rEMM: Extensible Markov Model for Data Stream Clustering in R

Michael Hahsler and Margaret H. Dunham

Abstract

Clustering streams of continuously arriving data has become an important application of data mining in recent years and efficient algorithms have been proposed by several researchers. However, clustering alone neglects the fact that data in a data stream is not only characterized by the proximity of data points which is used by clustering, but also by a temporal component. The Extensible Markov Model (EMM) adds the temporal component to data stream clustering by superimposing a dynamically adapting Markov Chain. In this paper we introduce the implementation of the R~extension package~**rEMM** which implements EMM on top of a simple clustering algorithm for data streams and discuss some examples and applications.

1 Introduction

Clustering data streams~(Guha, Mishra, Motwani, and O'Callaghan, 2000) has become an important field in recent years. Streams of continuously arriving data are generated by many types of applications including web click-stream data, computer network monitoring data, telecommunication connection data, readings from sensor nets, stock quotes, etc. An important property of data streams for clustering is that data streams often produce massive amounts of data which have to be processed in (or close to) real time since it is impractical to permanently store the data (transient data). This leads to the following requirements:

- The data stream can only be processed in a single pass or scan and typically only in the order of arrival.
- Only a minimal amount of data can be retained and the clusters have to be represented in an extremely concise way.
- Data stream characteristics may change over time (e.g., clusters move, merge, disappear or new clusters may appear).

Many algorithms for data stream clustering have been proposed recently. For example, O'Callaghan, Mishra, Meyerson, Guha, and Motwani (2002)~(see also Guha, Meyerson, Mishra, Motwani, and O'Callaghan, 2003) study the k-median problem. Their algorithm called *LOCALSEARCH* divides the data stream into pieces, clusters each piece individually and then iteratively recluster the resulting centers to obtain a final clustering. Aggarwal, Han, Wang, and Yu (2003) present *CluStream* which uses micro-clusters (an extension of cluster feature vectors used by BIRCH~(Zhang, Ramakrishnan, and Livny, 1996)). Micro-clusters can be deleted and merged and permanently stored at different points in time to allow to create final clusterings (recluster micro-clusters with k-means) for different time frames. Even though CluStream allows clusters to evolve over time and keeps a timestamp, the ordering of arrival of stream data is lost. Kriegel, Kröger, and Gotlibovich (2003) and Tasoulis, Ross, and Adams (2007) present variants of the density based method *OPTICS*~(Ankerst, Breunig, Kriegel, and Sander, 1999) suitable for streaming data. Aggarwal, Han, Wang, and Yu (2004) introduce *HPStream* which finds clusters that are well defined in different subsets of the dimensions of the data. The set of dimensions for each cluster can evolve over time and a fading function is used to discount the influence of older data points by fading the entire cluster structure. Cao, Ester, Qian, and Zhou (2006) introduce *DenStream* which maintains micro-clusters in real time and uses a variant of GDBSCAN~(Sander, Ester, Kriegel, and Xu, 1998) to produce a final clustering for users. Tasoulis, Adams, and Hand (2006) present *WSTREAM,* which uses kernel density estimation to find rectangular windows to represent clusters. The windows can move, contract, expand and be merged over time.

All approaches center on finding clusters of data points based on some notion of proximity, but neglect the temporal structure of the data stream which might be crucial to understanding the underlying processes. For example, for intrusion detection a user might change from behavior A to behavior B, both represented by clusters labeled non-suspicious behavior, but the transition form A to B might be extremely unusual and give away an intrusion event. The Extensible Markov Model~(EMM) originally developed by Dunham, Meng, and Huang (2004) provides a technique to add temporal information in form of an evolving Markov Chain~(MC) to data stream clustering algorithms. Clusters correspond to states in the Markov Chain and transitions represent the temporal information in the data. EMM was successfully applied to rare event and intrusion detection~(Meng, Dunham, Marchetti, and Huang, 2006; Isaksson, Meng, and Dunham, 2006; Meng and Dunham, 2006b), web usage mining~(Lu, Dunham, and Meng, 2006), and identifying emerging events and developing trends~(Meng and Dunham, 2006a,c). In this paper we describe an implementation of EMM in the extension package~**rEMM** for the R~environment for statistical computing~(R Development Core Team, 2005).

Although the traditional Markov Chain is an excellent modeling technique for a static set of temporal data, it can not be applied directly to stream data. As the content of stream data is not known apriori, the requirement of a fixed topology in the MC graph is too restrictive. The dynamic nature of EMM makes it an excellent choice to apply the temporal ordering to data stream clusters. Although there have been a few other approaches to the use of dynamic markov chains ~(Cormack and Horspool, 1987; Ostendorf and Singer, 1997; Goldberg and Mataric, 1999), none of the others provide the complete flexibility needed by stream clustering to create, merge, and delete clusters. Thus the pairing of EMMs and stream clustering algorithms are an ideal marriage.

This paper is organized as follows. In the next section we introduce the concept of EMM and show that all operations needed for adding EMM to data stream clustering algorithms can be performed. Section~3 introduces the simple data stream clustering algorithm implemented in **rEMM**. In Section~4 we discuss implementation details of the package. Sections~5 and 6 provide examples for the package's functionality and apply EMM to analyzing river flow data and to genetic sequences. We conclude with Section~7.

2 Extensible Markov Model

The Extensible Markov Model (EMM) can be understood as an evolving Markov Chain (MC) which at each point in time represents a regular time-homogeneous MC which is updated when new data is available. In the following we will restrict the discussion to first order EMM but, as for a regular MC, it is straight forward to extend EMM to higher order models~(Kijima, 1997).

Markov Chain. A (first order) discrete parameter Markov Chain~(Parzen, 1999) is a special case of a Markov Process in discrete time and with a discrete state space. It is characterized by a sequence of random variables $\{X_t\} =< X_1, X_2, \cdots >$ with t being the time index. All random variables have the same domain $\mathrm{dom}(X_i) = S = \{s_1, s_2, \ldots, s_k\}$, a countable set called the state space. The Markov property states that the next state is only dependent on the current state. Formally,

$$P(X_{l+1} = s \mid X_l = s_l, \ldots, X_1 = s_1) = P(X_{l+1} = s \mid X_l = s_l) \tag{1}$$

For simplicity we use for transition probabilities the notation $a_{ij} = P(X_{l+1} = s_j \mid X_l = s_i)$ where it is appropriate. Time-homogeneous MC can be represented by a graph with the states as vertices and the edges labeled with transition probabilities. Another representation is as a $k \times k$ transition matrix $\mathbf{A}$ containing the transition probabilities from each state to all other states.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \dots & a_{kk} \end{pmatrix} \tag{2}$$

MCs are very useful to keep track of temporal information using the Markov Property as a relaxation. With a MC it is easy to forecast the probability of future states. For example the probability to get from a given state to any other state in n time steps is given by the matrix $\mathbf{A}^n$. With an MC is also easy to calculate the probability of a new sequence of length l as the product of transition probabilities:

$$P(X_l = s_l, X_{l-1} = s_{l-1} \ldots, X_1 = s_1) = \prod_{i=1}^{l-1} P(X_{i+1} = s_{i+1} \mid X_i = s_i) \tag{3}$$

The probabilities of a Markov Chain can be directly estimated from data using the maximum likelihood method by

$$a_{ij} = c_{ij}/n_i, \tag{4}$$

where c_{ij} is the observed count of transitions from s_i to s_j in the data and $n_i = \sum c_{i\cdot}$, the sum of all outgoing transitions from s_i.

Stream Data and Markov Chains. Data streams typically contain dimensions with continuous data and/or have discrete dimensions with a large number of domain values~(Aggarwal, 2009). In addition, the data may continue to arrive resulting in an possibly infinite number of observations. Therefore data points have to be mapped onto a manageable number of states. This mapping is done online as data arrives using data stream clustering where each cluster (or micro-cluster) is represented by a state in the MC. The transition count information is obtained during the clustering process by using an additional data structure efficiently representing the MC transitions. Since it only uses information (assignment of a data point to a cluster) which is created by the clustering algorithm anyway, the computational overhead is minimal. When the clustering algorithm creates, merges or deletes clusters, the corresponding states in the

MC are also created, merged or deleted resulting in the evolving MC, the EMM. Note that k, the size of the set of clusters/states $S = \{s_1, s_2, \ldots, s_k\}$, is not fixed for EMMs and will change over time.

In the following we look at the additional data structures and the operations on these structure which are necessary to add an EMM to a data stream clustering algorithm.

Data Structures for the EMM. Typically algorithms for data stream clustering use a very compact representation for each cluster consisting of a description of the center and how many data points were assigned to the cluster so far. Some algorithms also keep summary information of the dispersion of the data points assigned to each cluster. Since the cluster represents a state in the EMM we need to add a data structure to store the outgoing edges and their counts. For each cluster/state s_i we need to store a transition count vector $c_{i,\cdot}$. Conceptually, all transition counts in an EMM can be seen as a transition count matrix $\mathbf{C}$ composed of all transition count vectors. It is easy to calculate the estimated transition probability matrix from the transition count matrix $\mathbf{C}$ by

$$\mathbf{A} = \mathbf{C}/\mathbf{n}, \qquad (5)$$

where $\mathbf{n} = (n_1, n_2, \ldots, n_k)$ is the vector of row sums of $\mathbf{C}$. Note that $n_i \in \mathbf{n}$ normally is the same as the number of data points assigned to cluster $s_i \in S$ maintained by the clustering algorithm. If we manipulate the clustering using certain operations, e.g., by deleting clusters or fading the cluster structure (see below), the values of $\mathbf{n}$ calculated from $\mathbf{C}$ will diverge from the number of assigned data points maintained by the clustering algorithm. However, this is desirable since it ensures that the probabilities calculated for the transition probability matrix $\mathbf{A}$ stay consistent and keep adding up to unity.

For data with an existing temporal structure, EMM will produce a transition count matrix $\mathbf{C}$ with many of the k^2 entries containing a count of zero. Storing the transition count matrix in a sparse data structure which allows for efficient updating helps to reduce space requirements significantly.

For EMM we need to keep track of the current state $s_c \in \{\epsilon, 1, 2, \ldots, k\}$ which is either no state˜(ϵ; before the first data point has arrived) or the index of one of the k states. We store the transitions from ϵ to the first state in form of an initial transition count vector $\mathbf{c}_\epsilon$ of length k. The initial transition probability vector is calculated by $\mathbf{p}_\epsilon = \mathbf{c}_\epsilon / \sum_{i=1}^{k} c_{\epsilon,i}$. For a single continuous data stream, only one of the elements of $\mathbf{p}_\epsilon$ is one and all others are zero. However, if we have a data stream that naturally should be split into several sequences (e.g., a sequence for each day for stock exchange data), $\mathbf{p}_\epsilon$ is the probability of each state to be the first state in a sequence (see also the genetic sequence analysis application in Section˜6.2).

Thus in addition to the current state s_c there are only two data structures needed by EMM: the transition count matrix, $\mathbf{C}$, and and the initial transition count vector, $\mathbf{c}_\epsilon$. These are only related to maintaining the transition information. No additional data is needed for the clusters themselves.

EMM Clustering Operations. We now define how the operations typically performed by data stream clustering algorithms on (micro-)clusters can be mirrored for the EMM:

- **Adding a data point to an existing cluster.** When a data point is added to an existing cluster s_i, the EMM has to update the transition count from the current state to the new state by setting $c_{s_c,i} = c_{s_c,i} + 1$. Finally the current state is set to the new state by $s_c = i$.

- **Creating a new cluster.** Whenever the clustering algorithm creates a new (micro-)cluster s_{k+1}, the transition count matrix $\mathbf{C}$ has to be enlarged by a row and a column. Since the matrix is stored in a sparse data structure, this modification incurs only minimal overhead.
- **Deleting clusters.** When a cluster s_i (typically an outlier cluster) is deleted by the clustering algorithm, all we need to do is to remove the row i and column i in the transition count matrix $\mathbf{C}$. This deletes the state.
- **Merging clusters.** When two clusters~s_i and s_j are merged into a new cluster s_m, we need to:
  1. Create new state s_m in $\mathbf{C}$ (see creating a new cluster above).
  2. Merge the outgoing edges for s_m by $c_{m.} = c_{i.} + c_{j.}$.
  3. Merge the incoming edges for s_m by $c_{.m} = c_{.i} + c_{.j}$.
  4. Delete columns and rows for the old states s_i and s_j from $\mathbf{C}$ (see deleting clusters above).

  It is straight forward to extend the merge operation to an arbitrary number of clusters at a time. Merging states also covers reclustering which is done by many data stream clustering algorithm to create a final clustering for the user/application.
- **Splitting clusters.** Splitting micro-clusters is typically not implemented in data stream clustering algorithms since the individual data points are not stored and therefore it is not clear how to create two new meaningful clusters. When clusters are "split" by algorithms like BIRCH, it typically only means that one or several micro-clusters are assigned to a different cluster of micro-clusters. This case does not affect the EMM, since the states are attached to the micro-clusters and thus will move with them to the new cluster.

  However, if splitting cluster s_i into two new clusters s_n and s_m is necessary, we can create two states with equal incoming and outgoing transition probabilities by proportionally splitting the counts between s_n and s_m:

  $$\begin{aligned} c_{n.} &= n_n(c_{i.}/n_i)\\ c_{.n} &= n_n(c_{.i}/n_i)\\ c_{m.} &= n_m(c_{i.}/n_i)\\ c_{.m} &= n_m(c_{.i}/n_i) \end{aligned}$$

  After the split we delete s_i.
- **Fading the cluster structure.** Clusterings and EMMs adapt to changes in data over time. New data points influence the clusters and transition probabilities. However, to enable the EMM to learn the temporal structure, it also has to forget old data. Fading the cluster structure is for example used by HP-Stream~(Aggarwal et~al., 2004). Fading is achieved by reducing the weight of old observations in the data stream over time. We use a learning rate λ to specify the weight over time. We define the weight for data that is t timesteps in the past by the following strictly decreasing function:

  $$w_t = 2^{-\lambda t}. \tag{6}$$

  Since data points are not stored, the weighting has to be performed on the transition counts. This is easy since the weight defined above is multiplicative:

  $$w_t = \prod_{i=1}^{t} 2^{-\lambda} \tag{7}$$

and thus can be applied iteratively. This property allows us to fade all transition counts in the EMM by

$$\mathbf{C_{t+1}} = 2^{-\lambda}\,\mathbf{C_t} \quad \text{and}$$
$$\mathbf{c}_{\epsilon_{\mathbf{t+1}}} = 2^{-\lambda}\,\mathbf{c}_{\epsilon_{\mathbf{t+1}}}$$

each time step resulting in a compounded fading effect.

The discussed operations cover all cases typically needed to incorporate EMM into existing data stream clustering algorithm. For example, BIRCH~(Zhang et~al., 1996), CluStream~(Aggarwal et~al., 2003), DenStream~(Cao et~al., 2006) or WSTREAM~(Tasoulis et~al., 2006) can be extended to maintain temporal information in form of an EMM.

Next we introduce the simple data stream clustering algorithm called threshold nearest neighbor clustering implemented in **rEMM**.

3 Data stream clustering

Although the EMM concept can be built on top of any stream clustering algorithm that uses the operations described above, here we discuss a simple algorithm used in our initial R implementation. We use a variation of the Nearest Neighbor (NN) algorithm which instead of always placing a new observation in the closest existing cluster creates a new cluster if no existing cluster is near enough. To specify what near enough means, a threshold value must be provided. We call the algorithm threshold NN. The clusters produced by threshold NN can be considered micro-clusters which can be merged later on in an optional reclustering phase. To represent (micro-)clusters, we use the following information:

- Cluster centers
- Number of data points assigned to the cluster

The cluster centers are either centroids (for Euclidean distance) or pseudo medoids. We use medoids since finding canonical centroids in non-Euclidean space typically has no closed form and therefore is a computationally very expensive optimization problem which needs access to all data points belonging to the cluster~(Leisch, 2006). However, since we do not store the data points for our clusters, even exact medoids cannot be found. Therefore, we use fixed pseudo medoids which we define as the first data point which creates a new cluster. The idea is that since we use a fixed threshold around the center, points will be added around the initial data point which makes it a reasonable center possibly close to the real medoid.

Note, that we do not store the sums and sum of squares of observations like BIRCH~(Zhang et~al., 1996) and similar micro-cluster based algorithms since this only helps with calculating measures meaningful in Euclidean space and the clustering algorithm here is independent from the chosen proximity measure.

Algorithm to add a new data point to a clustering:

1. Compute dissimilarities between the new data point and the k centers.
2. Find the closest cluster with a dissimilarity smaller than the threshold.
3. If such a cluster exists then assign the new point to the cluster.
4. Otherwise create a new cluster for the point.

To observe memory limitations, clusters with very low counts (outliers) can be removed or close clusters can be merged during clustering.

The clustering produces a set of micro-clusters. These micro-clusters can be directly used for an application or they can be recluster to create a final clustering to present to a user or to be used by an application. For reclustering, the micro-cluster centers are treated as data points and clustered by an arbitrary algorithm (hierarchical clustering, k-means, k-medoids, etc.). This choice in clustering algorithm gives the user the flexibility to accommodate apriori knowledge about the data and the shape of expected clusters. For example for spherical cluster k-means or k-medoids can be used and if clusters of arbitrary shape are expected, hierarchical clustering with single linkage make sense.

4 Implementation details

Package~**rEMM** implements the simple clustering algorithm threshold NN described above with an added temporal EMM layer. The package uses the S3 class system and builds on the infrastructure provided by the packages~**proxy**~(Meyer and Buchta, 2009) for dissimilarity computation, **cluster**~(Maechler, Rousseeuw, Struyf, and Hubert, 2009) for clustering, **graph**~(Gentleman, Whalen, Huber, and Falcon, 2009) to represent and manipulate the Markov chain as a graph, and **Rgraphviz**~(Gentry, Long, Gentleman, Falcon, Hahne, and Sarkar, 2009) for one of the visualization options.

The central class in the package is EMM which contains the clustering information used by threshold NN as

- Used dissimilarity measure
- Dissimilarity threshold for micro-clusters
- An indicator if centroids or pseudo medoids are used
- The cluster centers as a $k \times d$ matrix containing the centers (d-dimensional vectors) for the k clusters currently used. Note that k changes over time when clusters are added or deleted.
- The state count vector $\mathbf{n}$ with the number of data points currently assigned to each cluster.

and the EMM layer by

- The Markov Chain as an object of class graphNEL. The directed graph is represented by nodes and an edge list (see package~**graph**) and represents the transition count matrix~$\mathbf{C}$ in a sparse format which can efficiently be manipulated.
- Current state~s_c as a state index. `NA` represents no state~(ϵ).
- Initial transition count vector $\mathbf{c}_\epsilon$.

An EMM~object is created by the creator function `EMM()` which initializes an empty clustering with an EMM layer. To access information about the clustering, we provide the functions `size()` (number of states/clusters), `states()` (name of states/clusters), `state_counts()` (number of observations assigned to each state/cluster) and `state_centers()` (centroids/medoids of clusters). The EMM information in the EMM object can be accessed using `current_state()` (get current state), `transition()` (access count or probability of a certain transition), `transition_matrix()` (compute a transition count or probability matrix), `initial_transition()` (get initial transition count vector).

Clustering and building the EMM is integrated in the function~`build()`. It adds new data points by first clustering and then updating the MC structure. For convenience, `build()` can be called with several data points as a matrix, however, internally the data points (rows) are processed sequentially. To speed up processing, `build()` directly manipulated the `graphNEL` data structure without using the functions provides in **graph**. This removes the overhead of copying the whole graph data structure for each manipulation.

To process multiple sequences, `reset()` is provided. It sets the current state s_c to no state (ϵ). The next observation will start a new sequence and the initial transition count vector will be updated. For convenience, a row of all `NA` in a sequence of data points supplied to `build()` as a matrix also works as a reset.

rEMM implements clustering structure fading by two mechanisms. First, `build()` has a learning rate parameter `lambda`. If this parameter is set, `build()` automatically fades all counts before a new data point is added. The second mechanism is to explicitly call the function~`fade()` whenever fading is needed. This has the advantage that the overhead of manipulating all counts in the EMM can be reduced and that fading can be used in a more flexible manner. For example, if the data points are arriving at an irregular rate, `fade()` could be called at regular time intervals (e.g., every second).

To manipulate states/clusters and transitions, **rEMM** offers a wide array of functions. `remove_states()` and `remove_transitions()` remove user specified states/clusters or transitions from the model. To find rare states or transitions with a count below a specified threshold `rare_states()` and `rare_transitions()` can be used. `prune()` combines finding rare states or transitions and removing them into a convenience function. For some applications transitions from a state to itself might be not interesting. These transitions can be removed by using `remove_selftransitions()`. The last manipulation function is `merge_states()` which combines several clusters/states into a single cluster/state.

As described above, the threshold NN data stream clustering algorithm can use an optional reclustering phase to combine micro-clusters into a final clustering. For reclustering we provide several wrapper functions for popular clustering methods in **rEMM**: `recluster_hclust()` for hierarchical clustering, `recluster_kmeans()` for k-means and `recluster_pam()` for k-medoids. However, it is easy to use any other clustering method. All that is needed is a vector with the cluster assignments for each state/cluster. This vector can be supplied to `merge_states()` with `clustering=TRUE` to create a reclustered EMM. Optionally new centers calculated by the clustering algorithm can also be supplied to `merge_states()` as `new_center`.

Predicting a future state finding the transition probabilites for a new sequence and calculating the probability of a new sequence given an EMM are implemented as `predict()`, `transition_table()` and `score()`, respectively.

The helper function `find_states()` can be used to return the cluster/state sequence for the new sequence. The matching can be nearest neighbor or exact. Nearest neighbor always returns a matching state, while exact will return no state (`NA`) if a data point does not fall within the threshold of any cluster.

Finally, for visualization `plot()` is implemented for class~`EMM`.

In the next section we give some examples of how to use **rEMM** in practice.

5 Examples

5.1 Basic usage

First, we load the package and a simple data set called *EMMTraffic,* which comes with the package and was used by~Dunham et~al. (2004) to illustrate EMMs. Each of the 12 observations in this hypothetical data set is a vector of seven values obtained from sensors located at specific points on roads. Each sensor collects a count of the number of vehicles which have crossed this sensor in the preceding time interval.

```
R> library("rEMM")
R> data(EMMTraffic)
R> EMMTraffic

   Loc_1 Loc_2 Loc_3 Loc_4 Loc_5 Loc_6 Loc_7
1     20    50   100    30    25     4    10
2     20    80    50    20    10    10    10
3     40    30    75    20    30    20    25
4     15    60    30    30    10    10    15
5     40    15    25    10    35    40     9
6      5     5    40    35    10     5     4
7      0    35    55     2     1     3     5
8     20    60    30    11    20    15    10
9     45    40    15    18    20    20    15
10    15    20    40    40    10    10    14
11     5    45    55    10    10    15     0
12    10    30    10     4    15    15    10
```

We use `EMM()` to create a new EMM object using extended Jaccard as proximity measure and an dissimilarity threshold of 0.2. For the extended Jaccard measure pseudo medoids are automatically chosen. Then we build a model using the EMMTraffic data set. Note that `build()` takes the whole data set at once, but this is only for convenience. Internally the data points are processed strictly one after the other in a single pass.

```
R> emm <- EMM(measure = "eJaccard", threshold = 0.2)
R> emm <- build(emm, EMMTraffic)
R> size(emm)

[1] 7
```

The resulting EMM has 7 states. The number of data points represented by each state/cluster can be accessed via `state_counts()`.

```
R> state_counts(emm)

1 2 3 4 5 6 7
2 3 1 2 2 1 1
```

State˜2 has with a count of three the most assigned data points. The state/cluster centers can be inspected using `state_centers()`.

```
R> state_centers(emm)

  Loc_1 Loc_2 Loc_3 Loc_4 Loc_5 Loc_6 Loc_7
1    20    50   100    30    25     4    10
2    20    80    50    20    10    10    10
3    40    15    25    10    35    40     9
4     5     5    40    35    10     5     4
5     0    35    55     2     1     3     5
6    45    40    15    18    20    20    15
7    10    30    10     4    15    15    10
```

`plot()` for EMM objects provides several visualization methods. For example as a graph.

```
R> plot(emm, method = "graph")
```

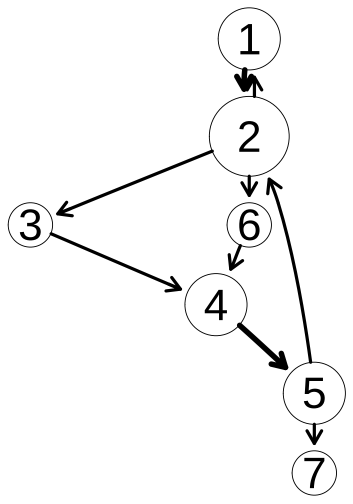


Figure 1: Graph representation of an EMM for the EMMTraffic data set.

The resulting graph is presented in Figure~1. In this representation the vertex size and the arrow width code for the number of observations represented by each state and the transition counts, i.e., more popular states and transitions are more prominently displayed.

The current transition probability matrix of the EMM can be calculated using `transition_matrix()`.

```
R> transition_matrix(emm)

       1   2      3 4 5      6   7
1 0.0000 1.0 0.0000 0 0 0.0000 0.0
2 0.3333 0.0 0.3333 0 0 0.3333 0.0
3 0.0000 0.0 0.0000 1 0 0.0000 0.0
4 0.0000 0.0 0.0000 0 1 0.0000 0.0
5 0.0000 0.5 0.0000 0 0 0.0000 0.5
6 0.0000 0.0 0.0000 1 0 0.0000 0.0
7 0.0000 0.0 0.0000 0 0 0.0000 1.0
```

Alternatively we can get also get the raw transition count matrix.

```
R> transition_matrix(emm, type = "counts")

  1 2 3 4 5 6 7
1 0 2 0 0 0 0 0
2 1 0 1 0 0 1 0
3 0 0 0 1 0 0 0
4 0 0 0 0 2 0 0
5 0 1 0 0 0 0 1
6 0 0 0 1 0 0 0
7 0 0 0 0 0 0 0
```

Individual transition probabilities or counts can be obtained more efficiently via `transition()`.

```
R> transition(emm, "1", "2", type = "probability")

[1] 1
```

Using the EMM model, we can predict a future state given a current state. For example, we can predict the most likely state two time steps away from state˜2.

```
R> predict(emm, n = 2, current = "2")

[1] "4"
```

predict() with probabilities=TRUE produced the probability distribution over all states.

```
R> predict(emm, n = 2, current = "2", probabilities = TRUE)

     1      2      3      4      5      6      7 
0.0000 0.3333 0.0000 0.6667 0.0000 0.0000 0.0000 
```

In this example state˜4 was predicted since it has the highest probability. If several states have the same probability the tie is randomly broken.

5.2 Manipulating EMMs

EMMs can be manipulated by removing states or transitions and by merging states. Figure˜2(a) shows again the EMM for the EMMTraffic data set created above. We can remove a state with remove_states(). For example, we remove state˜3 and display the resulting EMM in Fig˜2(b).

```
R> emm_3removed <- remove_states(emm, "3")
R> plot(emm_3removed, method = "graph")
```

Removing transitions is done with remove_transitions(). In the following example we remove the transition from state˜5 to state˜2 from the original EMM for EMMTraffic in Figure˜2(a). The resulting graph is shown in Fig˜2(c).

```
R> emm_52removed <- remove_transitions(emm, "5", "2")
R> plot(emm_52removed, method = "graph")
```

States can be merged using merge_states(). Here we merge states˜2 and 5 into a combined state. The combined state automatically gets the name of the first state in the merge vector. The resulting EMM is shown in Fig˜2(d).

```
R> emm_25merged <- merge_states(emm, c("2", "5"))
R> plot(emm_25merged, method = "graph")
```

Note that a transition from the combined state˜2 to itself is created which represents the transition from state˜5 to state˜2 in the original EMM.

5.3 Using cluster structure fading and pruning

EMMs can adapt to changes in data over time. This is achieved by fading the cluster structure using a learning rate. To show the effect, we learn an EMM for the EMMTraffic data with a rather high learning rate of $\lambda = 1$. Since the weight is calculated by $w_t = 2^{-\lambda t}$, the observations are weighted $1, \frac{1}{2}, \frac{1}{4}, \ldots$.

```
R> emm_fading <- EMM(measure = "eJaccard", threshold = 0.2,
+     lambda = 1)
R> emm_fading <- build(emm_fading, EMMTraffic)
R> plot(emm_fading, method = "graph")
```

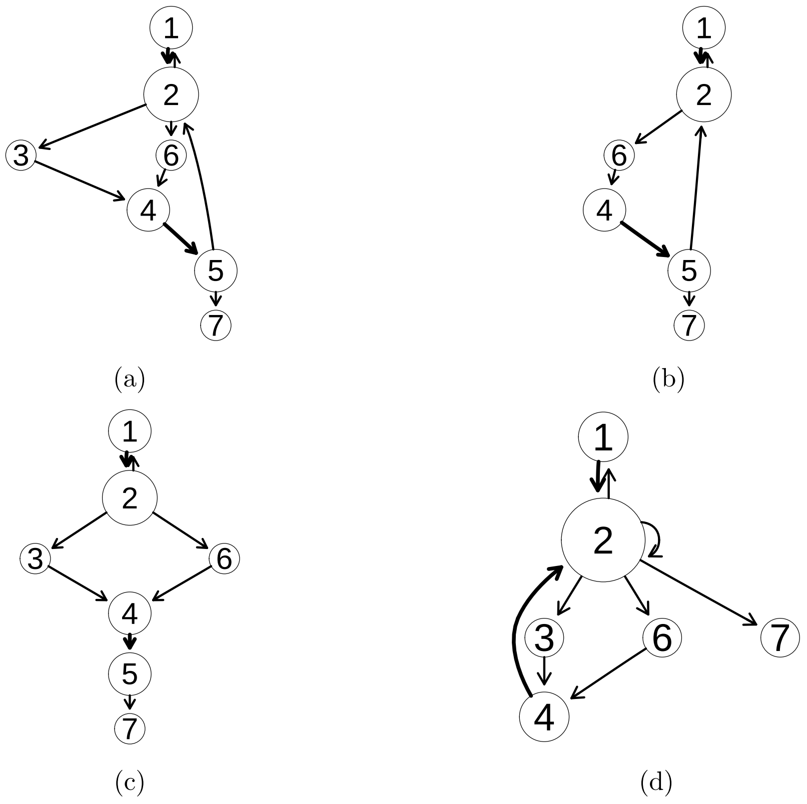


Figure 2: Graph representation for an EMM for the EMMTraffic data set. (a) shows the original EMM, in (b) state˜3 is removed, in (c) the transition from state˜5 to state˜2 is removed, and in (d) states˜2 and 5 are merged.

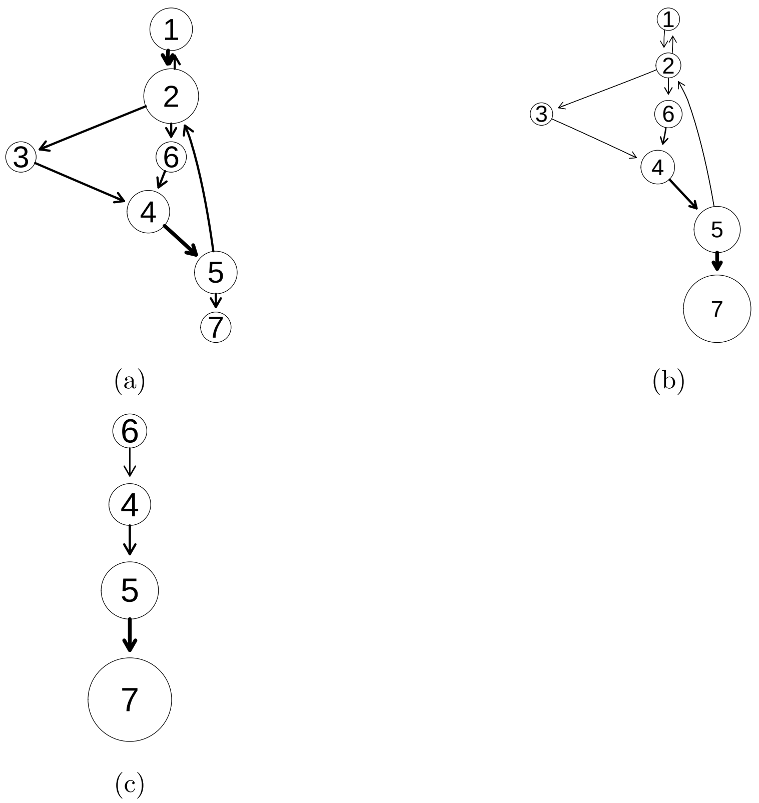


Figure 3: Graph representation of an EMM for the EMMTraffic data set. (a) shows the original EMM. (b) shows an EMM with a learning rate of $\lambda = 1$. (c) EMM with learning rate after pruning with a count threshold of 0.1.

The resulting graph is shown in Figure˜3(b). The states which were created earlier on (states with lower number) are smaller (represent a lower weighted number of observations) compared to the original EMM without fading displayed in Figure˜3(a).

Over time states in an EMM can become obsolete and no new observations are assigned to them. Similarly transitions might become obsolete over time. To simplify the model and improve efficiency, such obsolete states and transactions can be pruned. For the example here, we prune all states and transitions which have a weighted count of less than 0.1 and show the resulting model in Figure˜3(c).

```
R> emm_pruned <- prune(emm_fading, count_threshold = 0.1)
R> plot(emm_pruned, method = "graph")
```

5.4 Visualization options

We use a simulated data set called *EMMsim* which is included in˜**rEMM**. The data contains four well separated clusters in $\mathbb{R}^2$. Each cluster is represented by a bivariate normally distributed random variable $X_i \sim N_2(\mu, \mathbf{\Sigma})$. μ are the coordinates of the mean of the distribution and $\mathbf{\Sigma}$ is the covariance matrix.

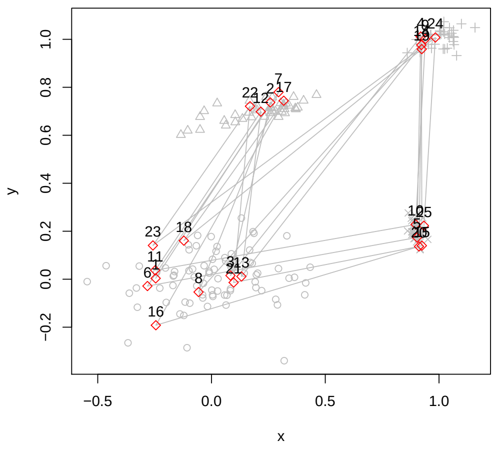


Figure 4: Simulated data set with four clusters. The points of the test data set are plotted in red and the temporal structure is depicted by sequential numbers and lines between the data points.

The temporal structure of the data is modeled by the fixed sequence $< 1, 2, 1, 3, 4 >$ through the four clusters which is repeated 40 times (200 data points) for the training data set and 5 times (25 data points) for the test data.

```
R> data("EMMsim")
```

Since the data set is in 2-dimensional space, we can directly visualize the data set as a scatter plot (see Figure~4). We overlayed the test sequence and to show the temporal structure, the points in the test data are numbered and lines connecting the point in sequential order.

```
R> plot(EMMsim_train, col = "gray", pch = EMMsim_sequence_train)
R> lines(EMMsim_test, col = "gray")
R> points(EMMsim_test, col = "red", pch = 5)
R> text(EMMsim_test, labels = 1:nrow(EMMsim_test), pos = 3)
```

We create an EMM by clustering using Euclidean distance and a threshold of 0.1.

```
R> emm <- EMM(measure = "euclidean", threshold = 0.1)
R> emm <- build(emm, EMMsim_train)
R> plot(emm, method = "graph")
```

The EMM visualized as a graph is shown in Figure~5(a). The positions of the vertices of the graph are solely chosen to optimize the layout which results in a not very informative visualization. The next visualization method (`method="MDA"` which is the default method for `plot()`) uses the relative position of the vertices to represent the proximity between the centers of the states.

```
R> plot(emm)
```

This results in the visualization in Figure~5(b) which shows the same EMM graph but the relative position of the vertices was determined in a way to preserve the proximity information between the centers of the states they represent as much as possible.

A 2-dimensional layout is computed using multidimensional scaling~(MDA, Cox and Cox, 2001). The size of the states and the width of the arrows represent again state and transition counts.

We can also project the points in the data set into 2-dimensional space and then add the centers of the states (see Figure~5(c)).

```
R> plot(emm, data = EMMsim_train)
```

The simple graph representation in Figure~5(a) shows a rather complicated graph for the EMM. However, Figure~5(b) with the vertices positioned to represent dissimilarities between state centers shows more structure. The states clearly fall into four groups. The projection of the state centers onto the data set in Figure~5(c) shows that the four groups represent the four clusters in the data where the larger clusters are split into more states (micro-clusters). We will introduce reclustering to simplify the structure in a later section.

5.5 Scoring new sequences

A score of how likely it is that a sequence was generated by a given EMM model can be calculated by the length-normalized product or sum of probabilities on the path along the new sequence. The scores for a new sequence of length l are defined as:

$$P_{\text{prod}} = \sqrt[l-1]{\prod_{i=1}^{l-1} a_{s(i),s(i+1)}} \tag{8}$$

$$P_{\text{sum}} = \frac{1}{l-1} \sum_{i=1}^{l-1} a_{s(i),s(i+1)} \tag{9}$$

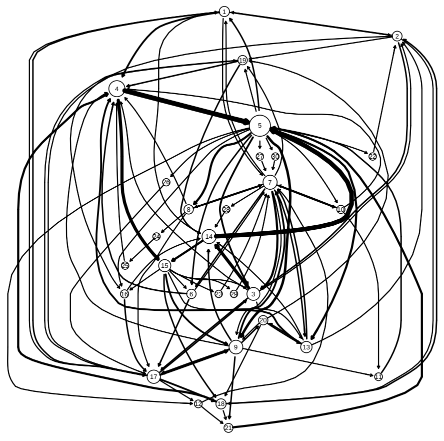
where $s(i)$ is the state the i^{th} data point in the new sequence is assigned to. Points are assignment to the closest cluster only if the distance to the center it is smaller than the threshold. Data points which are not within the threshold of any cluster stay unassigned. Note that for a sequence of length l we have $l-1$ transitions. If we want to take the initial transition probability also into account we extend the above equations by the additional initial probability $a_{\epsilon,s(1)}$:

$$P_{\text{prod}} = \sqrt[l]{a_{\epsilon,s(1)} \prod_{i=1}^{l-1} a_{s(i),s(i+1)}} \tag{10}$$

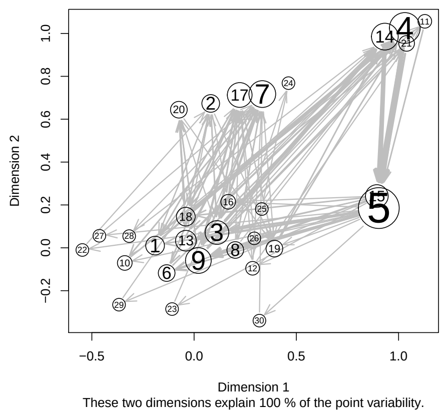
$$P_{\text{sum}} = \frac{1}{l}\left(a_{\epsilon,s(1)} + \sum_{i=1}^{l-1} a_{s(i),s(i+1)}\right) \tag{11}$$

As an example, we calculate how well the test data fits the EMM created for the EMMsim data in the section above. The test data is supplied together with the training set in **rEMM**.

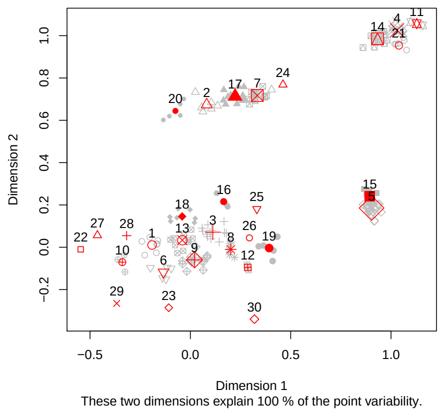
```
R> score(emm, EMMsim_test, method = "prod", match_state = "exact",
+     plus_one = FALSE, initial_transition = FALSE)

[1] 0

R> score(emm, EMMsim_test, method = "sum", match_state = "exact",
+     plus_one = FALSE, initial_transition = FALSE)

[1] 0.227
```



(a)



(b)



(c)

Figure 5: Visualization of the EMM for the simulated data. (a) As a simple graph. (b) A graph using vertex placement to represent dissimilarities. (c) Projection of state centers onto the simulated data.

Even though the test data was generated using exactly the same model as the training data, the normalized product~(`method="prod"`) produces a score of 0 and the normalized sum~(`method="sum"`) is also low. To analyze this problem we can look at the transition table which is used to calulate the score for the test sequence. The transition table is computed by `transition_table()`.

```
R> transition_table(emm, EMMsim_test, match_state = "exact",
+     plus_one = FALSE)

   from   to    prob
1     1   17 0.14286
2    17    3 0.15385
3     3   14 0.58333
4    14    5 0.73333
5     5   10 0.03571
6    10    7 0.33333
7     7    9 0.06667
8     9   14 0.14286
9    14   15 0.26667
10   15    1 0.00000
11    1   17 0.14286
12   17    3 0.15385
13    3   14 0.58333
14   14    5 0.73333
15    5 <NA> 0.00000
16 <NA>    7 0.00000
17    7   18 0.00000
18   18   14 0.00000
19   14    5 0.73333
20    5    3 0.10714
21    3   17 0.16667
22   17 <NA> 0.00000
23 <NA>    4 0.00000
24    4   15 0.36842
```

The low score is caused by data points that do not fall within the threshold for any cluster (`<NA>` above) and by missing transitions in the matching sequence of states (counts and probabilities of zero above). These missing transitions are the result of the fragmentation of the real clusters into many micro-clusters (see Figures~5(b) and (c)). Suppose we have two clusters called cluster~A and cluster~B and after an observation in cluster A always an observation in cluster B follows. If now cluster A and cluster B are represented by many micro-clusters each, it is likely that we find a pair of micro-clusters (one in A and one in B) for which we did not see a transition yet and thus will have a transition count/probability of zero.

To reduce the problem of not being able to match a data point to a cluster we can use a nearest neighbor approach instead of exact matching (`match_state="nn"` is the default for `score()` and `transition_table()`). Here a new data point is assigned to the closest cluster even if it falls outside the threshold. The problem with missing transactions can be reduced by starting with a prior distribution of transition probabilities. In the simplest case we start with a uniform transition probability distribution, i.e., if no data is available we assume that the transitions to all other states are equally likely. This can be done by giving each transition an initial count of one and is implemented as the option `plus_one=TRUE` (also the default for `score()`). It is called plus one since

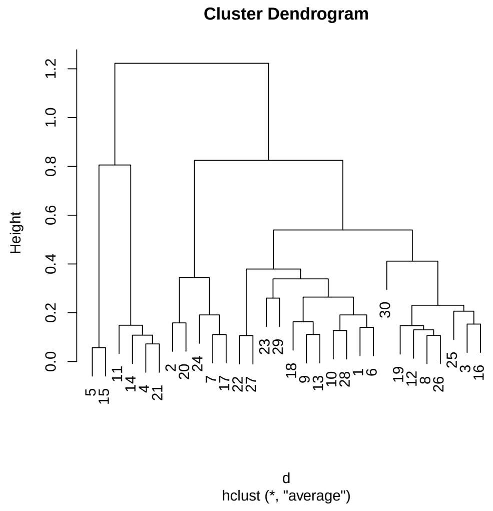


Figure 6: Dendrogram for clustering state centers of the EMM build from simulated data.

one is added to the counts at the time when `score()` is executed. It would be inefficient to store all ones in the data structure for the transition count matrix.

Using nearest neighbor and uniform initial counts of one produces the following scores.

```
R> score(emm, EMMsim_test, method = "prod")
```

```
[1] 0.06982
```

```
R> score(emm, EMMsim_test, method = "sum")
```

```
[1] 0.09538
```

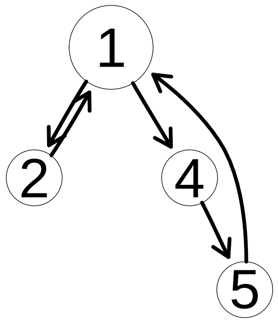
Since we only have micro-clusters, the scores are still extremely small. To get a better model, we will recluster the states in the following section.

5.6 Reclustering states

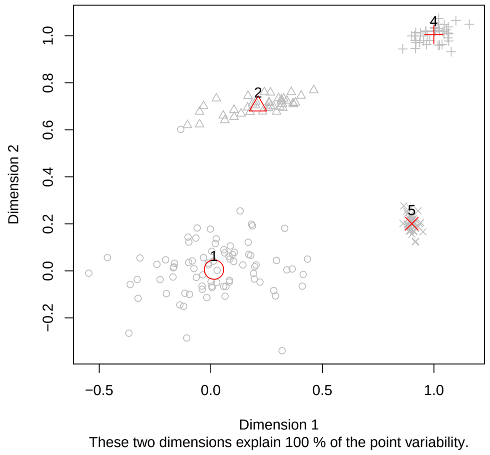
For this example, we use the EMM created in the previous section for the EMMsim data set. For reclustering, we use here hierarchical clustering with average linkage. To find the best number of clusters k, we create clustered EMMs for different values of k and then score the resulting models using the test data.

We use `recluster_hclust()` to create a list of clustered EMMs for $k = 2, 3, \ldots, 10$. Any recluster function in **rEMM** returns with the resulting EMM information about the clustering as the attribute `cluster_info`. Here we plot the dendrogram which is shown in Fig˜6.

```
R> sq <- 2:10
R> emmc <- recluster_hclust(emm, k = sq, method = "average")
R> plot(attr(emmc, "cluster_info")$dendrogram)

R> sc <- sapply(emmc, score, EMMsim_test)
R> names(sc) <- sq
R> sc
```



(a)



(b)

Figure 7: Best performing final clustering for EMM with a k of 4.

```
     2      3      4      5      6      7      8      9     10 
0.5183 0.5668 0.7115 0.5503 0.5373 0.5195 0.4697 0.4140 0.3303 
```

The best performing model has a score of 0.712 for a k of 4. This model is depicted in Figure˜7(a).

Since the four groups in the data set are well separated, reclustering finds the original structure (see Figure˜7(b)) with all points assigned to the correct state.

6 Applications

6.1 Analyzing river flow data

The **rEMM** package also contains a data set called *Derwent* which was originally used by˜Dunham et˜al. (2004). It consists of river flow readings (measured in m^3 per second) from four catchments of in the river Derwent and two of its main tributaries in northern England. The data was collected daily for roughly 5 years (1918 observations) from November˜1,˜1971 to January˜31,˜1977. The catchments are Long Bridge, Matlock Bath, Chat Sworth, What Stand Well, Ashford (river Wye) and Wind Field Park (river Amber).

The data set is interesting since it contains annual changes of river levels and also some special flooding events.

```
R> data(Derwent)
R> summary(Derwent)
```

```
  Long Bridge      Matlock Bath      Chat Sworth     What Stand Well
 Min.   :  2.78   Min.   :  2.61   Min.   : 0.30   Min.   : 0.74   
 1st Qu.:  7.10   1st Qu.:  5.08   1st Qu.: 1.32   1st Qu.: 2.27   
 Median : 10.95   Median :  7.89   Median : 2.16   Median : 3.13   
 Mean   : 14.33   Mean   : 10.64   Mean   : 2.67   Mean   : 4.51   
```

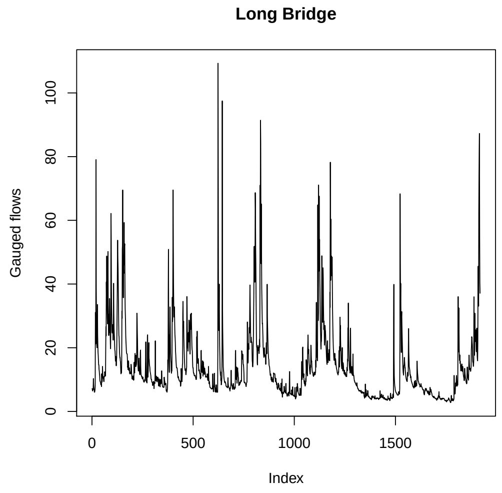


Figure 8: Gauged flows (in m^3/s) of the river Derwent at the Long Bridge catchment.

```
 3rd Qu.: 17.09   3rd Qu.: 12.78   3rd Qu.: 3.45   3rd Qu.: 4.87
 Max.   :109.30   Max.   :104.60   Max.   :16.06   Max.   :72.79

  Wye@Ashford      Amber@Wind Field Park
 Min.   : 0.030   Min.   :  0.010
 1st Qu.: 0.180   1st Qu.:  0.040
 Median : 0.330   Median :  0.090
 Mean   : 0.544   Mean   :  0.143
 3rd Qu.: 0.640   3rd Qu.:  0.160
 Max.   : 6.280   Max.   :  4.160
 NA's   :31.000   NA's   :252.000
```

From the summary we see that the average flows vary among catchments significantly (from 0.143 to 14.238). The influence of differences in averages flows can be removed by scaling the data before building the EMM. Form the summary we also see that for the Ashford and Wind Field Park catchments a significant amount of observations is not available. EMM deals with these missing values by using only the non-missing dimensions of the observations for the proximity calculations (see package~**proxy** for details).

```
R> plot(Derwent[, 1], type = "l", ylab = "Gauged flows",
+     main = colnames(Derwent)[1])
```

In Figure~8 we can see the annual flow pattern for the Long Bridge catchment with higher flows in September to March and lower flows in the summer months. The first year seams to have more variability in the summer months and the second year has an unusual event (around the index of 600 in Figure~8) with a flow above $100m^3/s$ which can be classified as flooding.

We build an EMM from the (centered and) scaled river data using Euclidean distance between the vectors containing the flows from the six catchments and experimentally found a distance threshold of 3 (just above the 3rd quartile of the distance distribution between all scaled observations) to give useful results.

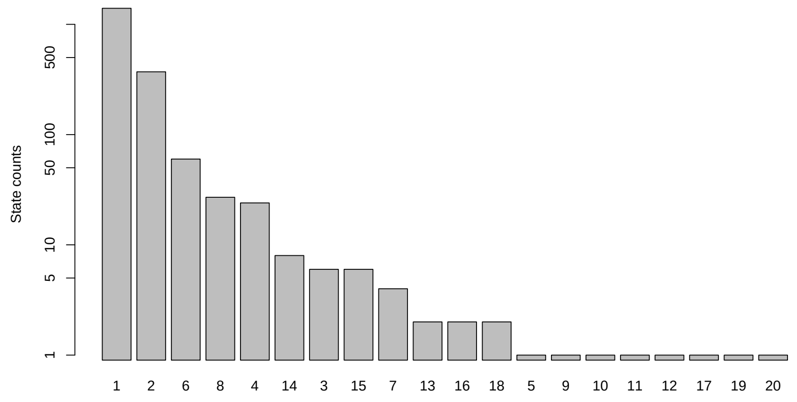


Figure 9: Distribution of state counts of the EMM for the Derwent data.

```
R> Derwent_scaled <- scale(Derwent)
R> emm <- EMM(measure = "euclidean", threshold = 3)
R> emm <- build(emm, Derwent_scaled)
R> plot(emm, method = "state_counts", log = "y")
```

The resulting EMM has 20 states. In Figure~9 shows that the counts for the states have a very skewed distribution with states~1 and 2 representing most observations and stares~5, 9, 10, 11, 12, 17, 19 and 20 being extremely rare.

```
R> plot(emm)
```

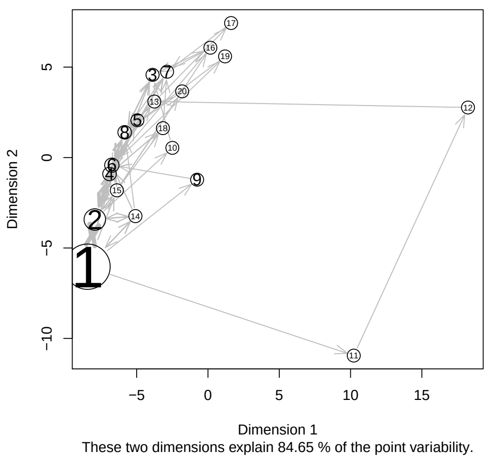
The projection of the state centers into 2-dimensional space in Figure~10(a) reveals that all states except state~11 and 12 are placed closely together.

Next we look at frequent states and transitions. We define rare here as all states/transitions that represent less than 0.5% of the observations. On average this translates into less than two daily observation per year. We calculate a count threshold, use `prune()` to remove rare states/transitions and then we plot the pruned EMM.

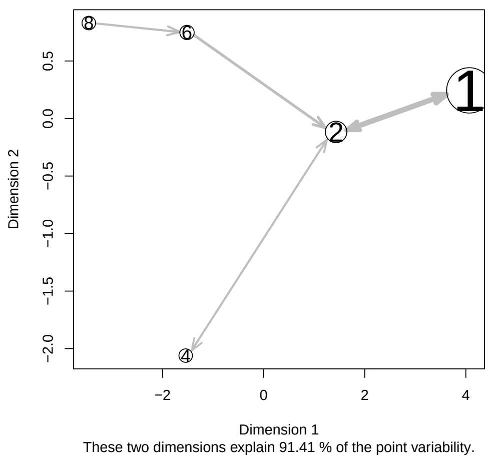
```
R> rare_threshold <- sum(state_counts(emm)) * 0.005
R> rare_threshold

[1] 9.59

R> plot(prune(emm, rare_threshold))
```

The pruned model depicted in Figure~10(b) shows that 5 states represent approximately 99.5% of the river's behavior. All five states come from the lower half of the large cluster in Figure~10(a). States~1 and 2 are the most frequently used states and the wide bidirectional arrow connecting them means that observing transitions between these two states are common. To analyze the meaning of the two outlier states (11 and 12) identified in Figure~10(a) above, we plot the flows at a catchment and mark the observations for these states.

```
R> catchment <- 1
R> plot(Derwent[, catchment], type = "l", ylab = "Gauged flows",
```



(a)



(b)

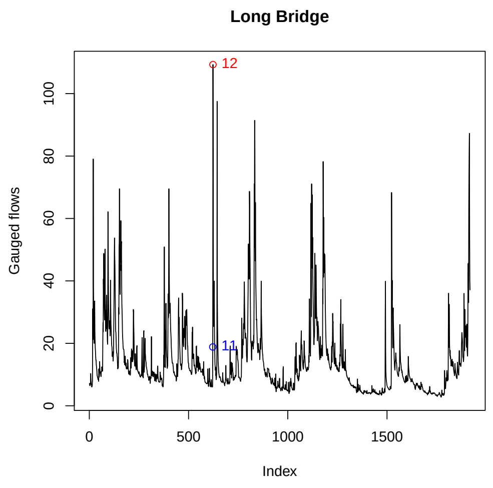
Figure 10: State centers of the EMM for the Derwent data set projected on 2-dimensional space. (a) shows the full EMM and (b) shows a pruned EMM (only the most frequently used states)

```
+     main = colnames(Derwent)[catchment])
R> state_sequence <- find_states(emm, Derwent_scaled)
R> mark_states <- function(states, state_sequence, ys, col = 0,
+     label = NULL, ...) {
+     x <- which(state_sequence %in% states)
+     points(x, ys[x], col = col, ...)
+     if (!is.null(label))
+         text(x, ys[x], label, pos = 4, col = col)
+ }
R> mark_states("11", state_sequence, Derwent[, catchment],
+     col = "blue", label = "11")
R> mark_states("12", state_sequence, Derwent[, catchment],
+     col = "red", label = "12")
```

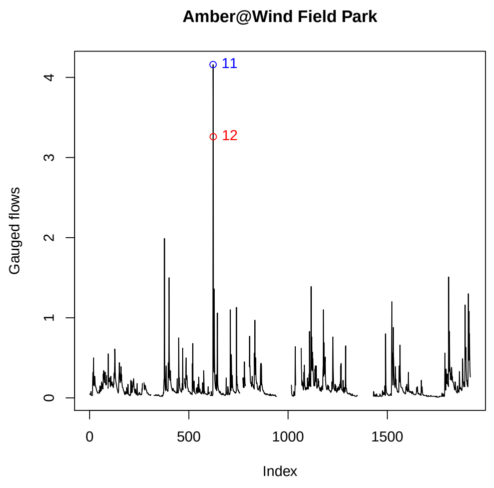
In Figure˜11(a) we see that state˜12 has a river flow in excess of $100m^3/s$ which only happened once in the observation period. State˜11 seems to be a regular observation with medium flow around $20m^3/s$ and it needs more analysis to find out why this state is also an outlier directly leading to state˜12.

```
R> catchment <- 6
R> plot(Derwent[, catchment], type = "l", ylab = "Gauged flows",
+     main = colnames(Derwent)[catchment])
R> mark_states("11", state_sequence, Derwent[, catchment],
+     col = "blue", label = "11")
R> mark_states("12", state_sequence, Derwent[, catchment],
+     col = "red", label = "12")
```

The catchment at Wind Field Park is at the Amber river which is a tributary of the Derwent and we see in Figure˜11(b) that the day before the flood occurs, the flow



(a)



(b)

Figure 11: Gauged flows (in m^3/s) at (a) the Long Bridge catchment and (b) the Amber at the Wind Field Park catchment. Outliers (states˜11 and 12) are marked.

shoots up to $4m^3/s$ which is caught by state˜11. The temporal structure gives a clear sign that a flood is imminent the next day.

6.2 Genetic sequence analysis

The **rEMM** package also contains examples for 16S ribosomal RNA (rRNA) sequences for the two phylogenetic classes, Alphaproteobacteria and Mollicutes. 16S rRNA is a component of the ribosomal subunit 30S and is regularly used for phylogenetic studies˜(e.g., see Wang, Garrity, Tiedje, and Cole, 2007). Typically alignment heuristics like BLAST˜(Altschul, Gish, Miller, Myers, and Lipman, 1990) or a Hidden Markov Model (HMM)˜(e.g., Hughey and Krogh, 1996) are used for evaluating the similarity between two or more sequences. However, these procedures are computationally very expensive.

An alternative approach is to describe the structure in terms of the occurrence frequency of so called n-words, subsequences of length n. Counting the occurrences of the 4^n (there are four different nucleotide types) n-words is straight forward and computing similarities between frequency profiles if very efficient. Because no alignment is computed, such methods are called alignment-free˜(Vinga and Almeida, 2003).

rEMM contains preprocessed sequence data for 30 16S sequences of the phylogenetic class Mollicutes. The sequences were preprocessed by cutting them into windows of length 100 nucleotides without overlap and then for each window the occurrence of triplets of nucleotides was counted resulting in $4^3 = 64$ counts per window. Each window will be used as an observation to build the EMM. The counts for the 30 sequences are organized as a matrix and sequences are separated by rows on `NA` resulting in resetting the EMM during the build process.

Vinga and Almeida (2003) review dissimilarity measures used for alignment-free methods. The most commonly used measures are Euclidean distance, d^2 distance (a weighted Euclidean distance), Mahalanobis distance and Kullback-Leibler discrepancy

(KLD). Since Wu, Hsieh, and Li (2001) find in their experiments that KLD provides good results while it still can be computed as fast as Euclidean distance, it is also used here. Since KLD becomes $-\infty$ for counts of zero, we add one to all counts which conceptually means that we start building the EMM with a prior that all triplets have the equal occurrence probability~(see Wu et~al., 2001). We use an experimentally found threshold of 0.1.

```
R> data("16S")
R> emm <- EMM("Kullback", threshold = 0.1)
R> emm <- build(emm, Mollicutes16S + 1)
```

```
R> plot(emm, method = "graph")
R> it <- initial_transition(emm)
R> it[it > 0]
```

```
      1      23      36      43      47
0.70000 0.06667 0.13333 0.03333 0.06667
```

The graph representation of the EMM is shown in Figure~12 Note that each state in the EMM corresponds to one or more windows of the rRNA sequence (the size of the state indicates the number of windows). The initial transition probabilities show that most sequences start the first count window in state~1. Several interesting observations can be made from this representation.

- There exists a path through the graph using only the largest states and widest arrows which represents the most common sequence of windows.
- There are several places in the EMM where almost all sequences converge (e.g., 4 and 14)
- There are places with high variability where many possible parallel paths exist (e.g., 7, 27, 20, 35, 33, 28, 65, 71)
- The window composition changes over the whole sequences since there are no edges going back or skipping states on the way down.

In general it is interesting that the graph has no loops since Deschavanne, Giron, Vilain, Fagot, and Fertil (1999) found in their study using Chaos Game Representation that the variability along genomes and among genomes is low. However, they looked at longer sequences and we look here at the micro structure of a very short sequence. These observations merit closer analysis by biologists.

7 Conclusion

Temporal structure in data streams is ignored by data stream clustering algorithms. A temporal EMM layer can be used to retain such structure. In this paper we showed that a temporal EMM layer can be added to any data stream clustering algorithm which works with dynamically creating, deleting and merging clusters. As an example, we implemented in **rEMM** a simple data stream clustering algorithm and the temporal EMM layer and demonstrated its usefulness with two applications.

Future work will include extending popular data stream clustering algorithms with EMM and to incorporate higher order models.

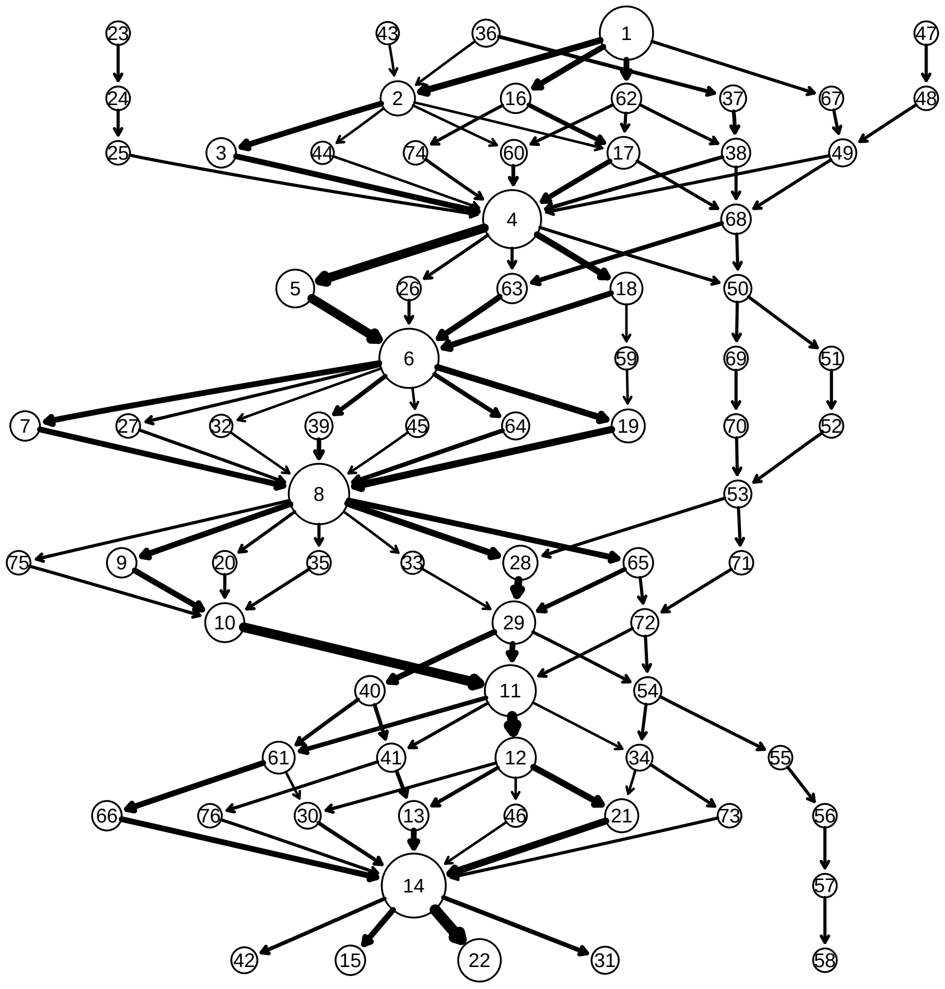


Figure 12: An EMM representing 16S sequences from the class Mollicutes represented as a graph.

References

C.~Aggarwal. A framework for clustering massive-domain data streams. In *IEEE 25th International Conference on Data Engineering (ICDE '09)*, pages 102–113, March 29 2009-April 2 2009.

C.~C. Aggarwal, J.~Han, J.~Wang, and P.~S. Yu. A framework for clustering evolving data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 81–92, 2003.

C.~C. Aggarwal, J.~Han, J.~Wang, and P.~S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 852–863. VLDB Endowment, 2004. ISBN 0-12-088469-0.

S.~F. Altschul, W.~Gish, W.~Miller, E.~W. Myers, and D.~J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.

M.~Ankerst, M.~M. Breunig, H.-P. Kriegel, and J.~Sander. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 49–60, 1999.

F.~Cao, M.~Ester, W.~Qian, and A.~Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pages 328–339. SIAM, 2006.

G.~V. Cormack and R.~N.~S. Horspool. Data compression using dynamic markov modeling. *The Computer Journal*, 30(6), 1987.

T.~Cox and M.~Cox. *Multidimensional Scaling*. Chapman and Hall, 2001.

P.~J. Deschavanne, A.~Giron, J.~Vilain, G.~Fagot, and B.~Fertil. Genomic signature: Characterization and classification of species assessed by chaos game representation of sequences. *Molecular Biology and Evolution*, 16(10):1391–1399, Oct 1999.

M.~H. Dunham, Y.~Meng, and J.~Huang. Extensible markov model. In *Proceedings IEEE ICDM Conference*, pages 371–374. IEEE, November 2004.

R.~Gentleman, E.~Whalen, W.~Huber, and S.~Falcon. **graph***: A Package to Handle Graph Data Structures*, 2009. R package version 1.21.7.

J.~Gentry, L.~Long, R.~Gentleman, S.~Falcon, F.~Hahne, and D.~Sarkar. **Rgraphviz***: Provides Plotting Capabilities for R Graph Objects*, 2009. R package version 1.20.3.

D.~Goldberg and M.~J. Mataric. Coordinating mobile robot group behavior using a model of interaction dynamics. In *Proceedings of the Third International Conference on Autonomous Agents*, May 1999.

S.~Guha, N.~Mishra, R.~Motwani, and L.~O'Callaghan. Clustering data streams. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 359–366, 12-14 Nov. 2000.

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

R.~Hughey and A.~Krogh. Hidden markov models for sequence analysis: Extension and analysis of the basic method. *Computational Applications in Bioscience*, 12(2): 95–107, Apr 1996.

C.~Isaksson, Y.~Meng, and M.~H. Dunham. Risk leveling of network traffic anomalies. *International Journal of Computer Science and Network Security*, 6(6):258–265, June 2006.

M.~Kijima. *Markov Processes for Stochastic Modeling.* Stochastic Modeling Series. Chapman & Hall/CRC, 1997.

H.-P. Kriegel, P.~Kröger, and I.~Gotlibovich. Incremental OPTICS: Efficient computation of updates in a hierarchical cluster ordering. In *Data Warehousing and Knowledge Discovery*, volume 2737 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2003.

F.~Leisch. A toolbox for k-centroids cluster analysis. *Computational Statistics and Data Analysis*, 51(2):526–544, 2006.

L.~Lu, M.~H. Dunham, and Y.~Meng. Mining significant usage patterns from clickstream data. In *Advances in Web Mining and Web Usage Analysis*, volume 4198 of *Lecture Notes in Computer Science*. Springer, August 2006.

M.~Maechler, P.~Rousseeuw, A.~Struyf, and M.~Hubert. **cluster***: Cluster Analysis Basics and Extensions*, 2009. R package version 1.11.13.

Y.~Meng and M.~H. Dunham. Efficient mining of emerging events in a dynamic spatiotemporal environment. In *Advances in Knowledge Discovery and Data Mining*, volume 3918 of *Lecture Notes in Computer Science*, pages 750–754. Springer, 2006a.

Y.~Meng and M.~H. Dunham. Online mining of risk level of traffic anomalies with user's feedbacks. In *Proceedings of the IEEE International Conference on Granular Computing*, pages 176–181, 2006b.

Y.~Meng and M.~H. Dunham. Mining developing trends of dynamic spatiotemporal data streams. *Journal of Computers*, 1(3):43–50, June 2006c.

Y.~Meng, M.~H. Dunham, F.~Marchetti, and J.~Huang. Rare event detection in a spatiotemporal environment. In *Proceedings of the IEEE International Conference on Granular Computing*, pages 629–634, May 2006.

D.~Meyer and C.~Buchta. **proxy***: Distance and Similarity Measures*, 2009. URL `http://CRAN.R-project.org/package=proxy`. R package version 0.4-2.

L.~O'Callaghan, N.~Mishra, A.~Meyerson, S.~Guha, and R.~Motwani. Streaming-data algorithms for high-quality clustering. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 685–. IEEE Computer Society, Mar 2002.

M.~Ostendorf and H.~Singer. Hmm topology desing using maximum likelihood successive state splitting. *Computer Speech and Language*, 11(1):17–41, 1997.

E.~Parzen. *Stochastic Processes.* Society for Industrial Mathematics, 1999.

R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2005. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

J.~Sander, M.~Ester, H.-P. Kriegel, and X.~Xu. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Minining and Knowledge Discovery*, 2(2):169–194, 1998.

D.~Tasoulis, N.~Adams, and D.~Hand. Unsupervised clustering in streaming data. In *Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on*, pages 638–642, Dec. 2006.

D.~K. Tasoulis, G.~Ross, and N.~M. Adams. Visualising the cluster structure of data streams. In *Advances in Intelligent Data Analysis VII*, Lecture Notes in Computer Science, pages 81–92. Springer, 2007.

S.~Vinga and J.~Almeida. Alignment-free sequence comparison—A review. *Bioinformatics*, 19(4):513–523, Mar 2003.

Q.~Wang, G.~M. Garrity, J.~M. Tiedje, and J.~R. Cole. Naive bayesian classifier for rapid assignment of rRNA sequences into the new bacterial taxonomy. *Applied and Environmental Microbiology*, 73(16):5261–5267, Aug 2007.

T.-J. Wu, Y.-C. Hsieh, and L.-A. Li. Statistical measures of DNA sequence dissimilarity under markov chain models of base composition. *Biometrics*, 57(2):441–448, June 2001.

T.~Zhang, R.~Ramakrishnan, and M.~Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114. ACM, 1996.