Comparing Time-Series Clustering Algorithms in R Using the dtwclust Package

Alexis Sardá-Espinosa

Abstract

Clustering strategies have not changed considerably since their initial definition. Most of the improvements are either related to the distance measure used to assess dissimilarity, or the function used to calculate prototypes or centroids. Time-series clustering is no exception, with the Dynamic Time Warping distance being particularly popular in that context. This distance is computationally expensive, so many related optimizations have been developed over the years. Since no single clustering algorithm can be said to perform best on all datasets, different strategies must be tested and compared, so a common infrastructure can be advantageous. In this paper, a general overview of time-series clustering is given, including many specifics related to Dynamic Time Warping and other recently proposed techniques. At the same time, a description of the **dtwclust** package for the R statistical software is provided, showcasing how it can be used to evaluate many different time-series clustering procedures.

Keywords: time-series, clustering, R, dynamic time warping, lower bound.

1. Introduction

Cluster analysis is a general task which concerns itself with the creation of groups of objects, where each group is called a cluster. Ideally, all members of the same cluster are similar to each other, but are as dissimilar as possible from objects in a different cluster. There is no single definition of a cluster, and the characteristics of the objects to be clustered vary. Thus, there are several algorithms to perform clustering. Each one defines specific ways of defining what a cluster is, how to measure similarities, how to find groups efficiently, etc. Additionally, each application might have different goals, so a certain clustering algorithm may be preferred depending on the type of clusters sought (Kaufman and Rousseeuw 1990).

Clustering algorithms can be organized differently depending on how they handle the data and how the groups are created. When it comes to static data, i.e. if the values do not change with time (Liao 2005), clustering methods are usually divided into five major categories: **partitioning (or partitional)**, **hierarchical**, **density-based**, **grid-based** and **model-based** methods (Liao 2005; Rani and Sikka 2012). They may be used as the main algorithm, as an intermediate step, or as a preprocessing step (Aghabozorgi *et al.* 2015).

Time-series are a common type of dynamic data that naturally arises in many different scenarios, such as stock data, medical data, and machine monitoring, just to name a few (Aghabozorgi *et al.* 2015; Aggarwal and Reddy 2013). They pose some challenging issues due to the

This paper corresponds to **dtwclust** version 2.3.0.9000, 2016-10-04.

large size and high dimensionality commonly associated with time-series (Aghabozorgi *et al.* 2015). In this context, dimensionality of a series is related to time, and it can be understood as the length of the series. Additionally, a single time-series object may be constituted of several values that change on the same time scale, in which case they are identified as multivariate time-series.

There are many techniques to modify time-series in order to reduce dimensionality, and they mostly deal with the way time-series are represented. Changing representation can be an important step, not only in time-series clustering, and it constitutes a wide research area on its own (cf. Table 2 in Aghabozorgi *et al.* (2015)). While choice of representation can directly affect clustering, it can be considered as a different step, and as such it will not be discussed further in this paper.

Time-series clustering is a type of clustering algorithm made to handle dynamic data. The most important elements to consider are the **similarity or distance measure**, the **prototype extraction function** (if applicable), the **clustering algorithm itself**, and **cluster evaluation** (Aghabozorgi *et al.* 2015). In most cases, algorithms developed for time-series clustering take static clustering algorithms and either modify the similarity definition or the prototype extraction function by an appropriate one, or apply a transformation to the series so that static features are obtained (Liao 2005). Therefore, the underlying basis for the different clustering procedures remains approximately the same across clustering methods. The most common approaches are hierarchical and partitional clustering (cf. Table 4 in Aghabozorgi *et al.* (2015)), the latter of which includes fuzzy clustering.

Aghabozorgi *et al.* (2015) classify time-series clustering algorithms based on the way they treat the data and how the underlying grouping is performed. One classification depends on whether the **whole** series, a **subsequence**, or individual **time points** are to be clustered. On the other hand, the clustering itself may be **shape-based**, **feature-based** or **model-based**. Aggarwal and Reddy (2013) makes an additional distinction between online and offline approaches, where the former usually deals with grouping incoming data streams on-the-go (see e.g. Rakthanmanon *et al.* (2011)), while the latter deals with data that no longer change.

In this context, it is common to change the distance measure for the **Dynamic Time Warping** (DTW) distance (Aghabozorgi *et al.* 2015). The calculation of the DTW distance involves a dynamic programming algorithm that tries to find the optimum warping path between two series under certain constraints. However, the DTW algorithm is computationally expensive, both in time and memory utilization. Over the years, several variations and optimizations have been developed in an attempt to accelerate or optimize the calculation. Some of the most common techniques will be discussed in more detail in Section 2.1.

Due to its nature, clustering procedures lend themselves for parallelization, since a lot of similar calculations are performed independently of each other. This can make a very significant difference, especially if the data complexity increases (which can happen really quickly in case of time-series), or some of the more computationally expensive algorithms are used.

Variations in the clustering procedure could have a big impact in performance with respect to cluster quality and execution time. As such, it is desirable to have a common platform on which clustering algorithms can be tested and compared against each other. The **dtwclust** package, developed for the R statistical software (R Core Team 2016), provides such functionality, and includes implementations of recently developed time-series clustering algorithms

and optimizations. Many of the included algorithms and optimizations are tailored to the DTW distance, hence the package's name. However, the main clustering function is flexible so that one can test many different clustering approaches, using either the time-series directly, or by applying suitable transformations and then clustering in the resulting space. Instead of providing a detailed explanation of the existing algorithms, we will concern ourselves with describing which ones are available in **dtwclust**, mentioning the most important characteristics of each and showing how the package can be used to evaluate them. Additionally, the variations related to DTW and other common distances will be explored, and the parallelization strategies and optimizations will be described. Refer to Appendix A for some technical notes about the package. For a more comprehensive overview of the state of the art in time-series clustering, the reader is referred to the included references and the articles mentioned therein.

The rest of this paper is organized as follows. The relevant information to the distance measures will be presented in Section 2. Supported algorithms for prototype extraction will be discussed in Section 3. The main clustering algorithms will be described in Section 4. The parallelization strategies included will be mentioned in Section 5. Some basic information with respect to cluster evaluation will be provided in Section 6, and the final remarks will be given in Section 7. Code examples will be given in the appendices. The data used throughout this paper is included in the package (saved in a list called `CharTraj`), and is a subset of the character trajectories dataset found in Lichman (2013); they are pen tip trajectories recorded for individual characters, and the subset contains 5 examples of the X velocity for each considered character.

2. Distance measures

Calculating distances, as well as cross-distance matrices, between time-series objects is one of the cornerstones of any time-series clustering algorithm. It is a task that is repeated very often and loops across all objects applying a suitable distance function. The **proxy** package (Meyer and Buchta 2016), also developed for R, provides a highly optimized and extensible framework for these calculations, and is used extensively by **dtwclust**. It includes several common metrics, which are saved in a database object called `pr_DB`. Additionally, users can register custom functions in the database. This has the advantage that all registered functions can be used with the `proxy::dist` function, which results in a high level of consistency. Refer to Appendix B for an example. Unless otherwise noted, all the distances discussed here are registered with **proxy** when **dtwclust** is attached.

It is important to note that the `proxy::dist` function parses all matrix-like objects **row-wise**, meaning that, in the context of time-series clustering, it would consider the rows of a *matrix* or *data frame* as the individual time-series. Matrices and data frames cannot contain series with different length, something that can be circumvented by encapsulating the series in a *list*, each element of the list being a single series. Internally, **dtwclust** coerces all provided data to a list, and as of version 2.3.0, it parses both matrices and data frames row-wise. Moreover, lists enable support for multivariate series, where a single multivariate time-series should be provided as a matrix where time spans the rows and the variables span the columns. Thus, several multivariate time-series should be provided as a *list of matrices* to ensure that they are correctly detected. Note, however, that not all distance and centroid functions support multivariate time-series.

The l_1 and l_2 vector norms, also known as Manhattan and Euclidean distances respectively, are the most commonly used distance measures, and are arguably the only competitive distances when measuring dissimilarity (Aggarwal *et al.* 2001; Lemire 2009). They can be efficiently computed, but are only defined for series of equal length and are sensitive to noise, scale and time shifts.

To facilitate notation in the following sections, we define a time-series as a vector (or set of vectors in case of multivariate series) x. Each vector has length n. In general, x_i^v represents the i-th element of the v-th variable of the (possibly multivariate) time-series x. We will assume that all elements are equally spaced in time in order to avoid the time index explicitly.

2.1. Dynamic time warping distance

At the core, DTW is a dynamic programming algorithm that compares two series and tries to find the optimum warping curve between them under certain constraints, such as monotonicity. It started being used by the data mining community to overcome some of the limitations associated with the Euclidean distance (Ratanamahatana and Keogh 2004; Berndt and Clifford 1994). In **dtwclust**, all DTW calculations are performed by the **dtw** package (Giorgino 2009), which comprehensively aggregates most of the related variations and optimizations.

The easiest way to get an intuition of what DTW does is graphically. Figure 1 shows the alignment between two sample time-series x and y. In this instance, the initial and final points of the series must match, but other points may be warped in time in order to find better matches.

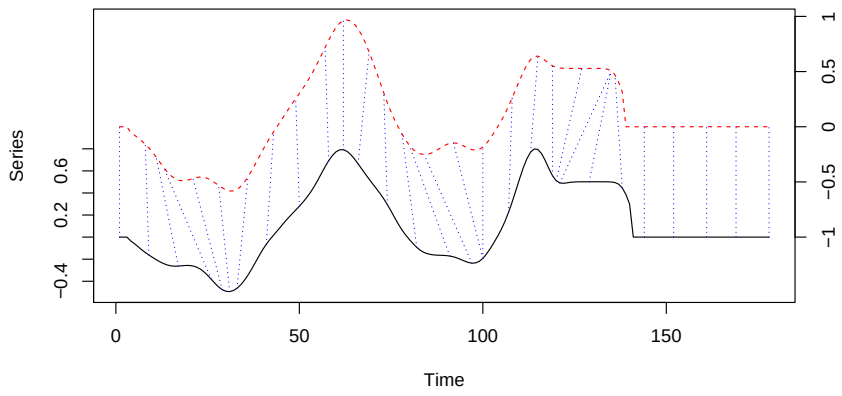


Figure 1: Sample alignment performed by the DTW algorithm between two series. The dashed blue lines exemplify how some points are mapped to each other, which shows how they can be warped in time. Note that the vertical position of each series was artificially altered for visualization.

DTW is computationally expensive. If x has length n and y has length m, the DTW distance between them can be computed in $O(nm)$ time, which is almost quadratic if m and n are similar. Additionally, DTW is prone to implementation bias since its calculations are not easily vectorized and tend to be very slow in non-compiled programming languages. The **dtw** package includes a C implementation of the dynamic programming step of the algorithm,

which should be very fast; its level of generality may sacrifice some performance, but in most situations it will be negligible. Optionally, a basic custom implementation of the DTW algorithm is included with **dtwclust** in the `dtw_basic` function, which has less functionality but still supports the most common options, and it should be faster. DTW is considered a more robust distance measure for time-series (Wang *et al.* 2013), and can potentially deal with series of different length directly. This is not necessarily an advantage, as it has been shown before that performing linear reinterpolation to obtain equal length may be appropriate if m and n do not vary significantly (Ratanamahatana and Keogh 2004). For a more detailed explanation of the DTW algorithm, see e.g. Giorgino (2009). However, there are some aspects that are worth discussing here.

The first step in DTW involves creating a **local cost matrix** (LCM or lcm), which has $n \times m$ dimensions. Such a matrix must be created for every pair of distances compared, meaning that memory requirements may grow quickly as the dataset size grows. Considering x and y as the input series, for each element (i, j) of the LCM, the l_p norm between x_i and y_j must be computed. This is defined in Eq. (1), explicitly denoting that multivariate series are supported as long as they have the same number of variables (note that for univariate series, the LCM will be identical regardless of the norm used). Thus, it makes sense to speak of a DTW_p distance, where p corresponds to the l_p norm that was used to construct the LCM. However, this norm also plays an important role in the next step of DTW.

$$lcm(i, j) = \left(\sum_v |x_i^v - y_j^v|^p \right)^{1/p} \tag{1}$$

In the seconds step, the DTW algorithm finds the path that minimizes the alignment between x and y by iteratively stepping through the LCM, starting at $lcm(1, 1)$ and finishing at $lcm(n, m)$, and aggregating the cost. At each step, the algorithm finds the direction in which the cost increases the least under the chosen constraints; see Fig. 2 for a visual representation of the path corresponding to Fig. 1. If we define $\phi = \{(1, 1), \ldots, (n, m)\}$ as the set containing all the points that fall on the optimum path, then the final distance would be computed with Eq. (2), where m_ϕ is a per-step weighting coefficient and M_ϕ is the corresponding normalization constant (Giorgino 2009).

$$\text{DTW}_p(x, y) = \left(\sum \frac{m_\phi \, lcm(k)^p}{M_\phi} \right)^{1/p}, \ \forall k \in \phi \tag{2}$$

It is clear that the choice of l_p norm comes into play twice during the DTW the algorithm, but why is this crucial? The **dtw** package also makes internal use of **proxy** to calculate the LCM, and it allows changing the norm by means of its `dist.method` argument. However, as previously mentioned, the norm only affects the LCM if multivariate series are used. By default, **dtw** does **not** consider the l_p norm in Eq. (2) during step two of the algorithm, regardless of what is provided in `dist.method`. For this reason, a special version of DTW_2 is registered with **proxy** by **dtwclust** (called simply `"DTW2"`), which also uses **dtw** for the core calculations, but appropriately uses the l_2 norm in the second step.

The way in which the algorithm traverses through the LCM is primarily dictated by the chosen *step pattern*. The step pattern is a local constraint that determines which directions are allowed when moving ahead in the LCM as the cost is being aggregated, as well as the

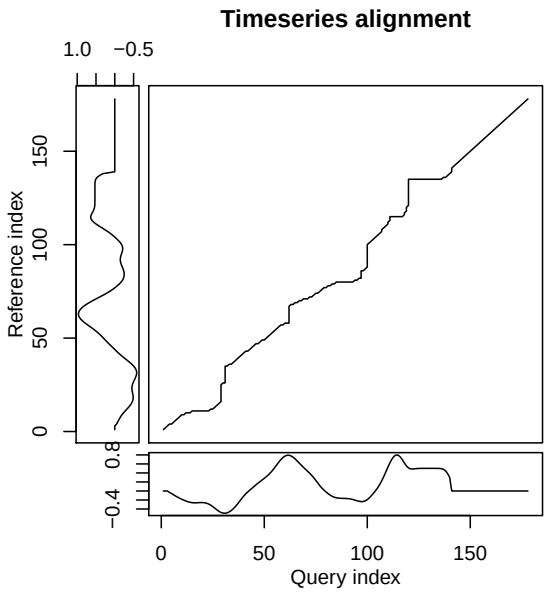


Figure 2: Visual representation of the optimum path found. The big square in the center represents the LCM created for these specific series.

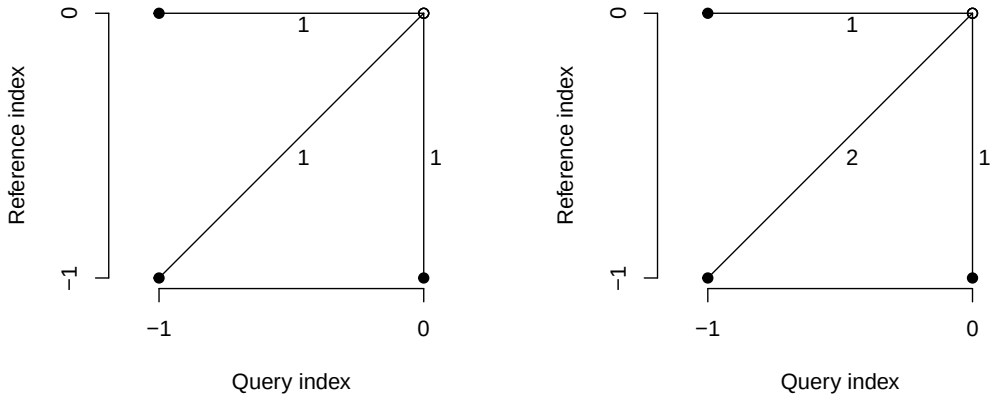
associated per-step weights. Figure 3 depicts two common step patterns and their names in the **dtw** package. Unfortunately, very few articles from the data mining community specify which pattern they use, although in the author's experience, the `symmetric1` pattern seems to be standard. By contrast, the `dtw` function uses the `symmetric2` pattern by default, but it is simple to modify this by providing the appropriate value in the `step.pattern` argument. The choice of step pattern also determines whether the corresponding DTW distance can be normalized or not (which may be important for series with different length). See Giorgino (2009) for a complete list of step patterns and to know which ones can be normalized.

As final comments, it should be noted that the DTW distance does **not** satisfy the *triangle inequality*, and it is **not** symmetric in general, e.g. for asymmetric step patterns (Giorgino 2009). Currently, out of the included distances in **dtwclust**, only those based on DTW support multivariate series.

Global DTW constraints

One of the possible modifications of DTW is to use global constraints, also known as window constraints. Such constraints limit the area of the LCM that can be reached by the algorithm. There are many types of windows (see e.g. Giorgino (2009)), but one of the most common ones is the Sakoe-Chiba window (Sakoe and Chiba 1978), with which an allowed region is created along the diagonal of the LCM (see Fig. 4). These constraints can marginally speed up the DTW calculation, but they are mainly used to avoid pathological warping. It is common to use a window whose size 10% of the series' length, although sometimes smaller windows produce even better results (Ratanamahatana and Keogh 2004).

Strictly speaking, if the series being compared have different length, a constrained path may



(a) `symmetric1` step pattern (b) `symmetric2` step pattern

Figure 3: Two common step patterns used by DTW when traversing the LCM. At each step, the lines denote the allowed directions that can be taken, as well as the weight associated with each one.

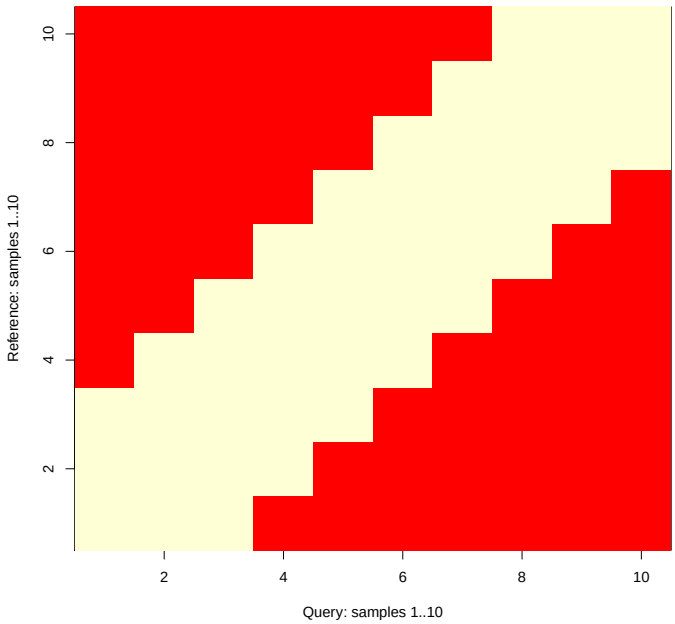


Figure 4: Visual representation of the Sakoe-Chiba constraint for DTW. The red elements will not be considered by the algorithm when traversing the LCM.

not exist, since the Sakoe-Chiba band may prevent the end point of the LCM to be reached (Giorgino 2009). In these cases, a *slanted band window* may be preferred, since it stays along the diagonal for series with different length and is equivalent to the Sakoe-Chiba window for series with equal length. If a window constraint is used with **dtwclust**, a slanted band is employed.

It is not possible to know *a priori* what window size, if any, will be best for a specific application, although it is usually agreed that no constraint is a poor choice. For this reason, it is better to perform tests with the data one wants to work with, perhaps taking a subset to avoid excessive running times.

It should be noted that, when reported, window sizes are always integers greater than zero. If we denote this number with w, and for the specific case of the slanted band window, the valid region of the LCM will be constituted by all valid points in the range $[(i, j-w), (i, j+w)]$ for all (i, j) along the LCM diagonal. Thus, $2w+1$ elements will fall within the window for a given window size w. This is the convention followed by **dtwclust**.

Lower bounds for DTW

Due to the fact that DTW itself is expensive to compute, *lower bounds* (LBs) for the DTW distance have been developed. These lower bounds guarantee being less than or equal to the corresponding DTW distance. They have been exploited when indexing time-series databases, classification of time-series, clustering, etc. (Keogh and Ratanamahatana 2005; Begum *et al.* 2015). Out of the existing DTW LBs, the two most effective are termed `LB_Keogh` (Keogh and Ratanamahatana 2005) and `LB_Improved` (Lemire 2009). The reader is referred to the respective articles for a detailed definition and proof of the LBs, however some important considerations will be further discussed here.

Each LB can be computed with a specific l_p norm. Therefore, it follows that the l_p norms used for DTW and LB calculations must match, such that $\text{LB}_p \leq \text{DTW}_p$. Moreover, $\text{LB\_Keogh}_p \leq \text{LB\_Improved}_p \leq \text{DTW}_p$, meaning that `LB_Improved` can provide a tighter LB. It must be noted that the LBs are **only** defined for series of equal length, and similar to DTW, are **not** symmetric regardless of the l_p norm used to compute them. Also note that the choice of step pattern affects the value of the DTW distance, changing the tightness of a given LB.

One crucial step when calculating the LBs is the computation of the so-called *envelops*. These envelops **require** a window constraint, and thus are dependent on both the type and size of the window. Based on these, a *running* minimum and maximum are computed and, respectively, a lower and upper envelop are generated. Figure 5 depicts a sample time-series with its corresponding envelops for a Sakoe-Chiba window of size 15.

When calculating the LBs between x and y, a copy of the *reference* series is created; denote this copy with H. Then the envelops for the other series (the *query*) are computed: if y is the query, then its upper and lower envelop (UE and LE respectively) are compared against x, so that the values of H are updated according to Eq. (3). Subsequently, `LB_Keogh` is defined as in Eq. (4), where $\|\cdot\|_p$ is the l_p norm of the series. The improvement made by `LB_Improved` consists in calculating a second vector H' by using H as query and y as reference, effectively defining the LB as in Eq. (5) (cf. Fig. 5 in Lemire (2009)).

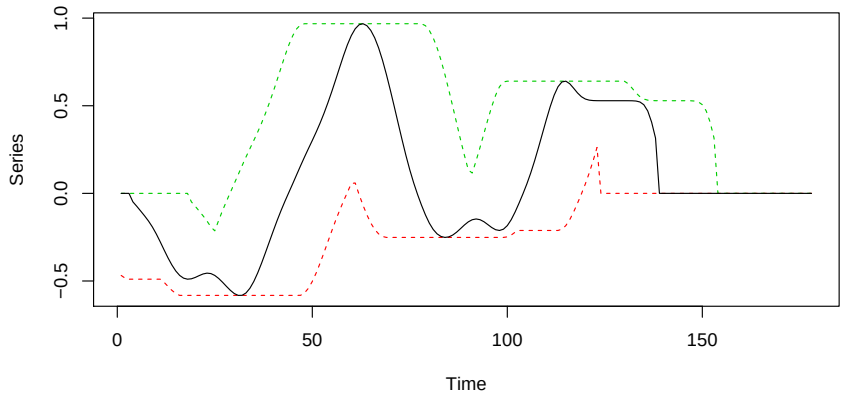


Figure 5: Visual representation of a time-series (shown as a solid black line) and its corresponding envelops based on the Sakoe-Chiba window. The green dashed line represents the upper envelop, while the red dashed line represents the lower envelop.

$$H_i = UE_i, \quad \forall x_i > UE_i \tag{3a}$$
$$H_j = LE_j \quad \forall x_j < LE_j \tag{3b}$$

$$\text{LB\_Keogh}_p(x, y) = \|x - H\|_p \tag{4}$$

$$\text{LB\_Improved}_p(x, y) = \sqrt[p]{\text{LB\_Keogh}_p(x, y)^p + \text{LB\_Keogh}_p(y, H)^p} \tag{5}$$

In order for the LBs to be worth it, they must be computed in significantly less time than it takes to calculate the DTW distance. Lemire (2009) developed a streaming algorithm to calculate the envelops using no more than $3n$ comparisons when using a Sakoe-Chiba window. This algorithm is implemented in **dtwclust** using the C++ programming language, ensuring an efficient calculation.

`LB_Keogh` requires the calculation of one set of envelops for every pair of series compared, whereas `LB_Improved` must calculate two sets of envelops for every pair of series. If the LBs must be calculated between several time-series, some envelops can be reused when a given series is compared against many others. This optimization is included in the LB functions registered with **proxy** by **dtwclust**.

The function `dtw_lb` included in **dtwclust** (and the corresponding version registered with **proxy**) leverages `LB_Improved` in order to find nearest neighbors faster. It first computes a distance matrix using only `LB_Improved`. Then, it looks for the nearest neighbor by taking the argument of the row-wise minima of the resulting matrix. Using this information, it updates the distance values of the nearest neighbors using the DTW distance. It continues this process iteratively until no changes in the nearest neighbors are detected. See Appendix C for an example.

Because of the way it works, `dtw_lb` may only be useful if one is only interested in nearest neighbors, which is usually the case in **partitional** clustering. However, if partition around medoids is performed (see Section 3.2), the distance matrix should **not** be precomputed so that the matrices are correctly updated on each iteration.

2.2. Shape-based distance

The shape-based distance (SBD) was recently proposed as part of the k-Shape clustering algorithm (Paparrizos and Gravano 2015); this algorithm will be further discussed in Section 3.4 and Section 4.2.2. SBD is based on the *cross-correlation* with coefficient normalization (NCCc) sequence between two series, and it is thus sensitive to scale, which is why Paparrizos and Gravano (2015) recommend z-normalization. The distance can be calculated with the formula shown in Eq. (6), where $\|\cdot\|_2$ is the l_2 norm of the series. Its range lies between 0 and 2, with 0 indicating perfect similarity.

$$SBD(x, y) = 1 - \frac{\max\left(NCCc(x, y)\right)}{\|x\|_2 \|y\|_2} \tag{6}$$

This distance can be efficiently computed by utilizing the Fast Fourier Transform (FFT) to obtain the NCCc sequence, although that might make it sensitive to numerical precision, which can produce discrepancies between systems with 32 and 64-bit architectures. Nevertheless, it can be very fast, it is symmetric, it was very competitive in the experiments performed in Paparrizos and Gravano (2015), and it supports (univariate) series of different length directly. Additionally, some FFTs can be reused when computing the SBD between several series. This optimization is also included in the SBD function registered with **proxy**.

3. Time-series prototypes

A very important element of time-series clustering has to do with the calculation of so-called time-series *prototypes*. It is expected that all series within a cluster are similar to each other, and one may be interested in trying to define a time-series that effectively summarizes the most important characteristics of all series in a given cluster. This series is sometimes referred to as an *average* series, and prototyping is also sometimes called time-series averaging, however we will prefer the term "prototyping" in this paper. The prototypes could be used for visualization aid or to perform time-series classification, but in the context of clustering, partitional procedures rely heavily on the prototyping function, since the resulting prototypes are used as cluster *centroids*; more information will be provided in Section 4.

The choice of prototyping function is closely related to the chosen distance measure, and in a similar fashion, it is not simple to know which kind of prototype will be better *a priori*. There are several strategies available for time-series prototyping, although due to their high dimensionality, it is debatable what exactly constitutes an average time-series, and some notions could worsen performance significantly. The following sections will briefly describe some of the common approaches, which are the ones included by default in **dtwclust**. Nevertheless, it is also possible to create custom prototyping functions and provide them in the `centroid` argument.

3.1. Mean and median

The arithmetic mean is used very often in conjunction with the Euclidean distance, and in many applications, this combination is very competitive, even with multivariate data. However, because of the structure of time-series, the mean is arguably a poor prototyping choice, and could even perturb convergence of a clustering algorithm (Petitjean *et al.* 2011).

Mathematically, the mean simply takes the average of each time-point i across all variables of the considered time-series. For a cluster C of size N, the (possibly multivariate) time-series mean μ can be calculated with Eq. (7), where $x_{c,i}^{v}$ is the i-th element of the v-th variable from the c-th series that belongs to cluster C.

$$\mu_i^v = \frac{1}{N} \sum_c x_{c,i}^v, \ \forall c \in C \tag{7}$$

Following this idea, it is also possible to use the median value across series in C instead of the mean, although we are not aware if this has been used in the existing literature. Also note that this prototype is limited to series with equal length and equal amount of variables.

3.2. Partition around medoids

Another very common approach is to use partition around medoids (PAM). A *medoid* is simply a representative object from a cluster, in this case also a time-series, whose average distance to all other objects in the same cluster is minimal. Since the medoid object is always an element of the original data, PAM is sometimes preferred over mean or median so that the time-series structure is not altered.

Another possible advantage of PAM is that, since the data does not change, it is possible to *precompute* the whole distance matrix once and re-use it on each iteration, and even across different number of clusters and random repetitions. While this precomputation is not necessary, it usually saves time overall, so it is done by default by **dtwclust**.

3.3. DTW barycenter averaging

The DTW distance is used very often when working with time-series, and thus a prototyping function based on DTW has also been developed in Petitjean *et al.* (2011). The procedure is called DTW barycenter averaging (DBA), and is an iterative, *global* method. The latter means that the order in which the series enter the prototyping function does **not** affect the outcome.

DBA requires a series to be used as reference (centroid), and it usually begins by **randomly** selecting one of the series in the data. In each iteration, the DTW alignment[1] between each series in the cluster C and the centroid is computed. Because of the warping performed in DTW, it can be that several time-points from a given time-series map to a single time-point in the centroid series, so for each time-point in the centroid, all the corresponding values from all series in C are grouped together according to the DTW alignment, and the mean is computed for each centroid point using the values contained in each group. This is then

[1]Internally, the DTW alignment can be computed by the **dtw** package or with the included `dtw_basic` function.

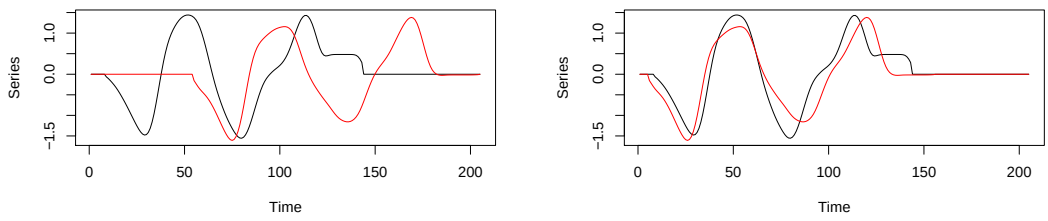
iteratively repeated until a certain number of iterations are reached, or until convergence is assumed.

One way in which DBA can be optimized is in its usage of random access memory (RAM). Since the series used on each iteration do not change, the LCMs generated by DTW always have the same size, so the required memory can be allocated once and re-used on each iteration. In order to ensure that this is the case, **dtwclust** updates the LCMs when using DBA by means of C code.

DBA is much more computationally expensive due to all the DTW calculations that must be performed. However, it is very competitive when using the DTW distance, and thanks to DTW itself, it can support series with different length directly, with the caveat that the length of the resulting prototype will be the same as the length of the reference series that was initially chosen by the algorithm.

3.4. Shape extraction

A recently proposed method to calculated time-series prototypes is termed *shape extraction*, and is part of the k-Shape algorithm (see Section 4.2.2) described in Paparrizos and Gravano (2015). As with the corresponding SBD (see Section 2.2), the algorithm depends on NCCc, and it first uses it to match two series as well as possible. Figure 6 depicts the alignment that is performed using two sample series.



(a) Series before alignment

(b) Series after alignment

Figure 6: Visualization of the NCCc-based alignment performed on two sample series. After alignment, the second (red) series is either truncated and/or prepended/appended with zeros so that its length matches the first(black) series.

As with DBA, a centroid series is needed, so one is usually **randomly** chosen from the data. The alignment can be done between series with different length, and since one of the series is *shifted* in time, it may be necessary to truncate and prepend or append zeros to the non-reference series, so that the final length matches that of the reference. This is because the final step of the algorithm builds a matrix with the matched series and performs a so-called maximization of Rayleigh Quotient to obtain the final prototype; see Paparrizos and Gravano (2015) for more details.

The output series of the algorithm **must** be z-normalized. Thus, the input series as well as the reference series **must** also have this normalization. Even though the alignment can be done between series with different length, it has the same caveat as DBA, namely that the length of the resulting prototype will depend on the length of the chosen reference. Technically, for multivariate series, the shape extraction algorithm could be applied for each variable v of all

involved series, but this was **not** explored by the authors of k-Shape.

3.5. Fuzzy-based prototypes

Even though fuzzy clustering will not be discussed until Section 4.3, it is worth mentioning here that its most common implementation uses its own centroid function, which works like a *weighted average*. Therefore, it will only work with data of equal dimension, and it may not be suitable for use directly with raw time-series data.

4. Time-series clustering algorithms

4.1. Hierarchical clustering

Hierarchical clustering, as its name suggests, is an algorithm that tries to create a hierarchy of groups in which, as the level in the hierarchy increases, clusters are created by merging the clusters from the next lower level, such that an ordered sequence of groupings is obtained (Hastie *et al.* 2009). In order to decide how the merging is performed, a similarity measure between *groups* should be specified, in addition to the one that is used to calculate pairwise similarities. However, a specific number of clusters does not need to be specified for the hierarchy to be created, and the procedure is deterministic, so it will always give the same result for a chosen set of similarity measures.

Algorithms for hierarchical clustering can be **agglomerative** or **divisive**, with the former being much more common than the latter (Hastie *et al.* 2009). In agglomerative procedures, every member of the data starts in its own cluster, and members are grouped together sequentially based on the similarity measure until all members are contained in a single cluster. Divisive procedures do the exact opposite, starting with all data in one cluster and dividing it until each member is in a *singleton*. Both strategies suffer from a lack of flexibility, because they can't perform adjustments once a split or merger has been done.

The intergroup dissimilarity is also known as *linkage*. As an example, single linkage takes the intergroup dissimilarity to be that of the closest (least dissimilar) pair (Hastie *et al.* 2009). There are many linkage methods available, although if the data can be "easily" grouped, they should all provide similar results. In **dtwclust**, the native R function `hclust` is used, and all its linkage methods are supported.

The created hierarchy can be visualized as a binary tree where the height of each node is proportional to the value of the intergroup dissimilarity between its two daughter nodes (Hastie *et al.* 2009). Such a plot is called a **dendrogram**, and an example can be seen in Fig. 7. These plots can be a useful way of summarizing the whole data in an interpretable way, although it may be deceptive, as the algorithms impose the hierarchical structure even if such structure is not inherent to the data (Hastie *et al.* 2009).

The dendrogram does **not** directly imply a certain number of clusters, but one can be induced. One option is to visually evaluate the dendrogram in order to assess the height at which the largest change in dissimilarity occurs, consequently *cutting* the dendrogram at said height and extracting the clusters that are created. Another option is to specify the number of clusters that are desired, and cut the dendrogram in such a way that the chosen number is obtained. In the latter case, several cuts can be made, and validity indices can be used to decide which

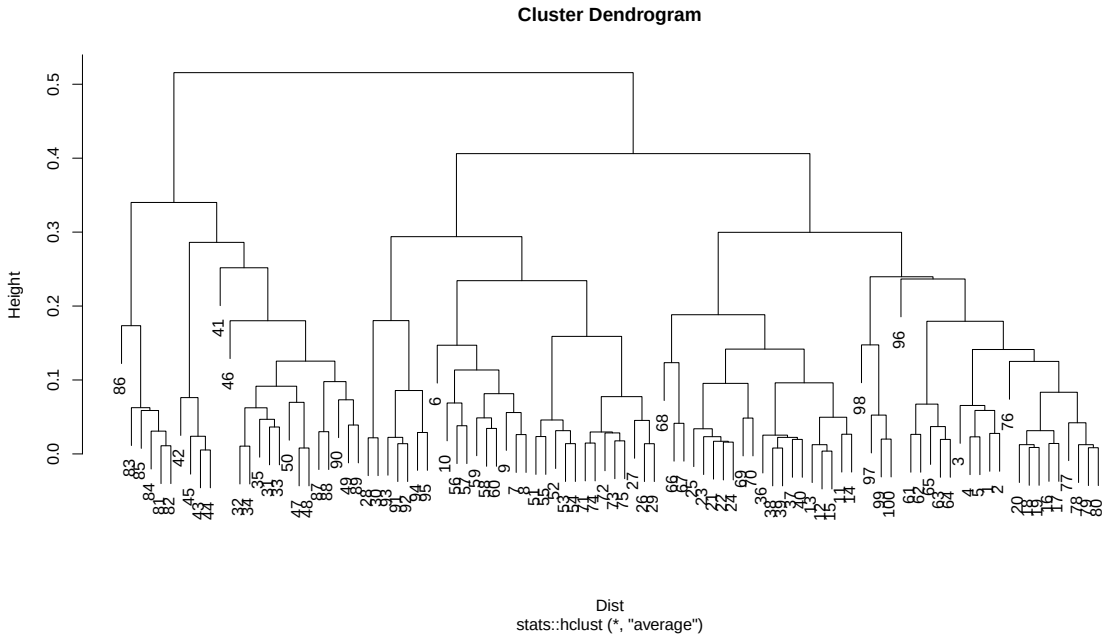


Figure 7: Sample dendrogram created by using SBD and average linkage.

value yields better performance (see Section 6).

Once the clusters have been obtained, it may be desirable to use them to create prototypes (see Section 3). At this point, any suitable prototyping function may be used, but we want to emphasize that this is not a mandatory step in hierarchical clustering. A complete example is provided in Appendix D.

Hierarchical clustering has the disadvantage that the whole distance matrix must be calculated for a given dataset, which in most cases has a time and memory complexity of $O(N^2)$ if N is the total number of objects in the dataset. Thus, hierarchical procedures are usually used with relatively small datasets.

4.2. Partitional clustering

Partitional clustering is a different strategy used to create partitions. In this case, the data is explicitly assigned to one and only one cluster out of k total clusters. The total number desired clusters **must** be specified beforehand, which can be a limiting factor, although this can be ameliorated by using validity indices (see Section 6).

Partitional procedures can be stated as combinatorial optimization problems that minimize the intracluster distance while maximizing the intercluster distance. However, finding a *global* optimum would require enumerating all possible groupings, something which is infeasible even for relatively small datasets (Hastie *et al.* 2009). Therefore, iterative greedy descent strategies are used instead, which examine a small fraction of the search space until convergence, but could converge to *local* optima.

Partitional clustering algorithms commonly work in the following way. First, k centroids are **randomly** initialized, usually by choosing k objects from the dataset at random; these are assigned to individual clusters. The distance between all objects in the data and all centroids

is calculated, and each object is assigned to the cluster of its closest centroid. A prototyping function is applied to each cluster to update the corresponding centroid. Then, distances and centroids are updated iteratively until a certain number of iterations have elapsed, or no object changes clusters anymore. It can happen that a cluster becomes empty at a certain iteration, in which case a new cluster is reinitialized at random by choosing a new object from the data (at least this is what **dtwclust** does). This tries to maintain the desired amount of clusters, but can result in instability or divergence in some cases. Perhaps using a different distance function or a lower value of k can help in those situations.

An optimization that the included centroid functions in **dtwclust** have is that, on each iteration, the centroid function is *only* applied to those clusters that changed either by losing or gaining data objects. Additionally, when PAM centroids are used, the function simply keeps track of the data indices, how they change and which ones are chosen as cluster centroids, so that no data is modified.

Some of the most popular partitional algorithms are k-means and k-medoids (Hastie *et al.* 2009). These usually use the Euclidean distance and, respectively, mean or PAM centroids (see Section 3). Most of the proposed algorithms for time-series clustering use the same basic strategy while changing the distance and/or centroid function.

Partitional clustering procedures are **stochastic** due to their random start. Thus, it is common practice to test different random starts to evaluate several local optima and choose the best result out of all the *repetitions*. They tend to produce *spherical* clusters, but have a lower complexity, so they can be applied to very large datasets. An example is given in Appendix E.

TADPole clustering

TADPole clustering was proposed in Begum *et al.* (2015). It adopts a relatively new clustering framework and adapts it to time-series clustering with the DTW distance. Because of the way the algorithm works, it can be considered a kind of PAM clustering, since the cluster centroids are always elements of the data. However, this algorithm is deterministic, depending on the value of a cutoff distance (d_c).

The algorithm first uses the DTW distance's upper (Euclidean distance) and lower (`LB_Keogh`) bounds to find series with many close neighbors (in DTW space). Anything below d_c is considered a neighbor. Aided with this information, the algorithm then tries to prune as many DTW calculations as possible in order to accelerate the clustering procedure. The series that lie in dense areas (i.e. that have lots of neighbors) are taken as cluster centroids. For a more detailed explanation of each step in the algorithm, please refer to Begum *et al.* (2015).

The algorithm relies on the DTW bounds, which are only defined for time-series of equal length. Consequently, it requires a Sakoe-Chiba constraint. Furthermore, it should be noted that the Euclidean distance is **only** valid as a DTW upper bound if the `symmetric1` step pattern is used (see Fig. 3).

k-Shape clustering

The k-Shape clustering algorithm was developed by Paparrizos and Gravano (2015). It is a partitional clustering algorithm with a custom distance measure (SBD; see Section 2.2), as well as a custom centroid function (shape extraction; see Section 3.4). It is also stochastic in

nature, and requires z-normalization in its default definition.

4.3. Fuzzy clustering

The previous procedures result in what is known as a **crisp** or **hard** partition, although hierarchical clustering only indirectly when the dendrogram is cut. In crisp partitions, each member of the data belongs to only one cluster, and clusters are mutually exclusive. By contrast, fuzzy clustering creates a **fuzzy** or **soft** partition in which each member belongs to each cluster to a certain degree. For each member of the data, the degree of belongingness is constrained so that its sum equals 1. Therefore, if there are N objects in the data and k clusters are desired, an $N \times k$ membership matrix u can be created, where all the rows must sum to 1.

One of the most widely used versions of the algorithm is **fuzzy c-means**, which is described in Bezdek (1981). It tries to create a fuzzy partition by minimizing the function in Eq. (8a) under the constraints given in Eq. (8b). The membership matrix u is initialized randomly in such a way that the constraints are fulfilled. The exponent m is known as the **fuzziness exponent** and should be greater than 1, with a common default value of 2. Originally, the distance $d_{p,c}$ was defined as the Euclidean distance between the p-th object and the c-th fuzzy centroid, so that the objective was written in terms of the squared Euclidean distance. However, the definition of this distance can change (see e.g. D'Urso and Maharaj (2009)).

$$\min \sum_{p=1}^{N} \sum_{c=1}^{k} u_{p,c}^{m} d_{p,c}^{2} \tag{8a}$$

$$\sum_{c=1}^{k} u_{p,c} = 1, \quad u_{p,c} \geq 0 \tag{8b}$$

The centroid function used by fuzzy c-means calculates the mean for each point across all members in the data, weighted by their degree of belongingness. If we define $\mu_{c,i}$ as the i-th element of the c-th centroid, and $x_{p,i}$ as the i-th element of the p-th object in the data, the centroid calculation can be expressed with Eq. (9). It follows that all members of the data must have the **same dimensionality** in this case. As with the normal mean prototype, this centroid function might not be suitable for time-series, and it is common to first apply a transformation to the data and cluster in the resulting space. For instance, D'Urso and Maharaj (2009) used the autocorrelation function to extract a certain amount of coefficients, resulting in data with equal dimensionality, and performed fuzzy clustering on the autocorrelation coefficients. See Appendix F for an example.

$$\mu_{c,i} = \frac{\sum_{p=1}^{N} u_{p,c}^{m} x_{p,i}}{\sum_{p=1}^{N} u_{p,c}^{m}} \tag{9}$$

Finally, the objective is minimized iteratively by applying Eq. (10) a certain number of iterations or until convergence is assumed. In Eq. (10), $d_{a,b}$ represents the distance between the a-th member of the data and the b-th fuzzy centroid, so great care must be given to the shown indices p, c and q. It is clear from the equation that this update only depends on the

chosen fuzziness exponent and the distance measure.

$$u_{p,c} = \frac{1}{d_{p,c}^{\frac{2}{m-1}} \sum_{q=1}^{k} \left(\frac{1}{d_{p,q}}\right)^{\frac{2}{m-1}}} \tag{10}$$

Technically, fuzzy clustering can be repeated several times with different random starts, since u is initialized randomly. However, comparing the results would be difficult, since it could be that the values within u are shuffled but the overall fuzzy grouping remains the same, or changes very slightly, once the algorithm has converged.

Note that it is straightforward to change the fuzzy partition to a crisp one by taking the *argument* of the row-wise maxima of u and assigning the respective series to the corresponding cluster only.

5. Parallel computation

Using parallelization is not something that is commonly explored explicitly in the literature, but it is something that can be extremely useful in practical applications. In the case of time-series clustering, and particularly when using the DTW distance, parallel computation can result in a very significant reduction in execution times.

There are some important considerations when using parallelization. The amount of *parallel workers*, i.e. subprocesses that can each handle a given task, is dependent on the computer processor that is being used and the number of physical processing cores and logical threads it can handle. Each worker may require additional RAM, and R can only work with data that is loaded on RAM, so there is no way around this limitation. Finally, the overhead introduced for the orchestration of the parallel workers may be too large when compared to the amount of time each worker needs to finish their assigned tasks, which is especially true for relatively simple workloads. Therefore, using parallelization does **not** guarantee faster execution times, and should be tested in the context of a specific application.

Handling parallelization has been greatly simplified in R by different software packages. The implementations done in **dtwclust** use the **foreach** package (Analytics and Weston 2015b), since it provides a flexible way of specifying the parallel backend that should be used, and can be extended by other packages. See Appendix G for a specific example.

Before describing the different cases where **dtwclust** can take advantage of parallel computation, it should also be noted that, by default, there is only one level of parallelization. This means that all tasks performed by a given parallel worker are done sequentially, and they cannot take advantage of further parallelization even if there are workers available.

When performing partitional clustering, it is common to do many repetitions with different random seeds to account for different starting points. When many repetitions are specified directly to **dtwclust**, the package attempts to assign each repetition to a different worker. This is also the case when the function is called with several values of `k`, i.e. when different number of clusters are to be tested; they are detected in partitional, fuzzy and TADPole clustering.

Calculating distance matrices involves performing several independent pairwise comparisons. The default distance function included with **dtwclust** parallelizes said calculations by grouping the amount of comparisons in *chunks* and assigning a chunk to each parallel worker. For

instance, if 100 distance calculations are needed, and there are 4 workers available, each of them would get a chunk with 25 calculations. This should provide a good load balance, since each distance calculation is usually very fast. This *chunking* is also performed by the distance functions used by **dtwclust** which are registered with **proxy**, especifically `LB_Keogh`, `LB_Improved`, `SBD` and `dtw_lb`.

The included implementation of the TADPole and DBA algorithms also try to create chunks when calculating DTW distances. The amount of calculations needed should be relatively small, so parallelization may not always be worth it in that case.

Finally, in the context of partitional clustering, calculating time-series prototypes is something that is done many times each iteration. Since clusters are mutually exclusive, the prototype calculations can be done in parallel, but this is only worth it if the calculations are time consuming. Therefore, in **dtwclust**, only DBA and shape extraction attempt to do parallelization when used in partitional clustering.

6. Cluster evaluation

Clustering is commonly considered to be an unsupervised procedure, so evaluating its performance can be rather subjective. However, a great amount of effort has been invested in trying to standardize cluster evaluation metrics by using *cluster validity indices* (CVIs). Many indices have been developed over the years, and they form a research area of their own, but there are some overall details that are worth mentioning. The discussion here is based on Arbelaitz *et al.* (2013), which provides a much more comprehensive overview.

CVIs can be classified as **internal**, **external** or **relative** depending on how they are computed. Focusing on the first two, the crucial difference is that internal CVIs only consider the partitioned data and try to define a measure of cluster purity, whereas external CVIs compare the obtained partition to the correct one. Thus, external CVIs can only be used if the *ground truth* is known.

Each index defines its range of values and whether they are to be minimized or maximized. In many cases, these CVIs can be used to evaluate the result of a clustering algorithm regardless of how the clustering works internally, or how the partition came to be. The Silhouette index is an example of an internal CVI, whereas the Variation of Information (Meilă 2003) is an external CVI. Several CVIs are included with **dtwclust** in the `cvi` function, although other indices can be implemented by the user.

Knowing which CVI will work best cannot be determined *a priori*, so they should be tested for each specific application. Usually, many CVIs are utilized and compared to each other, maybe using a majority vote to decide on a final result. Furthermore, it should be noted that many CVIs perform additional distance and/or centroid calculations when being computed, which can be very considerable if using DTW. For example, the Silhouette index effectively needs the whole distance matrix between the original series to be calculated, making it prohibitive in some cases.

Note that even though a fuzzy partition can be changed into a crisp one, making it compatible with many of the existing CVIs, there are also fuzzy CVIs tailored specifically to fuzzy clustering, and these may be more suitable in those situations.

7. Conclusion

In this paper, a general overview of the current state of the art in time-series clustering was provided. This included a lot of information related to the DTW distance and its corresponding optimizations, such as constraints and lower bounding techniques. At the same time, the **dtwclust** package for R was described and showcased, demonstrating how it can be used to test and compare different procedures efficiently and unbiasedly by providing a common infrastructure.

The goal was not to give a comprehensive and thorough explanation of all the discussed algorithms, but rather to provide information related to what has been done in the literature, including some more recent propositions, so that the reader knows where to start looking for further information, as well as what can or cannot be done with **dtwclust**.

Choosing a specific clustering algorithm for a given application is not an easy task. There are many factors to take into account and it is not possible to know *a priori* which one will yield the best results. The included implementations try to use the native (and heavily optimized) R functions as much as possible, relying on compiled code were needed, and we hope that, if time-series clustering is required, **dtwclust** can serve as a starting point.

References

Aggarwal CC, Hinneburg A, Keim DA (2001). "On the Surprising Behavior of Distance Metrics in High Dimensional Space." In JV den Bussche, V Vianu (eds.), *International Conference on Database Theory*, pp. 420–434. Springer.

Aggarwal CC, Reddy CK (2013). "Time-Series Data Clustering." In *Data Clustering: Algorithms and Applications*, chapter 15. CRC Press.

Aghabozorgi S, Shirkhorshidi AS, Wah TY (2015). "Time-Series Clustering–A Decade Review." *Information Systems*, **53**, 16–38.

Analytics R, Weston S (2015a). ***doParallel**: Foreach Parallel Adaptor for the '**parallel**' Package*. R package version 1.0.10, URL https://CRAN.R-project.org/package=doParallel.

Analytics R, Weston S (2015b). ***foreach**: Provides Foreach Looping Construct for R*. R package version 1.4.3, URL https://CRAN.R-project.org/package=foreach.

Arbelaitz O, Gurrutxaga I, Muguerza J, PéRez JM, Perona I (2013). "An Extensive Comparative Study of Cluster Validity Indices." *Pattern Recognition*, **46**(1), 243–256.

Begum N, Ulanova L, Wang J, Keogh E (2015). "Accelerating Dynamic Time Warping Clustering with a Novel Admissible Pruning Strategy." In *Conference on Knowledge Discovery and Data Mining*, KDD '15. ACM. ISBN 978-1-4503-3664-2/15/08. doi:http://dx.doi.org/10.1145/2783258.2783286.

Berndt DJ, Clifford J (1994). "Using Dynamic Time Warping to Find Patterns in Time Series." In *KDD Workshop*, volume 10, pp. 359–370. Seattle, WA.

Bezdek JC (1981). "Pattern Recognition with Fuzzy Objective Function Algorithms."

D'Urso P, Maharaj EA (2009). "Autocorrelation-Based Fuzzy Clustering of Time Series." *Fuzzy Sets and Systems*, **160**(24), 3565–3589.

Giorgino T (2009). "Computing and Visualizing Dynamic Time Warping Alignments in R: The **dtw** Package." *Journal of Statistical Software*, **31**(7), 1–24. URL http://www.jstatsoft.org/v31/i07/.

Hastie T, Tibshirani R, Friedman J (2009). "Cluster Analysis." In *The Elements of Statistical Learning 2nd Edition*, chapter 14.3. New York: Springer.

Kaufman L, Rousseeuw PJ (1990). *Finding Groups in Data. An Introduction to Cluster Analysis*, volume 1, chapter 1. John Wiley & Sons.

Keogh E, Ratanamahatana CA (2005). "Exact Indexing of Dynamic Time Warping." *Knowledge and information systems*, **7**(3), 358–386.

Lemire D (2009). "Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound." *Pattern Recognition*, **42**(9), 2169 – 2180. ISSN 0031-3203. doi:http://dx.doi.org/10.1016/j.patcog.2008.11.030. URL http://www.sciencedirect.com/science/article/pii/S0031320308004925.

Liao TW (2005). "Clustering of Time Series Data: A Survey." *Pattern recognition*, **38**(11), 1857–1874.

Lichman M (2013). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml.

Meilă M (2003). "Comparing Clusterings by the Variation of Information." In *Learning Theory and Kernel Machines*, pp. 173–187. Springer.

Meyer D, Buchta C (2016). ***proxy**: Distance and Similarity Measures*. R package version 0.4-16, URL https://CRAN.R-project.org/package=proxy.

Montero P, Vilar JA (2014). "**TSclust**: An R Package for Time Series Clustering." *Journal of Statistical Software*, **62**(1), 1–43. URL http://www.jstatsoft.org/v62/i01/.

Paparrizos J, Gravano L (2015). "k-Shape: Efficient and Accurate Clustering of Time Series." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pp. 1855–1870. ACM, New York, NY, USA. ISBN 978-1-4503-2758-9. doi:10.1145/2723372.2737793.

Petitjean F, Ketterlin A, Gançarski P (2011). "A Global Averaging Method for Dynamic Time Warping, with Applications to Clustering." *Pattern Recognition*, **44**(3), 678 – 693. ISSN 0031-3203. doi:http://dx.doi.org/10.1016/j.patcog.2010.09.013. URL http://www.sciencedirect.com/science/article/pii/S003132031000453X.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rakthanmanon T, Keogh EJ, Lonardi S, Evans S (2011). "Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data." In *2011 IEEE 11th International Conference on Data Mining*, pp. 547–556. IEEE.

Rani S, Sikka G (2012). "Recent Techniques of Clustering of Time Series Data: A Survey." *International Journal of Computer Applications*, **52**(15).

Ratanamahatana A, Keogh E (2004). "Everything you Know About Dynamic Time Warping is Wrong." In *3rd Workshop on Mining Temporal and Sequential Data, in Conjunction with 10th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD-2004), Seattle, WA*.

Sakoe H, Chiba S (1978). "Dynamic Programming Algorithm Optimization for Spoken Word Recognition." *Acoustics, Speech and Signal Processing, IEEE Transactions on*, **26**(1), 43–49. ISSN 0096-3518. doi:10.1109/TASSP.1978.1163055.

Wang X, Mueen A, Ding H, Trajcevski G, Scheuermann P, Keogh E (2013). "Experimental Comparison of Representation Methods and Distance Measures for Time Series Data." *Data Mining and Knowledge Discovery*, **26**(2), 275–309. ISSN 1384-5810. doi:10.1007/s10618-012-0250-5. URL http://dx.doi.org/10.1007/s10618-012-0250-5.

A. Technical notes

The main clustering function in **dtwclust** has the same name, i.e. `dtwclust`. It returns objects of type `S4`, which are one way in which R implements classes. The formal elements of the class are called `slots`, and can be accessed with the `@` operator (instead of the usual `$`). All documented information about the slots can be found by typing `?"dtwclust-class"` in the console. The associated methods can be found with `?"dtwclust-methods"`. Note that the `dtwclust` class contains the `S3` class `hclust` as a superclass. This can lead to some unexpected behavior, such as `is.list(new("dtwclust"))` being `TRUE`.

Many control parameters can be modified by using the `control` argument, which should have a `dtwclustControl` class. A **named list** is also accepted. The documentation of all control parameters, as well as their default values, can be accessed by typing `?"dtwclustControl"` in the console.

The random number generator of R is set to L'Ecuyer-CMRG when **dtwclust** is attached in an attempt to preserve reproducibility. The user is free to change this afterwards by using the `RNGkind` function.

B. Registering a custom distance with proxy

This example takes the autocorrelation-based distance included in the **TSclust** package (Montero and Vilar 2014) and registers it with **proxy** so that it can be used either directly, or with **dtwclust**.

```
require(TSclust)

proxy::pr_DB$set_entry(FUN = diss.ACF, names=c("ACFD"),
                       loop = TRUE, type = "metric", distance = TRUE,
                       description = "Autocorrelation-based distance")

# Taking just a subset of the data
# Note that subsetting with single brackets preserves the list format
proxy::dist(CharTraj[3:8], method = "ACFD", upper = TRUE)
```

```
##            A.V3       A.V4       A.V5       B.V1       B.V2       B.V3
## A.V3            0.53992661 0.52847864 1.78649047 0.81396557 0.38500499
## A.V4 0.53992661            0.06333488 4.00572755 2.46889508 1.47219504
## A.V5 0.52847864 0.06333488            4.07171283 2.60391486 1.66461589
## B.V1 1.78649047 4.00572755 4.07171283            0.30909583 0.98756308
## B.V2 0.81396557 2.46889508 2.60391486 0.30909583            0.20524086
## B.V3 0.38500499 1.47219504 1.66461589 0.98756308 0.20524086
```

C. Finding nearest neighbors in DTW space

In the following example, the nearest neighbors in DTW space of the first 5 time-series are found in two different ways, first calculating all DTW distance, and then using `dtw_lb` to

leverage `LB_Improved`. Since the LB is only defined for series of equal length, reinterpolation is performed.

```
# Reinterpolate to same length
data <- reinterpolate(CharTraj, new.length = max(lengths(CharTraj)))

# Calculate the DTW distances between all elements
system.time(D1 <- proxy::dist(data[1L:5L], data[6L:100L],
                              method = "dtw_basic",
                              window.size = 20L))

##    user  system elapsed
##   0.292   0.000   0.291

# Nearest neighbors
NN1 <- apply(D1, 1L, which.min)

# Calculate the distance matrix with dtw_lb
system.time(D2 <- dtw_lb(data[1L:5L], data[6L:100L],
                         window.size = 20L))

##    user  system elapsed
##   0.160   0.004   0.164

# Nearest neighbors
NN2 <- apply(D2, 1L, which.min)

# Same results?
all(NN1 == NN2)

## [1] TRUE
```

D. Hierarchical clustering example

In the following call to **dtwclust**, specifying the value of `k` indicates the number of desired clusters, so that the `cutree` function is called internally. Additionally, the shape extraction **function** is provided in the `centroid` argument so that, once the `k` clusters are obtained, their prototypes are extracted. Therefore, the data is z-normalized by means of the `zscore` function. The seed is provided because of the randomness in shape extraction when choosing a reference series (see Section 3.4).

```
hc_sbd <- dtwclust(CharTraj, type = "h", k = 20L,
                   method = "average", preproc = zscore,
                   distance = "sbd", centroid = shape_extraction,
                   seed = 899, control = list(trace = TRUE))
```

```
##
##  Calculating distance matrix...
##
##  Performing hierarchical clustering...
##
##  Elapsed time is 1.275 seconds.

# Cluster sizes
table(hc_sbd@cluster)

##
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
## 15 10 18  8  2  9  1  4  1  5  5  1  1  5  1  4  5  1  3  1

# By default, the dendrogram is plotted in hierarchical clustering
plot(hc_sbd)
```

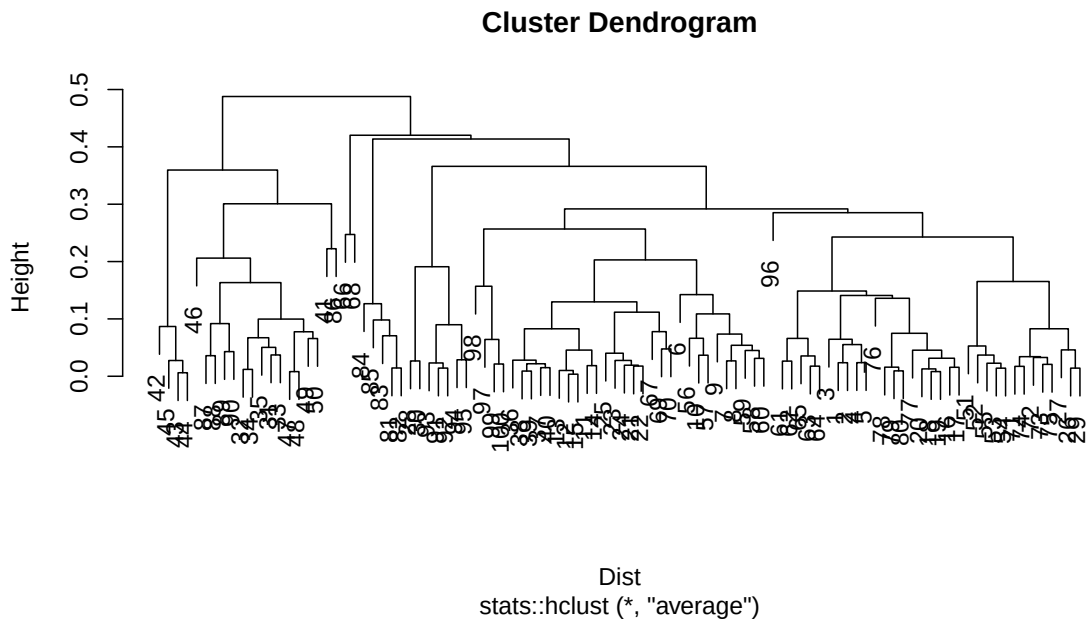


Figure 8: Resulting dendrogram after hierarchical clustering.

```
# The series and the obtained prototypes can be plotted too
plot(hc_sbd, type = "sc")
```

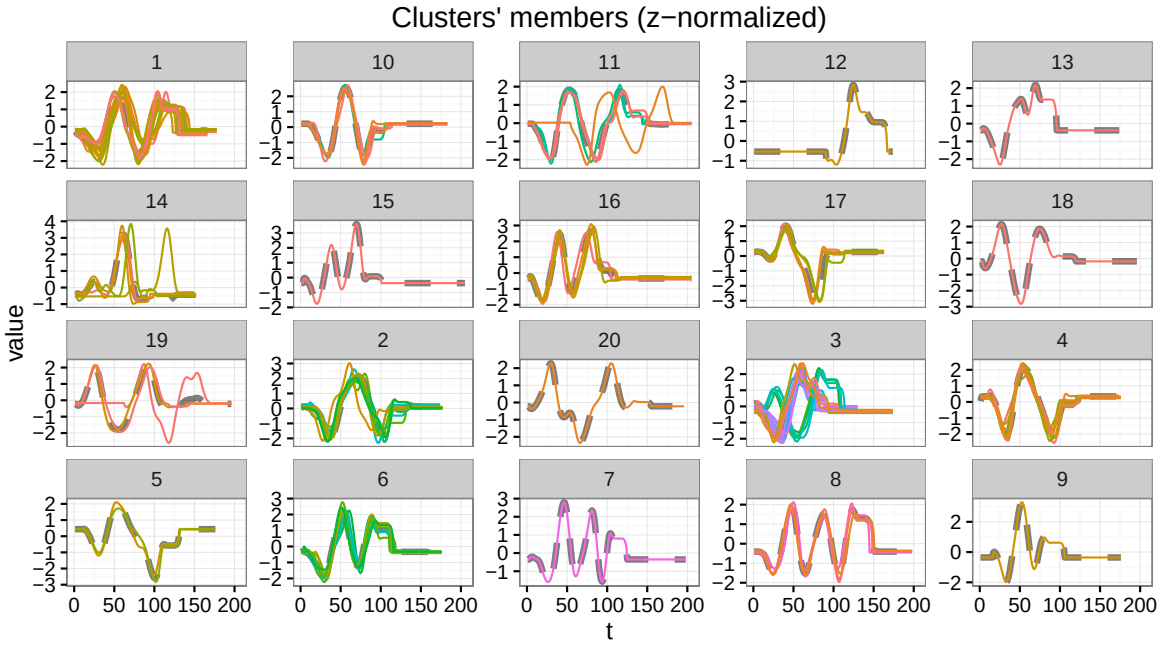
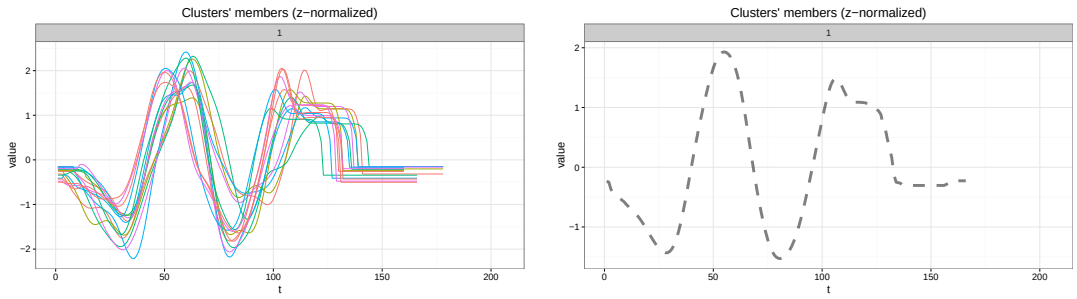


Figure 9: Obtained clusters and their respective prototypes (centroids) shown as dashed lines.

```
# Focusing on the first cluster
plot(hc_sbd, type = "series", clus = 1L)
plot(hc_sbd, type = "centroids", clus = 1L)
```



(a) Series in cluster 1

(b) Prototype obtained by applying shape extraction to cluster 1

Figure 10: Side by side comparison of the series in the first cluster and the obtained prototype.

E. Partitional clustering example

In this example, four different partitional clustering strategies are used: one uses the DTW_2 distance and DBA centroids, then `dtw_lb` and DBA centroids are used (which provides the same results as using the DTW distance directly; see Section 2.1.2), then k-Shape, and finally TADPole. The results are evaluated using Variation of Information (see Section 6), with

lower numbers indicating better results. Note that z-normalization is applied by default when selecting shape extraction as the centroid function. For fairness, all algorithms used the reinterpolated and normalized data, since some algorithms require series of equal length. A subset of the data is used for speed. The outcome should not be generalized to other data, and normalization/reinterpolation may actually *worsen* some of the algorithms' performance.

```
# Reinterpolate to same length
data <- reinterpolate(CharTraj, new.length = max(lengths(CharTraj)))

# z-normalization
data <- zscore(data[60L:100L])

# Extra 'trace' is for DBA
pc_dtw <- dtwclust(data, k = 4L,
                   distance = "dtw_basic", centroid = "dba",
                   trace = TRUE, seed = 8,
                   control = list(window.size = 20L,
                                  norm = "L2",
                                  trace = TRUE))

##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
## Iteration 1: Changes / Distsum = 41 / 274.5699
## Iteration 2: Changes / Distsum =  0 / 173.202
##
##  Elapsed time is 4.106 seconds.

pc_dtwlb <- dtwclust(data, k = 4L,
                     distance = "dtw_lb", centroid = "dba",
                     trace = TRUE, seed = 8,
                     control = list(window.size = 20L,
                                    norm = "L2",
                                    trace = TRUE))

##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
##   11, 12, 13, 14, 15, Did not 'converge'
##  DBA Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

```
##   11, 12, 13, 14, 15, Did not 'converge'
## Iteration 1: Changes / Distsum = 41 / 274.5699
## Iteration 2: Changes / Distsum =  0 / 173.202
##
##  Elapsed time is 4.178 seconds.

pc_ks <- dtwclust(data, k = 4L,
                  distance = "sbd", centroid = "shape",
                  seed = 8, control = list(trace = TRUE))

## Iteration 1: Changes / Distsum = 41 / 8.625422
## Iteration 2: Changes / Distsum = 8 / 6.107082
## Iteration 3: Changes / Distsum = 3 / 5.374779
## Iteration 4: Changes / Distsum = 2 / 5.019199
## Iteration 5: Changes / Distsum =  2 / 4.89067
## Iteration 6: Changes / Distsum = 3 / 4.696871
## Iteration 7: Changes / Distsum = 2 / 4.492085
## Iteration 8: Changes / Distsum = 0 / 4.322737
##
##  Elapsed time is 1.072 seconds.

pc_tp <-dtwclust(data, k = 4L,
                 type = "tadpole", dc = 1.5,
                 seed = 8, control = list(window.size = 20L,
                                          trace = TRUE))

##
## Entering TADPole...
##
## TADPole completed, pruning percentage = 62.1%
##
##  Elapsed time is 0.556 seconds.

sapply(list(DTW = pc_dtw, DTW_LB = pc_dtwlb, kShape = pc_ks, TADPole = pc_tp),
       cvi, b = CharTrajLabels[60L:100L], type = "VI")

##     DTW.VI  DTW_LB.VI  kShape.VI TADPole.VI
##  0.5017081  0.5017081  0.4353306  0.4901096
```

F. Fuzzy clustering example

This example performs autocorrelation-based fuzzy clustering as proposed by D'Urso and Maharaj (2009). Using the autocorrelation function overcomes the problem of time-series with different length.

```
# Calculate autocorrelation up to 50th lag
acf_fun <- function(dat, ...) {
     lapply(dat, function(x) {
          as.numeric(acf(x, lag.max = 50, plot = FALSE)$acf)}
     )
}

# Fuzzy c-means
fc <- dtwclust(CharTraj[1:20], type = "f", k = 4L,
               preproc = acf_fun, distance = "L2",
               seed = 42)

# Fuzzy membership matrix
fc@fcluster
```

```
##        cluster_1   cluster_2   cluster_3    cluster_4
## A.V1 0.943926036 0.010596011 0.020908497 0.0245694559
## A.V2 0.971956494 0.004723638 0.010179148 0.0131407207
## A.V3 0.907198198 0.013779291 0.027675078 0.0513474332
## A.V4 0.491187667 0.211376339 0.217256114 0.0801798799
## A.V5 0.561169672 0.171553489 0.186792803 0.0804840359
## B.V1 0.129884957 0.035294178 0.084089357 0.7507315074
## B.V2 0.011132604 0.002320093 0.005101198 0.9814461041
## B.V3 0.192006808 0.032433530 0.060851157 0.7147085050
## B.V4 0.161841130 0.030735878 0.049368720 0.7580542718
## B.V5 0.427494247 0.234508791 0.186845220 0.1511517421
## C.V1 0.311548957 0.047712389 0.199603474 0.4411351802
## C.V2 0.007122008 0.002616213 0.986827321 0.0034344574
## C.V3 0.076269200 0.051830617 0.838928681 0.0329715023
## C.V4 0.340136370 0.055547457 0.358925482 0.2453906915
## C.V5 0.015265286 0.006000406 0.970835866 0.0078984424
## D.V1 0.017768186 0.958603287 0.016188869 0.0074396587
## D.V2 0.048245701 0.902805583 0.030486494 0.0184622220
## D.V3 0.002215636 0.994977666 0.001844308 0.0009623899
## D.V4 0.004964049 0.988851870 0.004022021 0.0021620601
## D.V5 0.018863188 0.954832487 0.017566807 0.0087375175
```

```
# Are constraints fulfilled?
all.equal(rep(1, 20), rowSums(fc@fcluster), check.attributes = FALSE)
```

```
## [1] TRUE
```

```
# Plot crisp partition in the original space
plot(fc, data = CharTraj[1:20], type = "series")
```

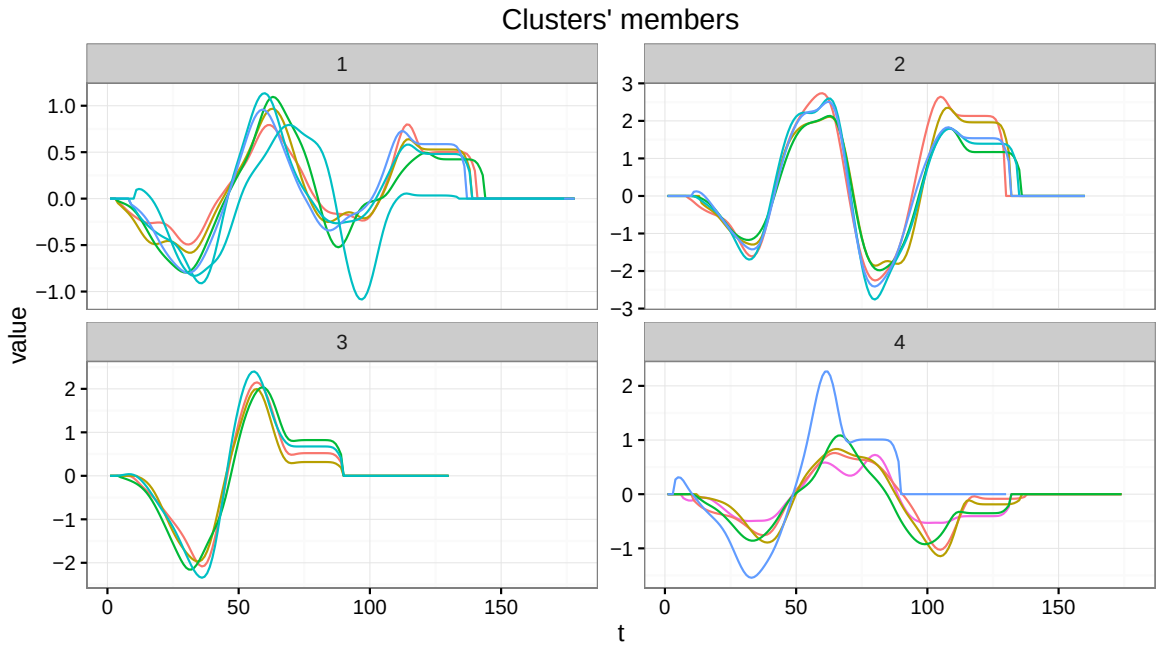


Figure 11: Visualization of the clusters that would result from changing the fuzzy partition to a crisp one. Note that the original time-series are used, so the centroids are not shown, since the centroids are made of autocorrelation coefficients.

G. Using the doParallel package for parallel computation

One way of using parallelization with R and **foreach** is by means of the **doParallel** package (Analytics and Weston 2015a). It provides a great level of abstraction so that users can easily configure a parallel backend which can be used by **dtwclust**. The example below does the backend registration and calls **dtwclust**, returning to sequential computation after it finishes. It only uses 2 parallel workers, but more can be configured depending on each processor (see function `detectCores` in R).

```
require(doParallel)

# Create parallel workers
workers <- makeCluster(2L)

# Preload dtwclust in each worker; not necessary but useful
invisible(clusterEvalQ(workers, library(dtwclust)))

# Register the backend; this step MUST be done
registerDoParallel(workers)

# Calling dtwclust
pc_par <- dtwclust(CharTraj[1L:20L], k = 4L,
                   distance = "dtw_basic", centroid = "dba",
                   seed = 938, control = list(trace = TRUE,
```

```
                                                window.size = 15L))

## Iteration 1: Changes / Distsum = 20 / 241.6139
## Iteration 2: Changes / Distsum = 0 / 148.4783
##
##  Elapsed time is 1.5 seconds.

# Stop parallel workers
stopCluster(workers)

# Go back to sequential computation
registerDoSEQ()
```

Affiliation:

Alexis Sardá-Espinosa
alexis.sarda@gmail.com