rEMM: Extensible Markov Model for Data Stream Clustering in R

Michael Hahsler
Southern Methodist University

Margaret H. Dunham
Southern Methodist University

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 and we discuss some examples and applications.

Keywords: data mining, data streams, clustering, Markov chain.

1. Introduction

Clustering data streams (**?**) has become an important field in recent years. A data stream is an ordered and potentially infinite sequence of data points $\langle \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots \rangle$. Such streams of constantly arriving data are generated by many types of applications and include 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, **?** (see also **?**) study the k-medians problem. Their algorithm called *STREAM* divides the data stream into pieces, clusters each piece individually and then iteratively reclusters the resulting centers to obtain a final clustering. **?** present *CluStream* which uses micro-clusters (an

extension of cluster feature vectors used by BIRCH (**?**)). 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, the ordering of the arriving data points in the stream is lost. **?** and **?** present variants of the density based method *OPTICS* (**?**) suitable for streaming data. **?** 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. **?** introduce *DenStream* which maintains micro-clusters in real time and uses a variant of GDBSCAN (**?**) to produce a final clustering for users. **?** 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. More recent density-based data stream clustering algorithms are *D-Stream* (**?**) and *MR-Stream* (**?**). *D-Stream* uses an online component to map each data point into a predefined grid and then uses an offline component to cluster the grid based on density. *MR-Stream* facilitates the discovery of clusters at multiple resolutions by using a grid of cells that can dynamically be sub-divided into more cells using a tree data structure.

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 **?** 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 (**???**), web usage mining (**?**), and identifying emerging events and developing trends (**??**). In this paper we describe an implementation of EMM in the extension package **rEMM** for the R environment for statistical computing (**?**).

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 transition matrix is too restrictive. The dynamic nature of EMM resolves this problem. Although there have been a few other approaches to the use of dynamic Markov chains (**???**), none of the others provide the complete flexibility needed by stream clustering to create, merge, and delete clusters.

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 efficiently. 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.

A previous version of this paper was published in the Journal of Statistical Software (**?**).

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 (**?**).

Markov Chain. A (first order) discrete parameter Markov Chain (**?**) is a special case of a Markov Process in discrete time and with a discrete state space. It is characterized by a sequence $\langle X_1, X_2, \dots \rangle$ of random variables X_t with t being the time index. All random variables have the same domain $\text{dom}(X_t) = S = \{s_1, s_2, \dots, s_K\}$, a set called the state space. The Markov property states that the next state is only dependent on the current state. Formally,

$$P(X_{t+1} = s \mid X_t = s_t, \dots, X_1 = s_1) = P(X_{t+1} = s \mid X_t = s_t) \tag{1}$$

where $s, s_t \in S$. For simplicity we use for transition probabilities the notation

$$a_{ij} = P(X_{t+1} = s_j \mid X_t = s_i)$$

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 it is also easy to calculate the probability of a new sequence of length t as the product of transition probabilities:

$$P(X_t = s_t, X_{t-1} = s_{t-1} \dots, X_1 = s_1) = P(X_1 = s_1) \prod_{i=1}^{t-1} P(X_{i+1} = s_{i+1} \mid X_i = s_i) \tag{3}$$

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 $n_i = \sum_{k=1}^{K} c_{ik}$, 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 (**?**). In addition, the data may continue to arrive resulting in a 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. Because of this one-to-one relationship we use cluster and state for EMM often as synonyms.

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. Note that K, the size of the set of clusters and of states S 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 extend an existing data stream clustering algorithm for EMM.

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 also represents a state in the EMM we need to add a data structure to store the outgoing edges and their counts. For each cluster i representing state s_i we need to store a transition count vector $\mathbf{c}_i$. All transition counts in an EMM can be seen as a transition $K \times K$ 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 (see Equation (4)). Note that n_i in Equation (4) normally is the same as the number of data points assigned to cluster i 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 n_i 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 EMM we also need to keep track of the current state $\gamma \in \{\epsilon, 1, 2, \ldots, K\}$ 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. Note that the superscript is used to indicate that this is the special count vector from ϵ to all existing states. The initial transition probability vector is calculated by $\mathbf{p}^\epsilon = \mathbf{c}^\epsilon / \sum_{k=1}^{K} c_k^\epsilon$. 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 γ 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 i, the EMM has to update the transition count from the current state γ to the new state s_i by setting $c_{\gamma i} = c_{\gamma i} + 1$. Finally the current state is set to the new state by $\gamma = i$.

Creating a new cluster. This operation increases the number of clusters/states from K to $K + 1$ by adding a new (micro-)cluster. To store the transition counts from/to this new cluster, we enlarge the transition count matrix $\mathbf{C}$ by a row and a column which are initialized to zero.

Deleting clusters. When a cluster 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 corresponding state s_i and reduces K to $K - 1$.

Merging clusters. When two clusters i and j are merged into a new cluster m, we need to:

1. Create new state s_m in $\mathbf{C}$ (see creating a new cluster above).
2. Compute the outgoing edges for s_m by $c_{mk} = c_{ik} + c_{jk},\ k = 1, 2, \ldots K$.
3. Compute the incoming edges for s_m by $c_{km} = c_{ki} + c_{kj},\ k = 1, 2, \ldots K$.
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 i into two new clusters n and m is necessary, we replace s_i by the two states, s_n and s_m, with equal incoming and outgoing transition probabilities by splitting the counts between s_n and s_m proportional to n_n and n_m:

$$
\begin{aligned}
c_{nk} &= n_n(c_{ik}/n_i),\ k = 1, 2, \ldots K \\
c_{kn} &= n_n(c_{ki}/n_i),\ k = 1, 2, \ldots K \\
c_{mk} &= n_m(c_{ik}/n_i),\ k = 1, 2, \ldots K \\
c_{km} &= n_m(c_{ki}/n_i),\ k = 1, 2, \ldots K
\end{aligned}
$$

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 HPStream (**?**). Fading is achieved by reducing the weight of old observations in the data stream over time. We use a decay rate $\lambda \geq 0$ 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:

$$w_t = 2^{-\lambda t}. \tag{5}$$

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:

$$w_t = \prod_{i=1}^{t} 2^{-\lambda} \tag{6}$$

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

$$\mathbf{C}_{t+1} = 2^{-\lambda}\, \mathbf{C}_t \quad \text{and}$$
$$\mathbf{c}^{\epsilon}_{t+1} = 2^{-\lambda}\, \mathbf{c}^{\epsilon}_{t}$$

each time step resulting in a compounded fading effect. The exact time of fading is decided by the clustering algorithm. Fading can be used before each new data point is added, or at other regular intervals appropriate for the application.

The discussed operations cover all cases typically needed to incorporate EMM into existing data stream clustering algorithms. For example, BIRCH (**?**), CluStream (**?**), DenStream (**?**) or WSTREAM (**?**) 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 algorithm implemented in **rEMM**.

3. Threshold Nearest Neighbor clustering algorithm

Although the EMM concept can be built on top of any stream clustering algorithm that uses exclusively the operations described above, we discuss here only a simple algorithm used in our initial R implementation. The clustering algorithm applies 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 this algorithm threshold NN (tNN). The clusters produced by tNN 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

In Euclidean space we use centroids as cluster centers since they can be easily incrementally updated as new data points are added to the cluster by

$$\mathbf{z}_{t+1} = n/(n+1)\mathbf{z}_t + 1/(n+1)\mathbf{y}$$

where $\mathbf{z}_t$ is the old centroid for a cluster containing n points, $\mathbf{y}$ is the new data point and $\mathbf{z}_{t+1}$ is the updated centroid for $n+1$ data points (see, e.g., BIRCH by **?**). Finding canonical centroids in non-Euclidean space typically has no closed form and is a computationally expensive optimization problem which needs access to all data points belonging to the cluster (**?**). Since we do not store the data points for our clusters, even exact medoids cannot be found and we have to resort to *fixed pseudo medoids* or *moving pseudo centroids.* We define fixed pseudo medoids 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. As an alternative approach, if we have at least a linear space, we define moving pseudo centroids as the first data point and then, to approximate the adjustment, we apply a simple updating scheme that moves a pseudo centroid towards each new data point that is assigned to its cluster:

$$\mathbf{z}_{t+1} = (1-\alpha)\mathbf{z}_t + \alpha\mathbf{y}$$

where α controls how much the pseudo centroid moves in the direction of the new data point. Typically we use $\alpha = \frac{1}{n+1}$ which results in an approximation of the centroid that is equal to adjustments made for centroids in Euclidean space.

Note, that we do not store the sums and sum of squares of observations like BIRCH (**?**) and similar micro-cluster based algorithms since this only helps with calculating measures meaningful in Euclidean space and the clustering algorithm here is intended to be 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 and adjust the cluster center.
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 reclustered 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 of clustering algorithm gives the user the flexibility to accommodate apriori knowledge about the data and the shape of expected clusters. For example for spherical clusters k-means or k-medoids can be used and if clusters of arbitrary shape are expected, hierarchical clustering with single linkage make sense. Reclustering micro-clusters results in merging the corresponding states in the MC.

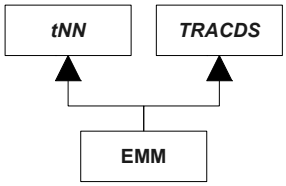


Figure 1: UML class diagram for EMM

4. Implementation details

Package **rEMM** implements the simple data stream clustering algorithm threshold NN (tNN) described above with an added temporal EMM layer. The package uses the S4 class system and builds on the infrastructure provided by the packages **proxy** (**?**) for dissimilarity computation, **cluster** (**?**) for clustering, and **Rgraphviz** (**?**) for one of the visualization options.

The central class in the package is EMM which contains two classes, class tNN which contains all information pertaining to the clustering and class TRACDS (short for temporal relationship among clusters for data streams) for the temporal aspects. Figure 1 shows the UML class diagram (**?**). The advantage of separating the classes is that for future development it is easier to replace the clustering algorithm or perform changes on the temporal layer without breaking the whole system.

Class tNN contains the clustering information used by threshold NN:

- Used dissimilarity measure
- Dissimilarity threshold for micro-clusters
- An indicator if (pseudo) 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 cluster count vector $\mathbf{n} = (n_1, n_2, \ldots, n_K)$ with the number of data points currently assigned to each cluster.

Class TRACDS contains exclusively temporal information:

- The Markov Chain is represented by an object of the internal class SimpleMC which allows for fast manipulation of the transition count matrix $\mathbf{C}$. It also stores the initial transition count vector $\mathbf{c}^\epsilon$.
- Current state γ as a state index. `NA` represents no state (ϵ).

To improve performance we emulate pass-by-reference semantics for objects of the classes EMM, TRACDS and tNN. This is achieved by storing all variable information in the objects inside of an environment which lets to the objects without copying the whole object. For example, for an object of class EMM called `emm` and some new data

```
build(emm, newdata)
```

will change the information inside the `emm` object even though the function's result is not assigned back to `emm`. Note that this means that EMM, TRACDS and tNN objects have to be explicitely copied with the provided `copy()` method if a true (deep) copy is needed.

An EMM object is created by function `EMM()` which initializes an empty clustering with a temporal layer. Several methods are defined for either classe tNN or TRACDS. Only methods which need clustering and temporal information together (e.g., building a new EMM or plotting an EMM) are directly defined for EMM. Since EMM contains tNN and TRACDS, all methods can directly be used for EMM objects. The reason of separation is flexibility for future development.

The temporal layer information from class TRACDS can be accessed using

- `nstates()` (number of states),
- `states()` (names of states),
- `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).

To access information about the clustering from class tNN, we provide the functions

- `nclusters()` (number of clusters),
- `clusters()` (names of clusters),
- `cluster_counts()` (number of observations assigned to each cluster),
- `cluster_centers()` (centroids/medoids of clusters).

For convenience, a method `size()` is provides for EMM which uses `nclusters()` in tNN to return the number of clusters/states in the model.

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

rEMM implements cluster structure fading by two mechanisms. First, `build()` has a decay 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_clusters()` and `remove_transitions()` remove user specified states/clusters or transitions from the model. To find rare clusters or transitions with a count below a specified threshold `rare_clusters()` and `rare_transitions()` can be used. `prune()` combines finding rare clusters or transitions and removing them into a convenience function. For some applications transitions from a state to itself might not be interesting. These transitions can be removed by using `remove_selftransitions()`. The last manipulation function is `merge_clusters()` 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_clusters()` with `clustering=TRUE` to create a reclustered EMM. Optionally new centers calculated by the clustering algorithm can also be supplied to `merge_clusters()` as the parameter `new_center`.

Predicting a future state and calculating the probability of a new sequence are implemented as `predict()` and `score()`, respectively.

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

Finally, `plot()` implements several visualization methods 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 **?** 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 a dissimilarity threshold of 0.2. For the extended Jaccard measure pseudo medoids are automatically chosen (use `centroids = TRUE` in `EMM()` to use pseudo centroids). 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 as a data stream, strictly one after the other in a single pass.

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

[1] 7
```

Note that we do not need to assign the result of `build()` back to `emm`. The information in `emm` is changed by `build()` inside the object since class EMM emulates pass-by-reference semantics.

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

```
R> cluster_counts(emm)

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

Cluster 2 has with a count of three the most assigned data points. The cluster centers can be inspected using `cluster_centers()`.

```
R> cluster_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
```

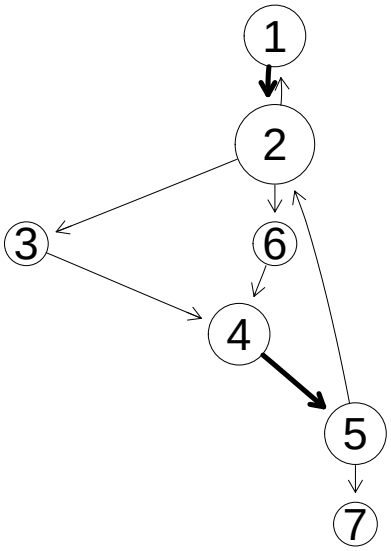


Figure 2: Graph representation of an EMM for the EMMTraffic data set.

`plot()` for EMM objects provides several visualization methods. For example, the default method is as a graph using **igraph**. We use here the method "graph" which uses **Rgraphviz**, a package which has to be installed separately from the Bioconduictor project[1].

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

The resulting graph is presented in Figure 2. 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 clusters 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] http://www.bioconductor.org/

```
  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
```

```
R> #transition_matrix(emm, type="log_odds")
```

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

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

```
[1] 0.3333
```

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

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

```
[1] "4"
```

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

```
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 cluster 4 was predicted since it has the highest probability. If several clusters have the same probability the tie is randomly broken.

5.2. Manipulating EMMs

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

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

Note that a copy of `emm` is explicitely created in order have two independent copies in memory.

Removing transitions is done with `remove_transitions()`. In the following example we remove the transition from cluster 5 to cluster 2 from the original EMM for EMMTraffic in Figure 3(a). The resulting graph is shown in Fig 3(c).

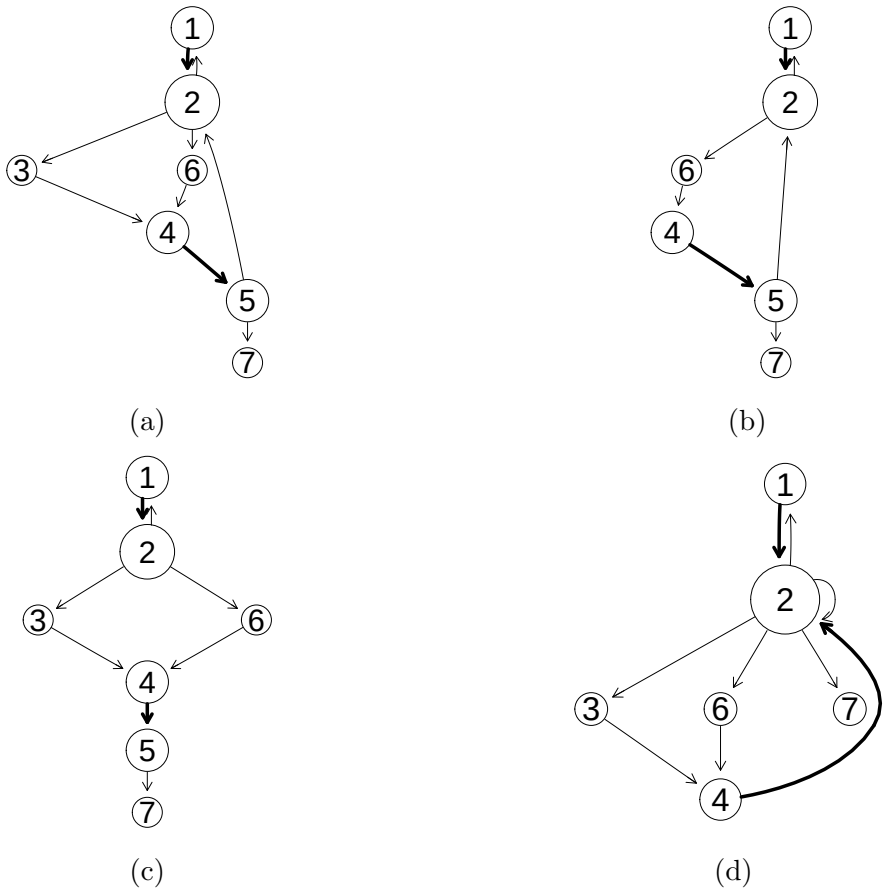


Figure 3: Graph representation for an EMM for the EMMTraffic data set. (a) shows the original EMM, in (b) cluster 3 is removed, in (c) the transition from cluster 5 to cluster 2 is removed, and in (d) clusters 2 and 5 are merged.

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

Here a reference of a copy of `emm` is passed on to `remove_transitions()` and then assigned to `emm_52removed`.

Clusters can be merged using `merge_clusters()`. Here we merge clusters 2 and 5 into a combined cluster. The combined cluster automatically gets the name of the first cluster in the merge vector. The resulting EMM is shown in Fig 3(d).

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

Note that a transition from the combined cluster 2 to itself is created which represents the transition from cluster 5 to cluster 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 decay rate. To show the effect, we train an EMM on the EMMTraffic data with a rather high decay 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(threshold=0.2, measure="eJaccard", lambda = 1)
R> build(emm_fading, EMMTraffic)
R> plot(emm_fading, method="graph")
```

The resulting graph is shown in Figure 4(b). The clusters which were created earlier on (clusters with lower index number) are smaller (represent a lower weighted number of observations) compared to the original EMM without fading displayed in Figure 4(a).

Over time clusters 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 clusters and transitions can be pruned. For the example here, we prune all clusters which have a weighted count of less than 0.1 and show the resulting model in Figure 4(c).

```
R> emm_fading_pruned <- prune(emm_fading, count_threshold=0.1,
+     clusters=TRUE, transitions=TRUE)
R> plot(emm_fading_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.

The temporal structure of the data is modeled by the fixed sequence $\langle 1, 2, 1, 3, 4 \rangle$ 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 5). We overlayed the test sequence to show the temporal structure, the points in the test data are numbered and lines connect the points 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.

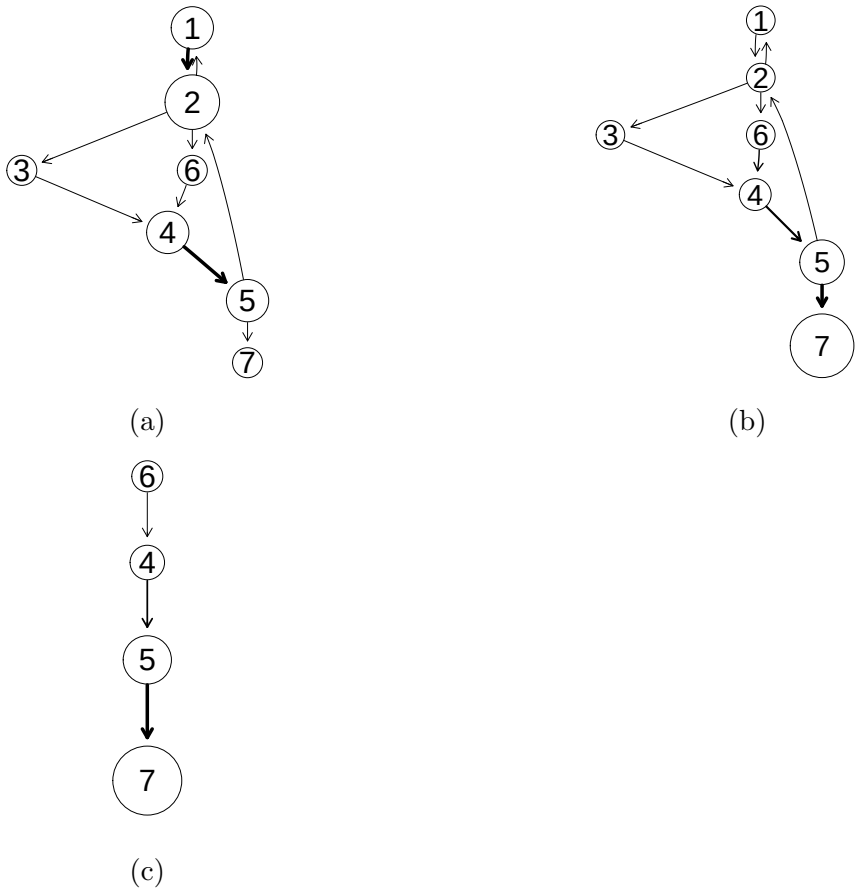


Figure 4: Graph representation of an EMM for the EMMTraffic data set. (a) shows the original EMM. (b) shows an EMM with a learning rate of $\lambda = 1$. (c) EMM with learning rate after pruning with a count threshold of 0.1.

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

The default EMM visualized as a graph using package **igraph** is shown in Figure 6(a).

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

Using method graph the same graph is rendered using the Graphviz library (if package **Rgraphviz** is installed). This visualization is shown in Figure 6(b). In both graph-based visualizations the positions of the vertices of the graph (states/clusters) are solely chosen to optimize the layout which results in a not very informative visualization. The next visualization method uses the relative position of the clusters to represent the proximity between the cluster centers.

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

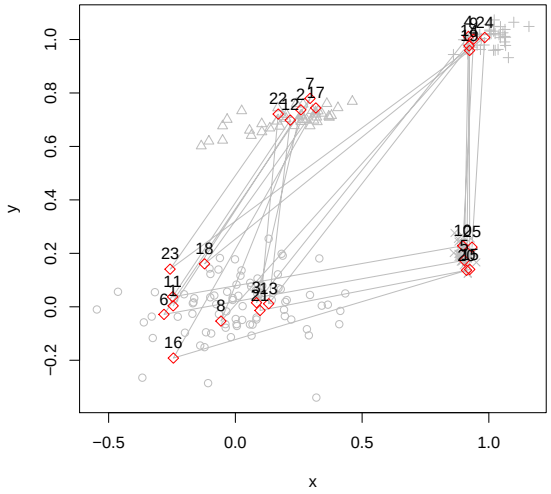


Figure 5: Simulated data set with four clusters. The points of the test data set are plotted in red and the temporal structure is depicted by sequential numbers and lines between the data points.

This results in the visualization in Figure 6(c) which shows the same EMM graph but the relative position of the clusters/states is determined by the proximity of the cluster centers. If the data space has more than two dimensions or a non-Euclidean distance measure is used, the position of the states will be determined using multidimensional scaling (MDS, **?**) to preserve the proximity information between clusters in the two dimensional representation as much as possible. Since the data set in this example is already in two-dimensional Euclidean space, the original coordinates of the centers are directly used. The size of the clusters and the width of the arrows represent again cluster and transition counts.

We can also project the points in the data set into 2-dimensional space and then add the centers of the clusters (see Figure 6(d)).

```
R> plot(emm, method = "MDS", data=EMMsim_train)
```

The simple graph representations in Figures 6(a) and (b) show a rather complicated graph for the EMM. However, Figure 6(c) with the vertices positioned to represent similarities between cluster centers shows more structure. The clusters clearly fall into four groups. The projection of the cluster centers onto the data set in Figure 6(d) shows that the four groups represent the four clusters in the data where the larger clusters are split into several 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 probabilities on the path along the new sequence. Alternatively we can use the sum of the log of probabilities.

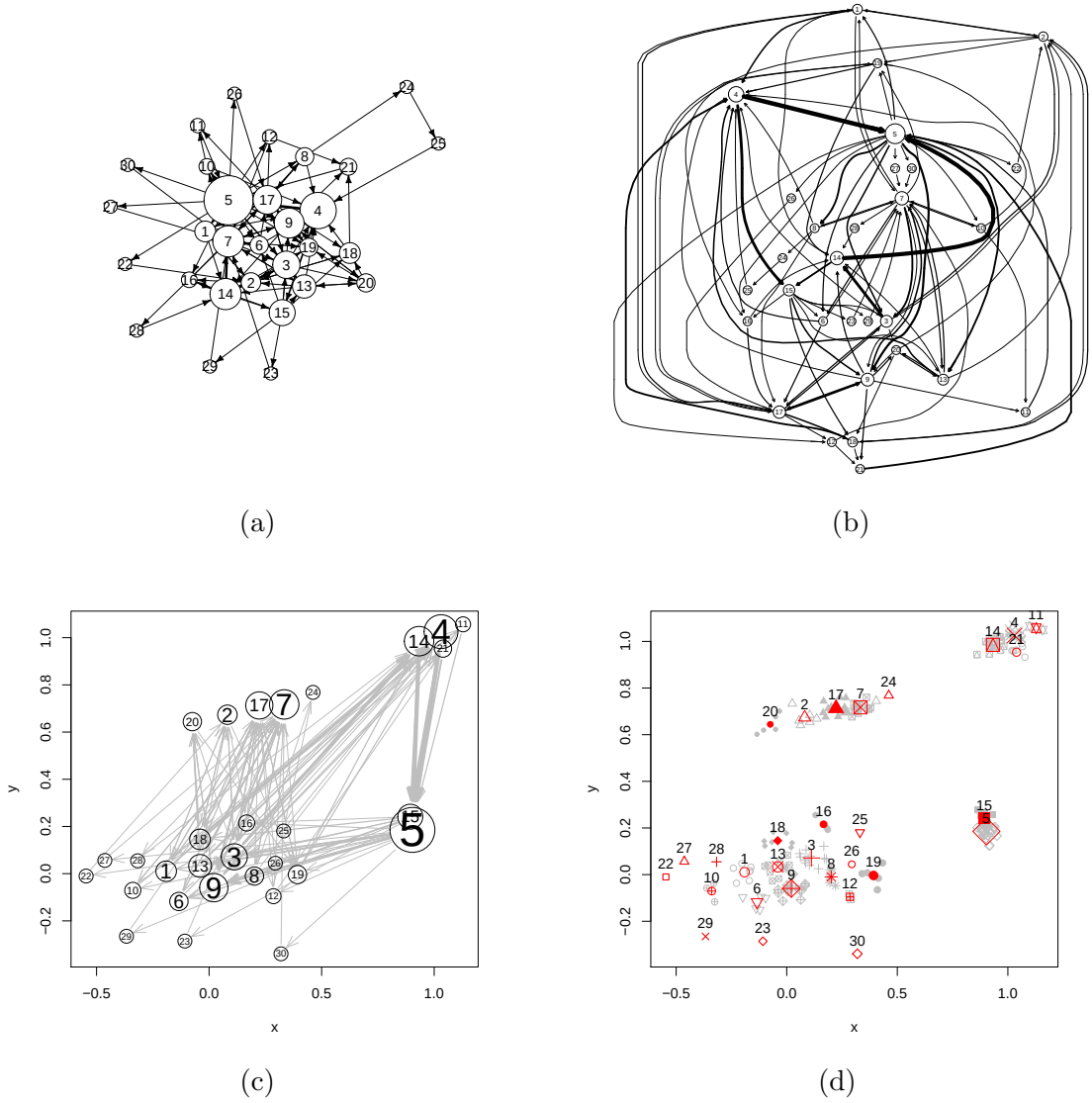


Figure 6: Visualization of the EMM for the simulated data. (a) and (b) As a simple graph. (c) A graph using vertex placement to represent dissimilarities. (d) Projection of state centers onto the simulated data.

The scores for a new sequence of length l are defined as:

$$S_{\text{product}} = \sqrt[l-1]{\prod_{i=1}^{l-1} a_{s(i),s(i+1)}} \tag{7}$$

$$S_{\text{log\_sum}} = \frac{1}{l-1} \sum_{i=1}^{l-1} \log(a_{s(i),s(i+1)}) \tag{8}$$

where $s(i)$ is the state of the i^{th} data point in the new sequence it is assigned to. Points are assigned to the closest cluster only if the distance to the center 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.

We can also use the normalized sum of probabilities as a score.

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

Since the assignment of the i^{th} data point in the new sequence to a state $s(i)$ in the model is unlikely to not be perfect we can extend the scores above with weights representing the similarity between a new data point and the state in the model.

$$S_{\text{weighted\_product}} = \sqrt[l-1]{\prod_{i=1}^{l-1} \text{simil}(x_i, s(i))\, a_{s(i),s(i+1)}} \tag{10}$$

$$S_{\text{weighted\_sum}} = \frac{1}{l-1} \sum_{i=1}^{l-1} \text{simil}(x_i, s(i))\, a_{s(i),s(i+1)} \tag{11}$$

$$S_{\text{weighted\_log\_sum}} = \frac{1}{l-1} \sum_{i=1}^{l-1} \log(\text{simil}(x_i, s(i))\, a_{s(i),s(i+1)}) \tag{12}$$

where x_i represents the i-th data point in the new sequence, $\text{simil}(x_i, s(i)) = 1/(1+\text{d}(x_i, s(i)))$ and d() is the distance function used to build the model.

A final very rough score can be obtained by just counting the number of transitions in the new sequnce which are missing (not supported) in the model.

$$S_{\text{missing\_transitions}} = \sum_{i=1}^{l-1} \text{I}(a_{s(i),s(i+1)}) \tag{13}$$

where $\text{I}(v)$ is indicator function which is 1 for $v = 0$ and 0 otherwise.

To take the initial transition probability also into account is straight forward to add the initial probability $a_{\epsilon,s(1)}$ to the equations above.

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="product", match_cluster="exact")
```

```
[1] 0
```

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

```
[1] 0.227
```

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 the problem we can look at the transition table for the test sequence. The transition table is computed by `transition_table()`.

```
R> transition_table(emm, EMMsim_test, match_cluster="exact")
```

```
   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 clusters (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 6(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. This is also shown by the "missing transition" score.

```
R> score(emm, EMMsim_test, method="missing_transitions", match_cluster="exact")

[1] 7
```

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_cluster="nn"` is the default for `score()`). Here a new data point is assigned to the closest cluster even if it falls outside the threshold. The problem with missing transitions 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`. It is called plus one since 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> methods <- c("product", "weighted_product", "log_sum",
+     "weighted_log_sum","sum", "weighted_sum")
R> sapply(methods, FUN = function(m)
+     score(emm, EMMsim_test, method=m, plus_one=TRUE))

        product weighted_product          log_sum weighted_log_sum
        0.06982          0.06631         -2.66189         -2.71344
            sum     weighted_sum
        0.09538          0.09128
```

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 7.

```
R> ## find best predicting model (clustering)
R> k <- 2:10
R> emmc <- recluster_hclust(emm, k=k, method ="average")
R> plot(attr(emmc, "cluster_info")$dendrogram)


R> sc <- sapply(emmc, score, EMMsim_test, "product")
R> names(sc) <- k
R> sc
```

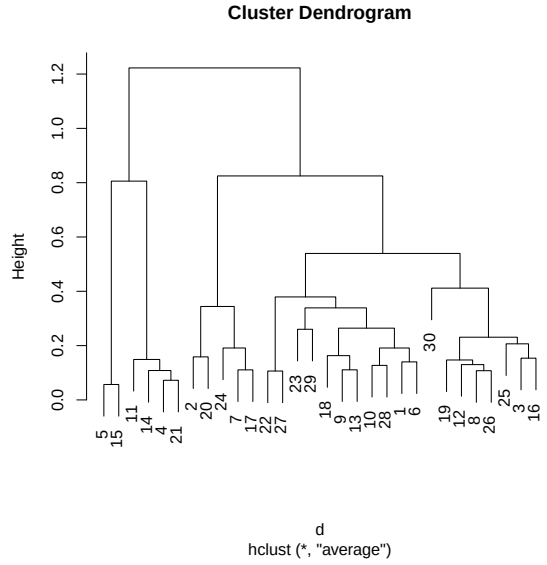


Figure 7: Dendrogram for clustering state centers of the EMM build from simulated data.

```
     2      3      4      5      6      7      8      9     10
0.5183 0.5780 0.7492 0.5925 0.5914 0.5844 0.5423 0.5059 0.4233
```

The best performing model has a score of 0.749 for a k of 4. This model is depicted in Figure 8(a). Since the four groups in the data set are well separated, reclustering finds the original structure (see Figure 8(b)) with all points assigned to the correct state.

6. Applications

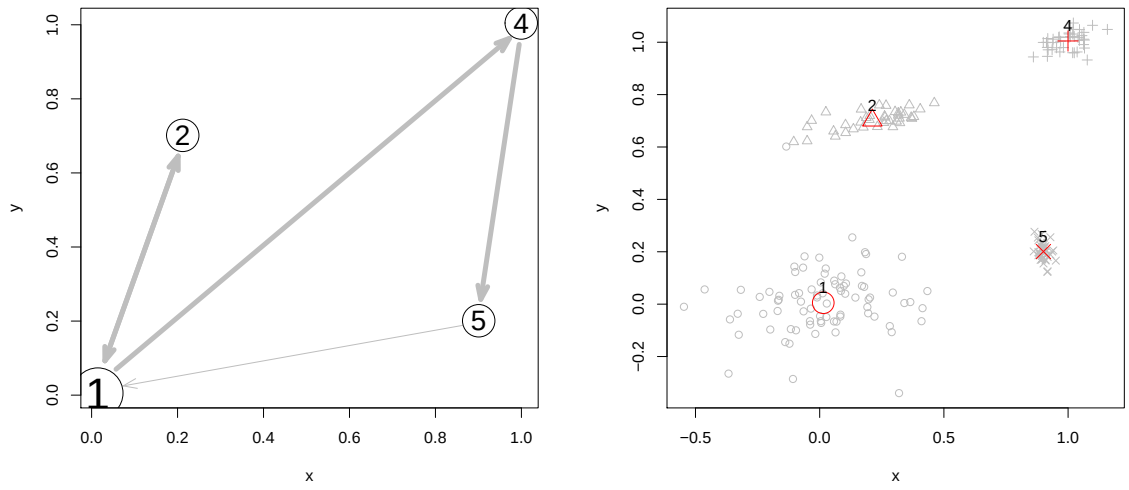
6.1. Analyzing river flow data

The **rEMM** package also contains a data set called *Derwent* which was originally used by **?**. It consists of river flow readings (measured in m^3 per second) from four catchments of 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
```



(a) (b)

Figure 8: Best performing final clustering for EMM with a k of 4.

```
Median : 10.95   Median :  7.89   Median : 2.16   Median : 3.13
Mean   : 14.33   Mean   : 10.64   Mean   : 2.67   Mean   : 4.51
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.   :0.01
1st Qu.:0.180   1st Qu.:0.04
Median :0.330   Median :0.09
Mean   :0.544   Mean   :0.14
3rd Qu.:0.640   3rd Qu.:0.16
Max.   :6.280   Max.   :4.16
NA's   :31      NA's   :252
```

From the summary we see that the gauged 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. From 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 flow",
+ main=colnames(Derwent)[1])
```

In Figure 9 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 seems

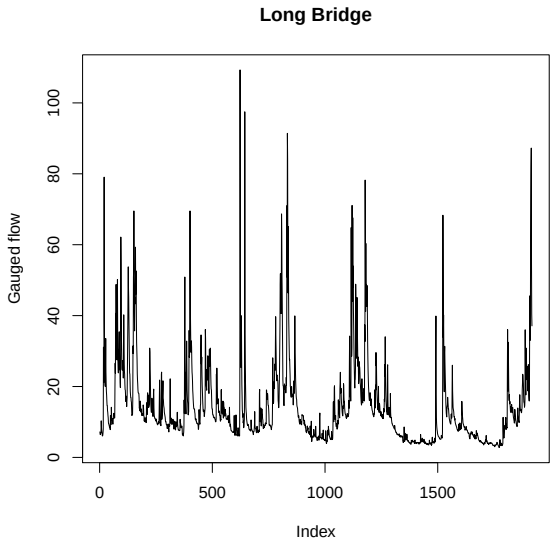


Figure 9: Gauged flow (in m^3/s) of the river Derwent at the Long Bridge catchment.

to have more variability in the summer months and the second year has an unusual event (around the index of 600 in Figure 9) 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.

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

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

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

The projection of the cluster centers into 2-dimensional space in Figure 11(a) reveals that all but clusters 11 and 12 are placed closely together.

Next we look at frequent clusters and transitions. We define rare here as all clusters/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 clusters.

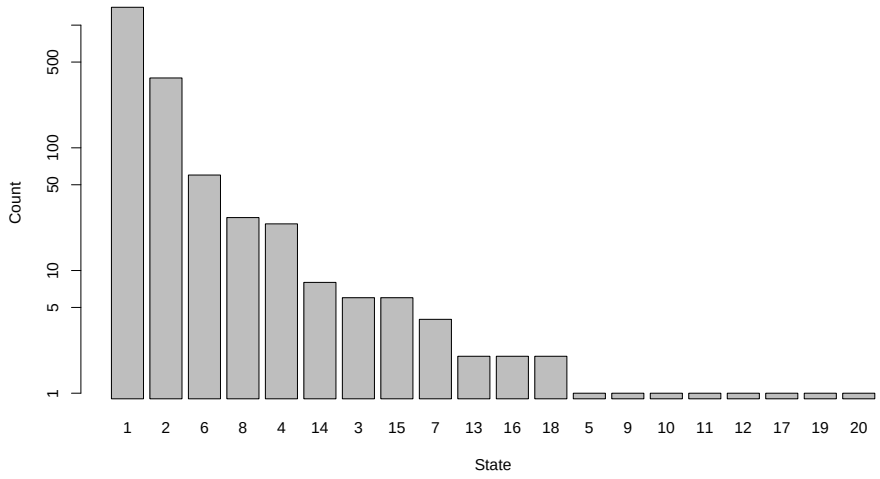
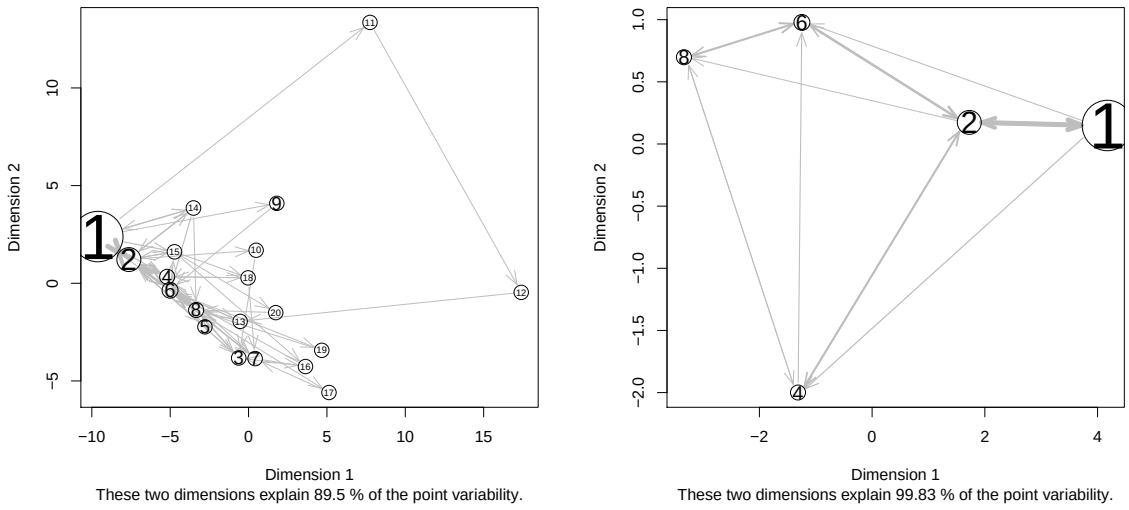


Figure 10: Distribution of state counts of the EMM for the Derwent data.



(a) (b)

Figure 11: Cluster 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)

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

[1] 9.59

R> plot(prune(emm, rare_threshold), method="MDS")
```

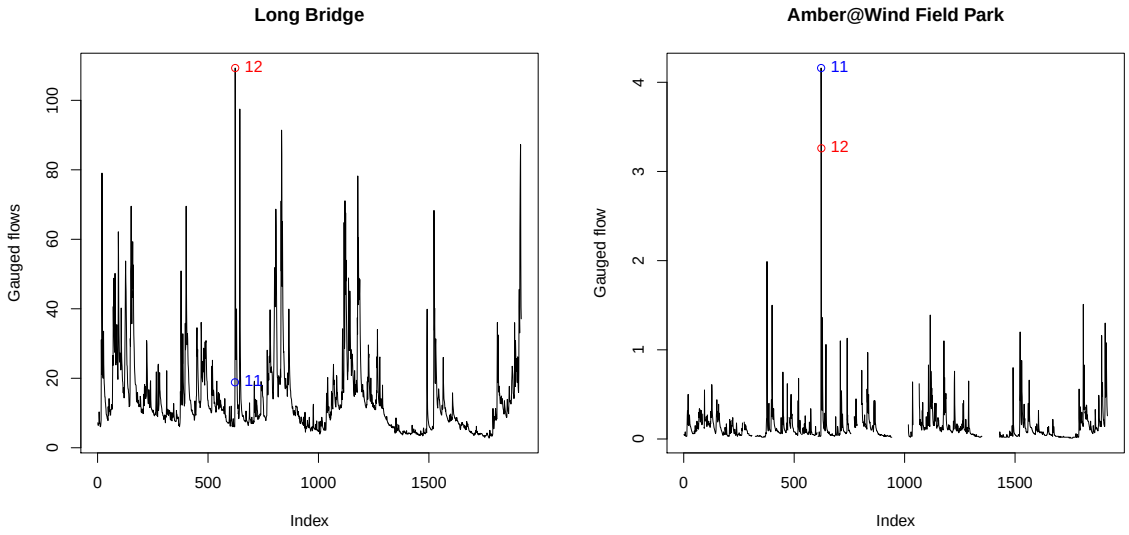
The pruned model depicted in Figure 11(b) shows that 5 clusters represent approximately 99.5% of the river's behavior. All five clusters come from the lower half of the large group of clusters in Figure 11(a). Clusters 1 and 2 are the most frequently occurring clusters and the wide bidirectional arrow connecting them means that observing transitions between these two clusters are common. To analyze the meaning of the two outlier clusters (11 and 12) identified in Figure 11(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",
+ main=colnames(Derwent)[catchment])
R> state_sequence <- find_clusters(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")
R> #mark_states("9", state_sequence, Derwent[,catchment], col="green", label="9")
R> #mark_states("3", state_sequence, Derwent[,catchment], col="blue")
```

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

```
R> catchment <- 6
R> plot(Derwent[,catchment], type="l", ylab="Gauged flow",
+ 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")
R> #mark_states("9", state_sequence, Derwent[,catchment], col="green", label="9")
R> #mark_states("3", state_sequence, Derwent[,catchment], col="blue")
```

The catchment at Wind Field Park is at the Amber river which is a tributary of the Derwent and we see in Figure 12(b) that the day before the flood occurs, the flow shoots up to $4m^3/s$ which is caught by cluster 11. The temporal structure clearly indicated that a flood is imminent the next day.



(a) (b)

Figure 12: Gauged flow (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.

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 **?**). Typically alignment heuristics like BLAST (**?**) or a Hidden Markov Model (HMM) (e.g., **?**) 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 (**?**).

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 of `NA` resulting in resetting the EMM during the build process.

? 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 **?** 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 **?**). We use an experimentally found threshold of 0.1.

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

R> plot(emm, method = "graph")
R> ## start state for sequences have an initial state probability >0
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 13. Note that each cluster/state in the EMM corresponds to one or more windows of the rRNA sequence (the size of the cluster indicates the number of windows). The initial transition probabilities show that most sequences start the first count window in cluster 1. Several interesting observations can be made from this representation.

- There exists a path through the graph using only the largest clusters 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 **?** 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 current 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.

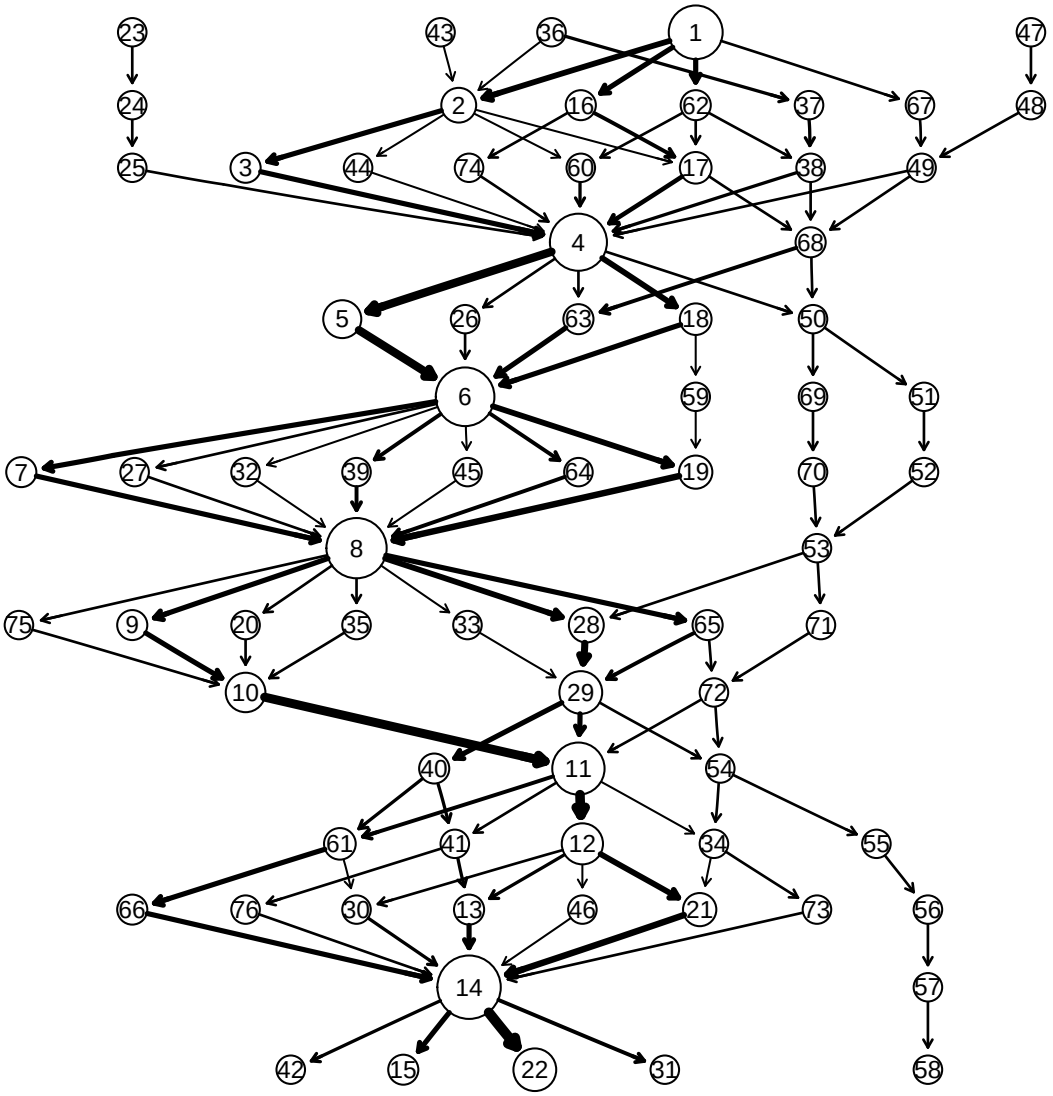


Figure 13: An EMM representing 16S sequences from the class Mollicutes represented as a graph.

Future work will include extending popular data stream clustering algorithms with EMM, incorporate higher order models and add support for reading data directly from data stream sources.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments.

Part of the research presented in this paper was supported in part by the National Science Foundation under Grant No. IIS-0948893 and the National Human Genome Research Institute under Grant No. R21HG005912.

Affiliation:

Michael Hahsler
Computer Science and Engineering
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

Margaret H. Dunham
Computer Science and Engineering
Lyle School of Engineering
Southern Methodist University
P.O. Box 750122
Dallas, TX 75275-0122
E-mail: mhd@lyle.smu.edu
URL: http://lyle.smu.edu/~mhd