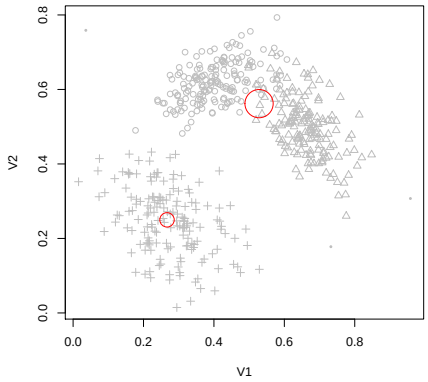
Furthermore, the presented infrastructure can be easily extended by adding new data sources and algorithms. We have abstracted each component to only require a small set of functions that are defined in each base class. Writing the framework in R means that developers have the ability to design components either directly in R, or design components in Java or C/C++, and then write a R light weight wrapper as we did for the MOA algorithms. This allows experimenting with a multitude of algorithms in a consistent way.



(a) DenStream with weighted DBSCAN



(b) CluStream with Weighted k-means



(c) D-Stream with CluStream with dense grid clusters

Figure 10: Reclustering results.

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.

References

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

Aggarwal CC, Han J, Wang J, Yu PS (2003a). "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 (2003b). "A framework for clustering evolving data streams." In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pp. 81–92. VLDB Endowment. ISBN 0-12-722442-4.

Aggarwal CC, Han J, Wang J, Yu PS (2004a). "A Framework for Projected Clustering of High Dimensional Data Streams." In *Proceedings of the International Conference on Very Large Data Bases (VLDB '04)*, pp. 852–863. ISBN 0-12-088469-0.

Aggarwal CC, Han J, Wang J, Yu PS (2004b). "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 twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pp. 1–16. ACM, New York, NY, USA.

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

Cao F, Ester M, Qian W, Zhou A (2006a). "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.

Cao F, Ester M, Qian W, Zhou A (2006b). "Density-based clustering over an evolving data stream with noise." In *In 2006 SIAM Conference on Data Mining*, pp. 328–339.

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

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.

Guha S, Meyerson A, Mishra N, Motwani R, O'Callaghan L (2003). "Clustering data Streams: Theory and Practice." *IEEE Transactions on Knowledge and Data Engineering*, **15**(3), 515–528.

Hahsler M, Dunham MH (2010a). *rEMM: Extensible Markov Model for Data Stream Clustering in R*. R package version 1.0-0., URL http://CRAN.R-project.org/.

Hahsler M, Dunham MH (2010b). "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/.

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

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.

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.

Kranen P, Assent I, Baldauf C, Seidl T (2011). "The ClusTree: indexing micro-clusters for anytime stream mining." *Knowledge and Information Systems*, **29**(2), 249–272.

Last M (2002). "Online classification of nonstationary data streams." *Intelligent Data Analysis*, **6**, 129–147. ISSN 1088-467X.

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 (with specified degree of separation)*. 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.

Tasoulis D, Adams N, Hand D (2006). "Unsupervised Clustering in Streaming Data." In *IEEE International Workshop on Mining Evolving and Streaming Data. Sixth IEEE International Conference on Data Mining (ICDM 2006)*, pp. 638–642.

Tasoulis DK, Ross G, Adams NM (2007). "Visualising the Cluster Structure of Data Streams." In *Advances in Intelligent Data Analysis VII*, Lecture Notes in Computer Science, pp. 81–92. Springer.

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.

Udommanetanakit K, Rakthanmanon T, Waiyamai K (2007). "E-Stream: Evolution-Based Technique for Stream Clustering." In *ADMA '07: Proceedings of the 3rd international conference on Advanced Data Mining and Applications*, pp. 605–615. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-73870-1.

Urbanek S (2010). *rJava: Low-level R to Java interface*. R package version 0.8-8, 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. URL http://doi.acm.org/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.

Zhang T, Ramakrishnan R, Livny M (1996). "BIRCH: An Efficient Data Clustering Method for Very Large Databases." In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 103–114. ACM.

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

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

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