ChatGPT

Time Series Clustering

Time series clustering involves partitioning a set of time-dependent data series into groups (clusters) such that series in the same cluster are more similar to each other than to those in other clusters [1]. It is an unsupervised learning task, meaning the clustering is done without predefined labels. The notion of "similarity" here depends on how we measure distance or similarity between time series and the objective of the analysis. Key components of a time-series clustering procedure include: (1) the choice of distance or similarity measure between time series, (2) the clustering algorithm or method used to form clusters, and (3) the evaluation of the clustering result [2]. In literature, time-series clustering methods are often categorized by how they handle the data, for example as **shape-based**, **feature-based**, or **model-based** approaches [3] (and we will also discuss some **structure-based** measures and other modern techniques). Below we outline the major aspects of time-series clustering:

Distance and Similarity Measures

A fundamental step is defining a distance or dissimilarity measure between time series. This measure determines what it means for two series to be "similar." Different types of distance measures have been developed for time-series, which we group into shape-based (direct comparison of the time-series sequences), feature-based (comparison of derived attributes), and structure-based (comparison of underlying model or information content) categories.

Shape-Based Measures

Shape-based distances compare time series *point-by-point* based on their raw shape (the sequence of values). These methods typically require the series to be aligned in time (or they employ mechanisms to align them) and focus on matching the observed patterns. We further distinguish **lock-step** distances (which compare values at the same time indices) and **elastic** distances (which allow some stretching or shifting of the time axis to find a better match).

Lock-step distances: These treat the time series as ordinary vectors and compare each time index directly, without any shifting. The series must be of equal length and aligned in time. Common lock-step distance metrics include:

- **Euclidean distance:** The straight-line (L^2 norm) distance between two time-series interpreted as vectors. For series $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, Euclidean distance is $\sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$. This is a simple and widely used metric; however, it is sensitive to phase shifts or misalignments (not invariant to time shifting) [4]. Even if two series have the same shape but one is lagged in time, Euclidean distance can be large.

- **Manhattan distance:** The sum of absolute differences (L^1 norm), $\sum_{i=1}^{n} |x_i - y_i|$. This metric is also straightforward and less sensitive to outliers than Euclidean, but similarly requires alignment.

- **Minkowski distance:** A generalization of Euclidean and Manhattan; L^p norm distance $(\sum_i |x_i - y_i|^p)^{1/p}$. For example, $p = 2$ gives Euclidean and $p = 1$ gives Manhattan. It allows tuning the emphasis on larger vs smaller differences via the parameter p.

- **Chebyshev (Maximum) distance:** The maximum absolute difference on any dimension (time point), i.e. $\max_i |x_i - y_i|$. It measures the worst-case deviation between two series at any time point.

- **Canberra distance:** A weighted version of Manhattan distance: $\sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i| + |y_i|}$. Each term is a fraction that ranges between 0 and 2, giving higher weight to differences when the values are small (since the denominator will be small). This can be useful when relative differences (percent changes) are more important than absolute differences.

- **Binary distance:** A distance for binary-valued series (containing 0/1 or TRUE/FALSE values). One example is the Hamming distance (count of positions where the two series have different values). This assumes the series are first transformed to a binary representation. The binary distance essentially counts mismatches and is appropriate when time series represent sequences of categorical events or binary states.

- **Correlation-based distance:** Instead of comparing raw values, one can measure how well correlated the two series are, and turn that into a distance. For example, **Pearson correlation** measures linear correlation (shape similarity regardless of scale/mean); a common approach is to define distance = 1 − Pearson's correlation coefficient [5] . High correlation (close to 1) means the series have a similar overall shape (even if one is an amplitude-scaled or shifted version of the other in value), yielding a small distance. **Spearman** or **Kendall** correlation can be used similarly to capture monotonic relationship or rank-based similarity. Correlation-based distances focus on the shape pattern rather than absolute differences [5] . Note that using correlation as a distance implies ignoring differences in mean and amplitude (after appropriate normalization), focusing purely on shape synchronization.

- **Cross-correlation similarity:** This is related to correlation but allows a lag. The cross-correlation function finds the correlation between one series and a lagged version of the other. A high maximum cross-correlation (at some optimal lag) indicates that one series is very similar to a shifted version of the other. One can define a distance based on the **maximum cross-correlation** (the higher the max correlation, the smaller the distance). In practice, one might align the series by the lag that maximizes correlation and then measure dissimilarity. This is useful if one series is a time-shifted version of the other.

- **Dissimilarity index (Dissim):** A specialized lock-step measure that can handle uneven sampling or time gaps. The **Dissim distance** integrates the pointwise difference between two time series over time [6] . It treats each series as a continuous function (assuming linear interpolation between sample points) and computes the area between their curves [6] . This measure allows the two series to have different sampling rates or time points (as long as they cover the same time span) by integrating over the continuous domain. If the series are identical when superimposed in time, the Dissim distance will be 0; the more they diverge, or the more one leads/lags the other without overlap, the larger the distance.

Many of these lock-step distances are available via base R functions (e.g., `dist()` can compute Euclidean, Manhattan, etc., on multivariate data) or via the `proxy` package which offers a variety of distance metrics. For time series specifically, the **TSdist** package provides implementations (e.g., `EuclideanDistance`, `ManhattanDistance`, etc.), and the **TSclust** package (from Montero & Vilar, 2014) has corresponding functions (prefixed with `diss.`) for certain measures.

Elastic distances: Lock-step measures can fail to capture similarity when series are misaligned in time or have different speeds (one series lags or stretches relative to the other). Elastic measures address this by allowing *non-linear alignments* of the time axis [7]. They "stretch" or "warp" the series along the time dimension to find an optimal alignment that minimizes distance. Here are some important elastic measures:

- **Dynamic Time Warping (DTW):** DTW is one of the most popular elastic distances for time series. It uses a dynamic programming algorithm to find the minimal “warp cost” alignment between two sequences [8]. Essentially, DTW will repeat or skip points (within constraints) so that the overall distances between aligned points is as small as possible. This allows DTW to handle sequences that are similar in shape but not perfectly synchronized in time [7]. For example, if one series is a slower version of another (same pattern stretched out), DTW can align the slower series to the faster one by mapping multiple points in the slower series to one point in the fast series, and vice versa, so that similar patterns line up. The result is a distance value (the sum of pointwise differences along the optimal warping path) which is small for series of similar shape, even if they are phase-shifted or have different lengths. DTW is widely used in time-series clustering and classification [8]. In R, the **dtw** package can compute DTW alignments and distances, and **dtwclust** uses DTW extensively for clustering.

- **Fréchet distance:** The Fréchet distance is a measure originally defined for curves (continuous trajectories) which intuitively represents the minimum “leash length” needed for a person to walk a dog so that the person walks along one curve and the dog along the other, without backtracking [9]. In time series terms, it considers the location and ordering of points along the curves. It finds an optimal alignment like DTW but emphasizes the overall shape of the trajectories. The discrete Fréchet distance can be computed for time series and is sensitive to both magnitude and the sequence of points. It is a true metric (unlike DTW, which does not satisfy triangle inequality in general). There are R implementations (e.g., the `tscR` package provides `frechetDist` for pairwise Fréchet distances [10]).

- **Longest Common Subsequence (LCSS):** LCSS is a similarity measure that focuses on finding a long matching subsequence between two time series, allowing for some elements to be unmatched. It counts the length of the longest sequence of time indices in which the two series match (within a tolerance) when allowed to skip unmatched parts [11]. One can specify a distance threshold epsilon (and sometimes a maximum allowed index shift) and count a match whenever two points are within epsilon in value. The LCSS distance can be defined as $1 - \frac{\mathrm{LCSLength}}{\min(n,m)}$ or simply as the complement of the similarity score [12]. LCSS is robust to outliers and gaps because it can ignore parts of the series that do not match, focusing on the longest similar portions. This method requires tuning a match threshold and is not a metric in the strict sense, but it’s useful when we expect only portions of the series to be similar. The TSdist package provides `LCSSDistance` for computing this measure.

- **Shape-Based Distance (SBD):** The shape-based distance was proposed by Paparrizos and Gravano (2015) as part of the k-Shape clustering algorithm [13]. SBD uses normalized cross-correlation to compare the shapes of two series. It effectively shifts one series relative to the other (circularly, via convolution) to find the alignment with maximum correlation, and then converts that into a distance [14] [15]. The formula for SBD is often given as $\mathrm{distance} = 1 - \max_{\mathrm{lag}} \mathrm{NCCc}(x, y)$, where NCCc is the normalized cross-correlation sequence [16]. SBD is invariant to circular time shifts and is computed efficiently using the Fast Fourier Transform. It assumes z-normalized series (zero mean, unit variance) so that only the shape (pattern of ups and downs) matters [17]. In practice, SBD is faster than DTW and works well when the primary

misalignment is a simple shift (rather than complex warping). The **dtwclust** package implements SBD (via the `SBD` function) and uses it in the k-Shape algorithm.

- **Shapelet-based (U-Shapelets):** *Shapelets* are small, discriminative subsequences that can represent local patterns in time series. In clustering, shapelet-based approaches transform each series into a feature vector of distances to a set of shapelets (which can be learned or extracted from the data). **U-Shapelets** refers to an unsupervised shapelet discovery method (unlike supervised shapelets originally developed for classification). Essentially, if certain sub-patterns frequently occur in a subset of series, those can define clusters. A series is represented by how well it matches each shapelet (usually the minimum distance to that shapelet in the series). Clustering is then done in this new feature space [18]. Shapelet approaches focus on *local* shape similarities. While not as widely used as a general-purpose method, they can capture clusters characterized by presence or absence of particular patterns. Implementations are less common in R; one might need to use Python libraries or custom code for unsupervised shapelet learning.

- **Kernel-based measures (Global Alignment Kernel):** Instead of explicit distances, one can use similarity *kernels* for time series and then apply kernel-based clustering (like kernel k-means or spectral clustering). A kernel is a function that returns a similarity score (larger for more similar objects) and implicitly defines a distance. One example is the **Global Alignment Kernel (GAK)** proposed by Cuturi et al., which is a positive-definite kernel based on a soft-DTW alignment cost [8]. GAK essentially sums over all possible alignments between two series, weighting them by how well they align, to produce a similarity measure. It can be viewed as a smoothed version of DTW that is symmetric and mercerized (i.e., valid kernel). While not directly a distance, it can be used in clustering by kernel methods. In R, one might find GAK in the `dynafit` or `kertime` packages, but it's not as commonly used as in Python (where libraries like `tslearn` provide it). Other kernels include specialized wavelet kernels or shapelet transform kernels. Using kernels can sometimes be advantageous in clustering by enabling the use of powerful kernel clustering algorithms.

Summary of shape-based measures: Shape-based distances are very intuitive because they try to match the actual time-series curves. Lock-step distances are simple and fast but require well-aligned data. Elastic distances handle misalignments and different lengths at the cost of higher computation (DTW and related methods are often $O(n^2)$ in the length of series, though there are speed-ups and constraints). In R, the **TSdist** package offers a comprehensive collection of these measures (Euclidean, DTW, LCSS, Frechet, CID, NCD, etc.), and **dtwclust** builds on many of them for clustering. Choosing a shape-based measure often depends on the nature of your data: if you expect similar shapes occurring out of phase, an elastic measure like DTW or SBD is more appropriate; if all series are roughly aligned events, lock-step might suffice.

Feature-Based Measures

Feature-based clustering means instead of comparing raw time series point-by-point, we first extract a set of descriptive features from each time series and then apply clustering on these feature vectors [19]. This approach reduces the problem to clustering in a feature space (much like clustering static data), leveraging characteristics of the series. It is especially useful to reduce dimensionality or to incorporate domain knowledge into features. Common feature-based methods include:

- **Autocorrelation and Partial Autocorrelation (ACF/PACF):** We can characterize a time series by its autocorrelation coefficients at various lags. The **Autocorrelation Function (ACF)** gives the correlation of the series with itself at lag 1, 2, etc., up to some maximum lag. Similarly, **Partial Autocorrelation (PACF)** controls for intermediate lags. These coefficients (a fixed number of

them) form a feature vector for the series [20]. For example, if two time series both exhibit strong seasonality at a certain lag, their autocorrelation feature for that lag will be high, making them appear closer in feature space. Autocorrelation-based distance measures (like those implemented in TSclust/TSdist as `ACFDistance` and `PACFDistance`) essentially compute the Euclidean distance between vectors of autocorrelation coefficients [20]. This approach captures similarity in the *structure* of temporal dependence (useful for distinguishing e.g. AR(1) vs AR(2) behavior, seasonal patterns, etc.).

- **Fourier spectrum (Frequency domain features):** Using the Fourier transform, each time series can be represented by its frequency spectrum or periodogram. **Fourier features** might include the dominant frequencies, spectral entropy, total power in certain frequency bands, etc. [19]. If two series have similar periodic behavior (say annual seasonality or similar oscillation frequency), their frequency-domain representations will be similar. Clustering in the frequency domain can thus group series by similar periodic patterns even if their time-domain waveforms differ in phase or amplitude. TSdist provides measures like `PerDistance` (periodogram-based distance) and `IntPerDistance` (integrated periodogram distance) that effectively compare spectral density estimates [21] [22]. These focus on whether two series have comparable power distribution across frequencies.

- **Wavelet features:** Wavelet transforms provide time-localized frequency information. By taking a discrete wavelet transform (DWT) of each series, we obtain coefficients corresponding to different frequency bands and time locations. One can use summary statistics of certain wavelet coefficient bands (e.g., energy in each band, or specific coefficient values) as features. This captures both scale (frequency) and localized time information, which can be useful if, say, two series both have a particular transient pattern. Wavelet-based distances are less common but some researchers use them to cluster series by shape features at multiple scales.

- **Trend/Seasonality/Statistical summaries:** Simpler feature vectors can be constructed from high-level summary characteristics of the series. For example: mean, variance, skewness, kurtosis, trend slope (from a linear model), seasonal strength (from something like an STL decomposition), number of peaks, etc. These features turn each time series into a point in a multivariate space of interpretable attributes. Clustering based on these can group series that are similar in broad behavior (e.g., all increasing trends in one cluster, all flat in another; or high-volatility series vs low-volatility). While this loses detailed shape information, it can be very useful in exploratory clustering when the interest is in these broad characteristics.

- **Shape descriptors (e.g., coefficients from shape transformations):** Other transformations like Principal Component Analysis (PCA) on the space of time series, or singular value decomposition (SVD) of the lag covariance matrix (a form of spectral analysis), can yield features. For example, if one performs a Fourier or wavelet transform and takes the first few coefficients, those can serve as features (this is essentially a truncated basis expansion of the series).

- **Symbolic Aggregate approXimation (SAX):** SAX is a method to convert a time series into a string of symbols by first applying Piecewise Aggregate Approximation (segmenting the series and averaging in segments) and then discretizing those averages into symbol bins. This produces a string like "abbaac...". One can then define distances between series based on string distance (like edit distance or a specialized **MINDIST** for SAX). SAX distances capture coarse shape patterns in a discrete form [23]. TSdist includes `MindistSaxDistance` implementing the distance between SAX representations. SAX is good for dimensionality reduction and highlighting large-scale shape features; it's often used in combination with indexable representations for very large datasets.

In feature-based clustering, once each series is represented as a feature vector, any standard distance (Euclidean, Mahalanobis, etc.) can be used on those feature vectors, and any clustering algorithm can then be applied. The choice of features is critical and should be guided by domain knowledge or experiments. R has many tools for computing these features: e.g., `acf()` and `pacf()` for autocorrelations, the **TSA** or **forecast** package for spectral analysis, the **WaveletComp** or **waveslim** for wavelets, and **jmotif** or **TSMining** packages for SAX. The **feasts** package in R (part of the fable ecosystem) provides a convenient way to compute a large set of time series features (like those used in forecasting competitions) which can then be input to clustering. Packages like **tsfeatures** also compute many such features automatically.

Structure-Based Measures

Structure-based approaches compare the underlying generating mechanisms or information content of time series, rather than their observed values or simple features. These typically involve fitting a model to each series or using information theory and then measuring how different those models or representations are [24] [25]. This can capture deep similarities that might not be obvious from raw data (for example, two series might be offset in mean or have different variance but actually follow the same autoregressive process).

- **Model-based distances:** In this approach, we assume each time series is generated by some statistical model (like an ARIMA model, state-space model, etc.), and we compare those models. A classic example is to fit an autoregressive (AR) model to each series and use the distance between models as the dissimilarity. One such distance is based on the **AR model coefficients**: for instance, Piccolo (1990) defined a distance between two ARIMA models by looking at the difference in their autoregressive operator polynomials (essentially, treating the AR coefficients as a vector and using a weighted Euclidean distance) [26]. Another approach by Maharaj (2000) is a hypothesis test for equality of two time series' spectral distributions (or AR models); one can use the test's p-value or statistic as a dissimilarity measure [24]. In R, the TSclust package implements these as `diss.AR.PIC` and `diss.AR.MAH` (Maharaj) distances. These model-based distances are good for capturing similarity in *dynamics*: e.g., if two series are both essentially AR(1) with coefficient 0.9 (a slow decaying autocorrelation), they will appear similar by these measures even if one is higher in level or has different noise realizations.

- **Cepstral coefficients (LPC distance):** This is related to model-based distances. By using Linear Predictive Coding (LPC), one can derive **cepstral coefficients** from the time series (these are coefficients of the Fourier transform of the log spectrum; in time series context, they relate to AR model parameters). Cepstral coefficients compactly describe the spectral shape of a series. The distance between the cepstral coefficient vectors of two series is another way to gauge similarity of their spectral structure. TSclust provides `diss.AR.LPC.CEPS`, which computes a dissimilarity based on LPC-derived cepstral coefficients [27]. If two series have similar frequency content (even if time domain looks different), this distance will be small.

- **Compression-based distances:** These stem from the idea of Kolmogorov complexity. The intuition is that if two sequences are similar, then a combined sequence will compress almost as well as each individual sequence. A prominent measure is the **Normalized Compression Distance (NCD)** [28]. To compute NCD between series X and Y: you compute C(X) = compressed size of X, C(Y) for Y, and C(XY) for the concatenation of X and Y. Then $\mathrm{NCD}(X,Y) = \frac{C(XY) - \min(C(X), C(Y))}{\max(C(X), C(Y))}$. If X and Y share patterns, the compressed size of XY will not be much larger than that of X or Y alone, yielding a small NCD. If they are very different, C(XY) will be roughly sum of both sizes. The range is 0 to 1 (approximately). NCD is **parameter-free and very**

general – it uses the compressor (like gzip) to implicitly identify any kind of regularity. In time series, it can detect similarity even if one series is a noisy or scaled version of another, as long as the pattern is algorithmically similar. TSdist implements this as `NCDDistance`. Another compression-based measure is CDM (compressed dissimilarity measure) which might use specific compressors or normalized variants [25]. These methods are computationally heavier and sometimes less interpretable, but they are domain-agnostic.

- **Complexity-Invariant Distance (CID):** Proposed by Batista et al. (2011), CID adjusts the Euclidean distance by a factor related to the series' complexity. The idea is that two series might look different in raw values but if one is simply a more "wiggly" (higher complexity) version of the other, we might still consider them similar in shape. CID multiplies the Euclidean distance by a factor that is a function of the sum of squared differences between adjacent points (a measure of complexity) [29]. Specifically, $\mathrm{CID}(X,Y) = \mathrm{Euclid}(X,Y) \times \sqrt{\frac{C(X)}{C(Y)}}$ or the symmetric variant with both complexities, where $C(X) = \sum_i (x_i - x_{i-1})^2$. If both series have similar complexity, this factor is ~1; if one series is much more erratic than the other, the distance is scaled up or down accordingly. This helps clustering to not penalize differences purely due to one series having higher amplitude fluctuations. TSdist provides `CIDDistance` [29].

- **Permutation Distribution Distance:** This is based on the concept of **Permutation Entropy** – a way to capture the complexity of a time series by looking at the ordering of values. For a time series, one can look at short subsequences of length `m` and record the order pattern of values (e.g., for three points, one possible pattern is "x1 < x3 < x2"). The collection of these ordinal patterns forms a **permutation distribution** for the series. Similar time series will have similar distributions of ordinal patterns. The **Permutation Distribution Distance (PDD or PDCDistance)** compares the distributions of these ordinal patterns between two series [30]. If series X and Y produce the same patterns with similar frequencies, the distance is low. If their underlying dynamics (as reflected in these ordinal patterns) differ, the distance is high. This method is robust to nonlinear transformations and noise to some extent, as it only cares about order relations. The `pdc` package in R (Permutation Distribution Clustering) implements this approach (M. Bandt and colleagues introduced permutation entropy). TSdist also includes `PDCDistance` to compute this distance directly. This is particularly useful for detecting similarities in the rank dynamics (it's invariant to monotonic transformations of the data).

(There are other structure/information-based measures too, like those based on edit distance on symbolic sequences after quantization, but the above cover many of the popular ones.)

In R, **TSdist** and **TSclust** again provide many of these: for example, `MaharajDistance`, `PiccoloDistance`, `PredDistance` (which uses a measure based on forecasting model error), `SpecDistance` (integrated or local linear estimate of spectral density [31]), etc. The **pdc** package specifically focuses on permutation entropy based clustering. The right choice of structure-based measure depends on what "structure" you consider important – spectral properties, predictability, algorithmic compressibility, etc. These measures can be powerful but might be less intuitive than shape-based measures.

Clustering Methods

Once we can measure the distance or (dis)similarity between time series, we can apply a clustering algorithm. Most classical clustering algorithms can be used for time series, either directly on a feature representation or with a custom distance matrix (for algorithms that support arbitrary distances). The main categories of clustering methods include **partitioning methods**, **hierarchical methods**, **fuzzy**

clustering, **density-based methods**, **model-based methods**, **grid-based methods**, and emerging **deep learning-based methods** [32] . We describe each category and note any special considerations for time-series data.

Partitioning Methods

Partitioning (or partitional) clustering algorithms aim to directly divide the data into a pre-specified number of clusters (k clusters) by optimizing some criterion. These algorithms usually start with an initial partition or initial cluster centers and then iteratively refine the clusters. They treat the clustering as an optimization problem (e.g., minimizing within-cluster variance).

- **k-Means:** k-means is a classic algorithm that seeks to minimize the sum of squared Euclidean distances of points from their cluster centroids. It iterates between assigning each data point (series) to the nearest centroid and then updating centroids as the mean of the series in each cluster. In time-series context, one must be careful because taking a mean of time series implicitly assumes alignment. If using Euclidean distance on raw time points, one can compute the centroid series as the mean of all series in that cluster (at each time point). This works if the series are aligned and of equal length. However, if using a custom distance like DTW, the notion of an average is non-trivial. Specialized algorithms exist to compute a **DTW barycenter** (an average series under DTW distance) – e.g., DBA (DTW Barycenter Averaging) – which can be used in a k-means-like algorithm. The **dtwclust** package provides a DTW-based partitional clustering that essentially is "k-means" but with DTW and DBA in place of Euclidean and arithmetic mean [33] . In general, k-means is efficient and works well for convex clusters in vector space; for time series one might first transform to features or use it with caution on raw data. R's built-in `kmeans()` can be used on feature matrices. For distance matrices (like DTW distances), one can use **partitioning around medoids (PAM)** or other approaches (see k-medoids below).

- **K-Medoids (PAM and CLARA):** k-medoids is a robust alternative to k-means that chooses actual data points as cluster centers (medoids). The most common algorithm is PAM (Partitioning Around Medoids). Instead of means, each cluster is represented by one of the series (the medoid) that minimizes the total distance to all others in the cluster. This is advantageous when using arbitrary distance measures, because computing a "mean" might not be well-defined, but a medoid (most centrally located series) is. PAM is implemented in R's **cluster** package (`pam()` function) and can take a precomputed distance matrix (so it can directly use DTW, Frechet, etc.). This makes it very flexible for time series clustering with any distance. CLARA is a variant of PAM for large datasets (it samples subsets for efficiency). K-medoids clustering is often preferred for time series when using non-Euclidean distances because it avoids defining an average in an inhospitable space. The trade-off is it's somewhat less efficient than k-means (PAM is $O(n^2)$ in number of series, though CLARA mitigates this).

- **Self-Organizing Map (SOM):** A Self-Organizing Map is a type of artificial neural network that produces a low-dimensional (usually 2D) representation of the input space while preserving topology. It can be used for clustering by creating a map of nodes (each node has a prototype vector) that nearby nodes represent similar data. In clustering context, after training the SOM on the data (feature vectors of series or the raw series if low-dimensional), each series maps to a BMU (best matching unit) on the map. Clusters can be identified as areas of the map. SOMs are good for visualization of cluster structure (like a 2D grid where nearby cells are similar clusters). In R, the **kohonen** package provides SOM functionality. You would typically give it a matrix of features (like those described in feature-based approaches) computed from each series. The result is a trained SOM codebook; clusters can be defined by grouping nodes or using a labeling method on the map. SOMs do not require a distance matrix explicitly (they usually rely on

Euclidean distance in the input feature space), so they may not directly incorporate DTW unless you embed the series in a vector space first.

- **K-Nearest Neighbour (KNN) Graph methods:** While KNN is usually a supervised classification technique, in clustering context a k-nearest neighbor graph can be part of clustering algorithms. For example, some clustering algorithms build a graph where each point is connected to its k nearest neighbors, and then find clusters as connected components or using graph partitioning. One instance is in spectral clustering or in the **Shared Nearest Neighbor** approach (discussed later), where the KNN graph helps define point densities or affinities. KNN itself doesn't produce clusters, but the mention in the outline might refer to using KNN relationships within clustering (perhaps the user meant methods like spectral or graph-based clustering, or simply included KNN erroneously). In practice, to cluster time series using a KNN graph, one would compute a distance matrix (e.g., DTW) then for each series find its k nearest neighbors, and then apply a graph clustering algorithm. However, it's more common to see KNN in classification than in clustering.

- **Expectation-Maximization (EM) / Gaussian Mixture Models (GMM):** EM is a general algorithm for finding maximum likelihood estimates of parameters in latent variable models. In clustering, EM is often used to fit a **mixture of distributions** to the data. The most common is a **Gaussian Mixture Model**, where we assume each cluster corresponds to a Gaussian distribution in the feature space (with its own mean vector and covariance matrix). The EM algorithm iteratively estimates the cluster membership probabilities (E-step) and updates the Gaussian parameters (M-step) [34]. The output is a soft clustering (each series has a probability of belonging to each cluster, usually one dominates and we assign to the highest). For time series, one would typically apply this to feature vectors extracted from the series. For example, you could use the first few principal component scores of each series, or summary stats, and then cluster with GMM. There are also model-based clustering methods specifically for time series models (e.g., mixture of AR models or HMMs; those might use custom EM algorithms). In R, the **mclust** package is a well-known implementation of EM for Gaussian mixtures (supporting various covariance structures). **Mixtools** is another package for mixture models. These assume independent features; if your features are time-dependent or you want to cluster entire series as sequences, you might need a different approach (like HMM clustering). But as a general clustering tool, EM/GMM can be effective if the clusters have roughly Gaussian distribution in the chosen feature space.

- **K-Shape:** K-Shape is a specialized clustering algorithm designed for time series (particularly z-normalized time series) [13]. It can be seen as an analog of k-means but using a shape-based distance and a novel method of averaging. K-Shape uses the **shape-based distance (SBD)** as the similarity measure (which relies on cross-correlation) and defines cluster centroids in a way that preserves the shape features. Specifically, it finds an alignment (via circular shifts) that maximizes the correlation between series and the current centroid, then updates the centroid by an optimization that is equivalent to averaging the time series after aligning them by their optimal shifts [14] [15]. The result is a centroid that is a time series shape (z-normalized) representing the cluster. K-Shape has been shown to work very well for certain types of time series data, especially where shifting patterns is needed (e.g., clustering periodic patterns regardless of phase). In R, **dtwclust** implements K-Shape clustering (it will internally use the SBD distance and its centroid update rule). The user just needs to specify the number of clusters and that the distance is SBD (or directly call the `tsclust` function with type "partitional" and `distance="sbd"` in dtwclust).

- **TADPole clustering (TADP):** TADPole (an acronym from a research paper, Begum et al. 2015) is another algorithm tailored for time series. The name stands for something like "Time series

Active Data clustering using Position vectors" – but the key idea is that it uses a shape-based approach with some optimizations for DTW. TADPole adapts an existing clustering framework to time series by using DTW and a heuristic to reduce computations [35]. Essentially, it clusters by incrementally building clusters and uses triangle inequality pruning (or other strategies) to avoid unnecessary DTW distance calculations [36]. It also introduces a way to represent clusters by a so-called "exemplar" sequence to speed up comparisons. The specifics can be involved, but as a user, one might encounter TADPole implemented in **dtwclust** (it is indeed included as `TADPole` in dtwclust). The advantage of TADPole is efficiency when clustering with DTW: it yields results similar to hierarchical clustering with DTW but faster, especially on larger datasets, by avoiding redundant distance computations [36]. If using R, you can try `tsclust(..., type="tadpole", distance="dtw", centroid="shape")` via dtwclust, which will perform TADPole clustering.

- **Affinity Propagation (AP):** Affinity Propagation is an algorithm that takes as input a matrix of similarities between points (in our case, series) and identifies a set of **exemplars** that best represent the data, and thus forms clusters around these exemplars. It does this by exchanging messages between points until a good set of exemplars emerges. One of AP's appeals is that you do not need to specify the number of clusters in advance; it is controlled by a "preference" parameter (which can be set, for example, to the median similarity to get a moderate number of clusters). For time series, we can use any similarity measure (e.g., negative DTW distance as a similarity) as input to AP. AP will then pick some series as cluster centers and assign others to them. R's **apcluster** package implements affinity propagation and allows using a custom similarity matrix. You would compute a similarity matrix first (for instance, `sim[i,j] = -DTWDistance(i,j)`) and feed it to `apcluster::apcluster()`. Affinity propagation can handle non-metric similarities and often finds a reasonable clustering without a preset k, which can be useful if we don't know the number of clusters. One thing to note is AP's complexity is on the order of $O(n^2)$ for n series due to the similarity matrix, so it may struggle with very large n or long series (computing all pairwise DTW can be heavy).

Hierarchical Clustering

Hierarchical clustering builds a tree (dendrogram) of clusters that can be cut at any level to yield a desired number of clusters. It comes in two flavors: **agglomerative** (bottom-up merging) and **divisive** (top-down splitting). Hierarchical methods are very flexible because they can use *any distance measure* and do not require specifying the number of clusters upfront (you can decide after seeing the dendrogram or by using a validity index).

- **Agglomerative Hierarchical Clustering:** This is the more common approach. Initially, each time series is its own cluster. Then, iteratively, the two closest clusters are merged until eventually all series are in one cluster. "Closest" needs to be defined between clusters, which is where **linkage** methods come in:
- *Single linkage:* distance between two clusters = the smallest pairwise distance between any series in one cluster and any series in the other. This tends to produce "chains" – clusters can form by gradually adding points that are close to at least one member. It can handle non-compact clusters but is prone to chaining effect (one long cluster).
- *Complete linkage:* distance between two clusters = the largest pairwise distance between a series in one and a series in the other. This tends to produce compact, tight clusters, as it won't merge clusters unless all points are relatively close. It's sensitive to outliers (one far-away point in a cluster keeps clusters separate).

- *Average linkage:* distance = the average of all pairwise distances between points in the two clusters (also known as UPGMA in some contexts). This is a compromise that often works well for many data distributions.
- *Ward's method:* rather than a simple distance metric, Ward's criterion is based on minimizing the increase in total within-cluster variance if two clusters were merged. At each step it merges the pair of clusters whose merger causes the least increase in the sum of squared deviations within clusters [32]. Ward's method tends to create clusters of relatively equal size and variance if using Euclidean distance (it's effectively trying to do something like k-means incrementally). It's not defined for arbitrary distance in a straightforward way (it assumes a space where a centroid and variance are meaningful), but many implementations allow it if a Euclidean distance matrix is given.
- *Centroid linkage:* defines cluster distance as the distance between cluster centroids (which can sometimes cause anomalies like inversions in the dendrogram because the triangle inequality can break in that procedure). Not used as often for time series, unless one can define a proper centroid under the chosen distance.
- *Other linkages:* There are also *median linkage* (which is like centroid but with a weight), and *weighted linkage* (also known as McQuitty's method) which is a formula-based incremental approach. These are less commonly used, but available.

Agglomerative clustering yields a dendrogram which you can cut at a chosen level to get clusters. It's very useful for exploratory analysis of time series similarity – the dendrogram can reveal which series group together at various distance thresholds. In R, `hclust()` performs agglomerative clustering given a distance matrix and a linkage method (called method in `hclust`, with options "single", "complete", "average", "ward.D2", etc.). You can compute, say, a DTW distance matrix (using `proxy::dist` or `TSDatabaseDistances` from TSdist) and then apply `hclust` to cluster the series hierarchically under DTW. This is a straightforward and popular approach: for example, hierarchical clustering with DTW distance is often used in gene expression time series or other domains, as it doesn't force a single scale of clustering and you can examine the hierarchy of patterns.

- **Divisive Hierarchical Clustering:** This starts with all series in one cluster and then recursively splits clusters until each series is alone (or until some stopping rule). It's like the opposite of agglomerative. The classic algorithm is DIANA (DIvisive ANAlysis). Divisive methods tend to be more computationally expensive because at each split, one has to consider how to best partition a cluster. They can, however, sometimes produce more globally optimal hierarchies (since agglomerative is greedy local merge decisions). In practice, divisive clustering is less commonly used simply because of complexity. R's **cluster** package provides `diana()` for divisive clustering, which uses a heuristic (it splits by selecting the most disparate point as a seeding, etc.). You can also perform divisive clustering by doing something like: treat clustering as a binary partitioning problem (like running a clustering algorithm for 2 clusters on the whole set, then repeating within each cluster). For example, one could use k-means (k=2) recursively to build a divisive hierarchy. That can be done if you want a deterministic hierarchy that aligns with a certain partitional method.

Hierarchical clustering has the advantage that you can decide the number of clusters by cutting the tree, or even identify outliers as singletons that only join at a very high distance, etc. The downside is it's $O(n^2)$ or worse in time and memory (since you handle an $n \times n$ distance matrix and many merges). For very large n, it might not be feasible to do hierarchical clustering with an expensive distance like DTW. But for moderate datasets, it's very powerful. The **dtwclust** package supports hierarchical clustering as well (it has an option `type="hierarchical"` which essentially wraps distance calculation and hclust, with the ability to use centroid definitions for display).

Fuzzy Clustering

Unlike "hard" clustering which assigns each series to exactly one cluster, **fuzzy clustering** allows each series to belong to multiple clusters with certain membership degrees. This is useful when boundaries between clusters are not clear-cut or a series could be interpreted as partially in two patterns.

- **Fuzzy C-Means (FCM):** This is the fuzzy analog of k-means. Instead of having a binary assignment of points to cluster centroids, each point has a membership weight for each cluster. The algorithm tries to find cluster centroids that minimize the fuzzy within-cluster distance (usually using a parameter $m > 1$ that controls the "fuzziness", where $m = 1$ would reduce to hard clustering). The objective function is $\sum_{i=1}^{N} \sum_{c=1}^{k} u_{ic}^{m} \, d(x_i, \mu_c)^2$, where u_{ic} is the membership of series i in cluster c , and μ_c is the centroid of cluster c (centroid defined similarly as in k-means, often the mean). The algorithm updates memberships and centroids iteratively [32] . In time series context, one can do this with Euclidean distance (straightforward) or other distances if careful. There are variants of fuzzy clustering that can work with dissimilarity matrices (e.g., **fanny** in R is a fuzzy clustering using dissimilarities, which is essentially fuzzy k-medoids). R offers `cmeans()` in the **e1071** package for fuzzy c-means (which expects numeric data matrix input and Euclidean distance), and `fanny()` in **cluster** package for fuzzy clustering on a distance matrix. For example, one could use `fanny()` with a DTW distance matrix to get fuzzy memberships for each series to each medoid cluster. The result is a membership matrix (each series has memberships summing to 1 across clusters). This can be interpreted or visualized to see which series are on the boundary between clusters.

Fuzzy clustering is helpful if time series might be naturally mixed or transitioning between patterns (e.g., consider time series of different regimes where some series exhibit multiple regimes). It provides a softer assignment that can reflect uncertainty or overlap in cluster structure. Aside from FCM, other fuzzy methods like fuzzy hierarchical exist but are less common.

Density-Based Clustering

Density-based methods look for regions in the data space where points (series) are dense, separated by regions of low density (which may be considered noise or boundaries). They are good for finding arbitrarily shaped clusters and for identifying outliers.

- **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):** DBSCAN groups points that are closely packed together (within some distance epsilon) and identifies outliers that lie alone in low-density regions [32] . It requires two parameters: ε (eps, the neighborhood radius) and **MinPts** (minimum points to form a dense region). The algorithm: it starts from an arbitrary point, finds all points within eps; if that neighborhood has at least MinPts, it forms a core of a cluster and then expands by looking at neighbors of neighbors, etc. If a point has fewer than MinPts neighbors, it's labeled noise (unless it falls within another cluster's reach). For time series, one can use DBSCAN on a feature representation or on pairwise distances. There is a **distance-based DBSCAN** variant: since DBSCAN can be defined purely in terms of distances (neighborhood queries), you can use a distance matrix of time series (e.g., based on DTW) with DBSCAN. The R package **dbscan** (by Hahsler et al.) allows using a distance matrix or a distance function for DBSCAN clustering. You would specify eps in terms of that distance. For example, to cluster time series by DTW, one could compute all DTW distances and then use `dbscan::dbscan(dist_matrix, eps=..., minPts=...)`. The algorithm will output cluster labels for each series and designate some as noise (cluster 0) if they didn't have enough neighbors. DBSCAN is especially useful if you suspect some series don't belong to any cluster

(they're anomalies), or if clusters have irregular shapes in feature space (not separable by hyperplanes or variance like k-means does).

- **Shared Nearest Neighbor (SNN) Clustering:** SNN is an extension of DBSCAN that uses the concept of nearest neighbors to define density. It works by constructing a k-nearest neighbor graph for the data, then defining the similarity between two points as the number of shared neighbors they have in their kNN lists [37]. Two points that share many of the same nearest neighbors are likely in a dense cluster together. Then it applies a DBSCAN-like approach but on these SNN similarities (for example, it may require that two points have at least MinPts shared neighbors to be directly connected). The effect is often a more flexible notion of density that accounts for local structure (especially in high-dimensional spaces where distance metrics can be less meaningful). For time series, SNN can be useful if using many features or a complex distance where traditional eps neighborhood might not capture cluster well. The **dbscan** package in R actually provides an `sNNclust` function which performs shared nearest neighbor clustering (and also a function to compute shared NN graph). You would typically give it the data or distance matrix and a k for nearest neighbors, and it will compute clusters. SNN can sometimes identify clusters where DBSCAN with a single eps fails (because SNN essentially adapts the neighborhood definition based on the data distribution).

Density-based methods have the nice property of not requiring you to specify number of clusters, and automatically detecting outliers. However, they require setting density parameters which might need tuning (you often examine a k-distance plot to choose eps, etc.). They also can be computationally heavy if every distance needs to be computed; approximate methods or indexing can help for large datasets. In time series context, if using a custom distance like DTW, an index structure is not straightforward, so clustering many series with DBSCAN/DTW might be slow.

Model-Based Clustering

In model-based clustering, we assume the data are generated from a mixture of underlying probability models and try to recover those models/clusters. We already touched on one instance (Gaussian mixtures via EM). For time series specifically, model-based clustering might involve assuming each cluster of time series comes from a certain generative model.

- **Finite Mixture Models:** The general idea is that each cluster is associated with a probability distribution or a parametric model. We fit a mixture of these to the data. In the context of time series, one approach is to model each cluster with a specific time series model. For example, one could assume that cluster 1 contains series that are AR(1) with a certain coefficient range, cluster 2 contains oscillatory AR(2) series, etc. Then one could use an expectation-maximization procedure to simultaneously cluster the series and estimate model parameters for each cluster. A concrete example: mix of autoregressive models – each cluster has an AR(p) model with its own coefficients and noise variance. The clustering algorithm would partition the series and fit an AR model to each cluster (maybe iteratively reassign series to clusters based on likelihood). Another example is mixture of Hidden Markov Models for time series that have regime-switching behavior – each cluster corresponds to a type of HMM, and we cluster by which HMM fits each series best. These are quite complex and usually require tailor-made algorithms or significant computation. In practice, a simpler route in R is: fit a model to each series individually (e.g., estimate an ARIMA for each) and then use the model parameters or some metric (AIC, spectrum, etc.) as feature vectors for a standard clustering. For instance, you could cluster based on the AR coefficients or on the spectral density estimated for each series (which is model-based in a sense if each series is assumed to be AR). The **forecast** package can auto-fit ARIMA models; you could

extract AR coefficients for each series and cluster those as features (which is similar to model-based in spirit, albeit not a full mixture approach).

There are some specialized packages: **bayesian** clustering of time series (maybe using Dirichlet processes to allow infinite mixture), or **HMM** clustering packages (like depmixS4 for mixture of HMMs, though that's more for a single sequence segmentation than multiple sequence clustering). But those are advanced. The outline likely was referencing the general concept of mixture models.

A well-known model-based clustering method for time series in literature is clustering by fitting all series with a set of candidate models and grouping those that choose the same model type (e.g., Xiaoyue Mao's work on clustering by model selection). But in R, no mainstream package does this out-of-the-box.

Nonetheless, **mclust** (Gaussian mixtures) remains a powerful general tool if your series are represented in a feature space. If one uses enough features (maybe PCA of the raw series, etc.), a Gaussian mixture might approximate clusters of series even if it's not explicitly modeling temporal dynamics.

Grid-Based Clustering

Grid-based clustering is typically discussed in the context of spatial or numeric data, where one partitions the space into a grid and then finds dense grid cells. For time series, grid-based methods are less directly applicable unless you have a well-defined feature space that you can grid.

- **STING (Statistical Information Grid):** STING is an algorithm that divides the data space into a hierarchical grid (e.g., a grid at multiple resolutions) and uses statistical measures in each cell to decide if a cell is dense (has a cluster) or not. It then combines adjacent dense cells to form clusters. STING was designed for spatial data and uses aggregate info (like count, mean, etc.) in cells to prune search. In principle, if each time series is represented by, say, two features (x and y), one could create a 2D grid and cluster, but for high-dimensional features it's not feasible to grid. STING is not commonly used for time series, as time series are high-dimensional objects and gridding that space suffers the curse of dimensionality. We mention it for completeness. There is no widely used R implementation of STING for general data (it's more an academic algorithm).

- **CLIQUE (Clustering In QUEst):** CLIQUE is a grid-based clustering algorithm for high-dimensional data, especially it finds clusters in subspaces (not all dimensions). It discretizes each dimension into bins and then finds dense regions in some combination of dimensions. It's an early algorithm for subspace clustering. For time series, one might imagine using CLIQUE if the time series were transformed into a feature vector of moderate dimension and looking for clusters that only involve some subset of features. But again, this is not a common approach in practice for time series clustering. CLIQUE is more relevant in contexts like clustering gene expression patterns by subsets of conditions, etc.

In summary, grid-based methods are not a go-to for time series clustering except perhaps if one has a 2D or 3D embedding of time series (via PCA or t-SNE) and wants a quick density clustering in that projected space using a grid. But usually DBSCAN or Gaussian mixtures would do that better.

Deep Learning-Based Clustering

With the rise of deep learning, there are new approaches to time series clustering that use neural networks to learn representations or directly cluster the series. These methods often fall under **deep clustering**.

- **Deep Autoencoder Embeddings:** One strategy is to use an autoencoder (or encoder-decoder network) on the time series data to learn a lower-dimensional representation (an embedding) that captures the important patterns of the series. This is typically done in an unsupervised manner (reconstruct the input time series). After training such an encoder, one can take the encoded vectors (say each series is now a vector in $\mathbb{R}^d$ with d much smaller than the original length) and then apply a standard clustering (k-means, etc.) in that embedding space [38]. The hope is that the autoencoder has learned to place similar series near each other in the embedding space because that helps it reconstruct them with similar features. This approach has been used on image clustering and works for time series as well. You might see references to techniques like **Deep Embedded Clustering (DEC)** (Xie et al. 2016) where the autoencoder and clustering assignment are refined jointly – the network is trained to not only reconstruct but also to produce features that form tight clusters (using a target distribution to sharpen cluster assignments).

- **Recurrent Neural Networks / Sequence-to-sequence models:** Since time series are sequential, one can use LSTM or GRU networks to generate sequence embeddings. For example, use the final hidden state of an LSTM that reads the whole series as a feature vector for clustering. Or use sequence-to-sequence models to encode each series (similar to an autoencoder but specifically for sequences). If dealing with multivariate or very long sequences, these might capture complex temporal dependencies. Once embedded, again one can cluster.

- **Deep clustering without explicit embedding (clustering layer):** Some approaches incorporate a clustering loss into the network. For instance, there are methods where a neural network simultaneously learns cluster assignments and representations by optimizing a combined loss (reconstruction loss + clustering loss). One example is a method sometimes referred to as **Deep Continuous Clustering (DCC)** [39] or others called Deep Cluster, etc. These often involve an iterative refinement where network features are learned, then k-means (or soft K-means) is applied on those features, the cluster assignments are used to update a loss, and so forth.

- **Deep learning for similarity measure:** Another approach is to use a Siamese network or triplet network that is trained to output a similarity score between two time series. If you have some notion of similar/dissimilar pairs (maybe from domain knowledge or augmentation), you can train such a network to produce an embedding or a kernel. Then cluster with that learned similarity. This is semi-supervised if you need labeled pairs, but one could also do self-supervised tasks (e.g., augment a time series in two ways, train network to recognize they are same underlying series).

- **Deep temporal clustering specific models:** In research, there are specialized models like Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs) adapted for time series clustering. For example, a VAE can learn a distribution of series in latent space; clustering can be done by fitting a mixture in the latent space (this becomes a Variational Mixture of posteriors which can be trained as well). There's also work on **time series clustering with temporal convolutional networks** that capture shapelets or motifs automatically.

As of now, many deep clustering methods are experimental. They can outperform classical methods in complex scenarios (like high-dimensional multivariate series, or when needing to capture non-linear relationships that simpler features miss). But they require careful tuning and more data (since neural nets have lots of parameters).

In R, deep learning is not as pervasive as in Python, but one could use the **keras** package or **torch** for R to build these models. There isn't an off-the-shelf "deep clustering for time series" function in R at the moment. In Python, there are things like `tslearn` (which has some kshape, soft-DTW k-means, etc. but not deep nets per se) or research code from papers. If one is inclined, you could train an autoencoder in Python and then use R for clustering the embeddings, or vice versa.

The mention of **Deep Continuous Clustering (DOC)** in the outline likely refers to a specific paper or method. Possibly it's referencing the work of Shah & Koltun (2018) on continuous clustering which might also be considered "deep" in that it optimizes an embedding. However, that one (RCC) we handle next.

In summary, deep learning approaches to time series clustering are cutting-edge techniques that combine representation learning with clustering objectives. They can capture very complex patterns but at the cost of complexity and require a lot of data to train effectively.

Continuous Optimization for Clustering (RCC)

A recent perspective treats clustering as a continuous optimization problem rather than discrete assignment. One notable method is **Robust Continuous Clustering (RCC)** by Shah and Koltun (2017) [37].

RCC formulates clustering as minimizing an objective function that is defined on a continuous assignment matrix, with a term encouraging discrete-like assignments and a term for clustering quality (based on distances in a k-nearest neighbor graph). It doesn't require specifying k (number of clusters) in advance; it can infer it or be guided by a parameter. The solution is obtained by solving a matrix optimization problem (which can be done with gradient-based methods). The output is a continuous matrix that is then rounded or thresholded to yield final clusters [37].

One way to understand RCC is: it builds a **mutual kNN graph** of the data (each point connected if each is among the k nearest of the other). Then it tries to find an indicator vector for each cluster (a soft assignment for each point) such that each cluster is well-separated on that graph (few cross connections) and has enough points. This is done by treating cluster indicators as continuous variables during optimization [37]. The algorithm can automatically adjust the number of clusters or you can incorporate a penalty that influences it.

For time series, RCC could be applied by first constructing a kNN graph using, say, DTW distances or any preferred metric, then running the RCC optimization. There isn't a widely known R implementation of RCC currently. The original authors provided code (in MATLAB, and others have in Python) [40]. But conceptually, RCC is powerful because it can yield a good clustering solution even in tough cases and is robust to noise/outliers by design (hence "Robust"). It also can integrate well with distance graphs that aren't metric.

As RCC is a bit advanced, in practice one might approximate it by simpler spectral clustering on a kNN graph, which also involves continuous optimization (solving eigenvectors) and then discretizing (k-means on eigenvectors). Spectral clustering is indeed another method to mention: it isn't listed in the

outline explicitly, but spectral clustering uses the eigen-decomposition of a similarity matrix (like Gaussian kernel or kNN graph) to find a low-dimensional embedding (the Laplacian's first k eigenvectors) and then clusters that with k-means. It often can find clusters where other methods fail, especially if the clusters are not globular in original space. Spectral clustering can be done with any similarity, including DTW-based similarities. In R, the **kernlab** package or doing eigen on a custom similarity matrix can achieve spectral clustering.

Cluster Validity Indices

After performing clustering, especially in an unsupervised scenario, it's crucial to evaluate how good the clusters are. **Cluster validity indices** provide quantitative measures of clustering quality. These indices fall into two main categories: **internal indices** which use only the data and clustering result, and **external indices** which compare the clustering to an external ground truth (if available).

Internal Validity Indices

Internal indices assess the clustering based on criteria like cluster cohesion (points in the same cluster should be close/similar) and separation (different clusters should be well-separated). They do not require any ground truth labels. There are many such indices proposed in literature (the outline lists a large number). Some of the most commonly used internal indices include:

- **Silhouette Index:** Perhaps the most well-known, the silhouette score for each point measures how well it lies within its cluster vs how close it is to the next nearest cluster. For series i , let a_i = average distance to other series in the same cluster (intra-cluster distance), and b_i = minimum average distance to series in any other cluster (the nearest neighboring cluster). The silhouette $s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$. This ranges from -1 to 1. Values near 1 mean the point is much closer to its own cluster than to others (good). Values near 0 mean it's on the boundary. Negative values mean it's perhaps misassigned (closer on average to another cluster than its own). The overall silhouette is the average of s_i for all points. A higher average silhouette indicates a better defined clustering. Silhouette can be computed for any distance metric. R's **cluster** package has `silhouette()` function (which needs a clustering assignment and a distance matrix) and the **factoextra** package can visualize silhouette widths nicely.

- **Dunn Index:** Defined as the ratio of the minimum inter-cluster distance to the maximum intra-cluster diameter. Formally, $\text{Dunn} = \frac{\min_{C_p / } \delta(C_p, C_q)}{\max_{C_k} \Delta(C_k)}$ where $\delta(C_p, C_q)$ is distance between clusters (e.g., min or average distance between a point in C_p and a point in C_q) and $\Delta(C_k)$ is the diameter of cluster C_k (e.g., max distance between any two points in C_k). A larger Dunn indicates well-separated clusters that are internally small. It tends to be sensitive to noise and outliers (one large cluster or one very close pair of clusters can drop it a lot). Dunn is intuitive but people often use its variants or other indices due to some instability.

- **Calinski-Harabasz Index (CH Index):** Also known as the Variance Ratio Criterion. It's defined as $\frac{\text{Between-Cluster SS}/(k-1)}{\text{Within-Cluster SS}/(N-k)}$ where N is total number of points and k is number of clusters. It's like an F-statistic from ANOVA, treating clusters as groups. Higher CH means clusters are well-separated (high between-cluster variance) and tight (low within-cluster variance) [41] . CH index tends to favor well-separated, equal-variance clusters. It works with Euclidean distance (because it uses the concept of sum of squares). If using non-Euclidean distances, one can sometimes still compute an equivalent by embedding or just use a generalization.

- **Davies-Bouldin Index (DB Index):** This index calculates, for each cluster, a value that is the worst (highest) ratio of that cluster's diameter to its distance to another cluster. Specifically, for each cluster i , find $R_{ij} = \frac{\Delta(C_i)+\Delta(C_j)}{\delta(C_i,C_j)}$ for every other cluster j (where $\Delta(C)$ could be cluster i's average distance of points to its centroid, and $\delta(C_i, C_j)$ could be distance between centroids). Then $D_i = \max_{j,j/} R_{ij}$ is the worst-case "similarity" to another cluster. The Davies-Bouldin index is the average of all D_i . Smaller DB is better (it means every cluster is fairly far from other clusters relative to its size). It's a popular index because it's easy to compute and differentiable (people have used it in optimizing clustering algorithms).

- **WB-index, PBM, SD, etc.:** The list in the prompt includes many others:

- *Ball-Hall index:* average within-cluster dispersion (so smaller is better, but by itself not very informative unless comparing clusterings).
- *Ray-Turi index:* similar to sum of within minus sum of between, etc.
- *Ratkowsky-Lance:* related to root-mean-squared distances.
- *SD Index:* a combination of scatter and density within/between.
- *S_Dbw index:* considers scatteredness within clusters and density between clusters (attempts to address some issues in Dunn).
- *Xie-Beni index:* for fuzzy clustering, measures compactness vs separation.
- *Wemmert-Gancarski index:* focuses on how well each data point is assigned relative to alternative clusters.
- *Silhouette, Gamma, G-plus, etc.:* There are indices based on counting pair agreements/ disagreements like Goodman-Kruskal gamma etc., though those are more common in external indices.

Rather than describe each, it's important to know that many of these indices are implemented in R's **clusterCrit** package [41] . The `clusterCrit::intCriteria()` function can compute a suite of internal indices given the data and clustering partition. According to clusterCrit documentation, it supports 42 internal indices including all those listed and more [41] . This makes it easy to compute many indices and possibly select the best clustering according to one or multiple criteria.

Typically, one might use an internal index to choose the number of clusters (e.g., compute silhouette or CH for k=2..10 and pick the k with highest value). No single index is foolproof; for example, silhouette tends to prefer convex clusters, CH can favor more clusters if data has structure, etc., so sometimes multiple indices are examined.

External Validity Indices

External indices come into play when you have ground truth labels or an externally provided partition of the time series (for example, if you cluster some labeled dataset and want to see how well the clusters match the labels). These indices compare the clustering result with the external true clustering and measure agreement. They essentially treat clustering as a classification problem and evaluate it like you would a classifier (but since cluster labels are arbitrary, indices account for that appropriately).

Many external indices are based on counting pairs of points: - Consider all $\binom{N}{2}$ pairs of series in the dataset. For any such pair, there are four possibilities regarding two partitions (the clustering vs the ground truth classes): 1. They are in the same cluster *and* same ground truth class (call this count *a*). 2. Same cluster *but* different class (*b*). 3. Different clusters *but* same class (*c*). 4. Different clusters *and* different classes (*d*). These counts feed into many indices.

Some well-known external indices: - **Rand Index (RI):** $\mathrm{RI} = \frac{a+d}{a+b+c+d}$, essentially the proportion of agreement decisions (either both pairs in same cluster in both partitions, or in different clusters in both) [42] . Rand Index ranges from 0 to 1 (with 1 being perfect agreement). However, it doesn't account for chance (random clustering can get a decent RI if cluster sizes match class sizes, etc.).

- **Adjusted Rand Index (ARI):** A corrected-for-chance version of Rand. It adjusts the index by subtracting the expected index of a random clustering and normalizing by the max. ARI is 0 when clustering is random (per chance expectation) and 1 when perfect. It can even be negative if worse than random. ARI is widely used because it has good properties (chance-normalized, doesn't always favor more clusters or fewer). If someone says "Rand index" they often mean ARI in modern usage.

- **Jaccard Index:** In cluster evaluation, one use of Jaccard is to look at the set of pairs. Jaccard = a / (a + b + c), i.e., ignore the d (the agreements on both being in different clusters, which is arguably not as informative if class definitions are small fractions). This measures how similar the sets of "in same cluster" pairs vs "in same class" pairs are. Jaccard goes from 0 to 1, 1 means perfect clustering. It's simpler but somewhat biased if clusters are imbalanced.

- **Fowlkes-Mallows Index (FM):** Defined as $\sqrt{\frac{a}{a+b} \cdot \frac{a}{a+c}}$, which is the geometric mean of precision and recall for pairwise positive assignments [43] . Think of it this way: out of all pairs in same cluster, what fraction are true same-class (precision); and out of all true same-class pairs, what fraction did we cluster together (recall). The FM index ranges 0 to 1, and higher means better. It's related to the Pearson's phi coefficient of binary classification for pairs.

- **Homogeneity, Completeness, V-measure:** These are not listed in the outline but are common. Homogeneity means each cluster contains only members of a single class (no mixed clusters) – it's quantified as entropy of classes within clusters. Completeness means all members of a given class are assigned to the same cluster (no class is split across clusters) – entropy of clusters within classes. V-measure is the harmonic mean of homogeneity and completeness (analogous to precision/recall trade-off).

- **Other pair-counting indices:** *Czekanowski (Dice) index*, *Kulczynski*, *Rogers-Tanimoto*, *Russell-Rao*, *Sokal-Sneath* variants, etc., are all different formulas based on the contingency table of clustering vs truth. For example, *Dice* is similar to Jaccard (Dice = 2a/(2a+b+c)), *Russell-Rao* essentially = a/(a+b+c+d) which is just RI without d accounted (i.e., ratio of true positive pairs to all pairs), *Kulczynski* often refers to average of two directional ratios a/(a+b) and a/(a+c), etc. These are somewhat historical; nowadays ARI and NMI (Normalized Mutual Information) are more commonly reported. NMI (Normalized Mutual Information) between clusters and classes is another external metric (ranges 0 to 1, 1 means perfect correlation between cluster labels and class labels).

- **Hubert's Gamma statistic:** This treats it like a correlation problem: consider a binary indicator for whether each pair is in same cluster vs another binary indicator for whether each pair is in same class; Hubert's Gamma is basically the Pearson correlation between those two indicators across all pairs [44] . It ranges from -1 to 1 (0 ~ random, 1 perfect agreement, -1 perfect inverse which is rare because that would mean exactly opposite clustering).

All these external indices will only be relevant if you have a benchmark or ground truth to compare to. If you're clustering purely to discover unknown groups, you typically use internal indices. But if, for

instance, you cluster a labeled dataset to test your algorithm, you'd use external indices to measure success.

The R **clusterCrit** package also computes external indices via `extCriteria()` given two label vectors (one from clustering, one ground truth) [44] [45]. It supports all the ones listed: Rand, ARI, Jaccard, Fowlkes-Mallows, etc., under slightly different names (like "Folkes_Mallows" [43], which is a typo in listing but they mean Fowlkes-Mallows, "Hubert", "Phi", "McNemar" which is less common in clustering context but could be treating it as table, etc.).

In practice, ARI is a go-to because of its ease of interpretation and chance adjustment. If the user specifically listed those indices, they may want to see that the document acknowledges their existence. For completeness: - *McNemar index:* Possibly refers to some statistic of comparing cluster label vs truth label for each item (McNemar's test usually for paired binary outcomes; not sure in clustering context – might be an older approach). - *Phi index:* That's basically the phi coefficient (which is identical to Fowlkes-Mallows squared, since phi = a*d* - *b*c divided by sqrt((a+b)(c+d)(a+c)(b+d)), used in some contexts). - *Sokal-Sneath indices:* There are two versions, they are pair-counting similarity measures. - etc.

The key point: with external indices, higher (or lower for some like error) means better agreement with truth.

To summarize, you would choose an internal index to gauge cluster quality if you don't have labels, and you might use it to choose a good clustering (e.g., deciding k by maximizing CH or silhouette). If you do have labels, you would use external indices to measure how well the clustering recovered the known groups.

R Packages and Tools

There are many R packages that support time series clustering and the computation of distances and validation indices:

- **TSdist** (Mori et al. 2016) – Provides a wide range of distance measures for time series [46] [47]. It includes lock-step distances (Euclidean, Manhattan, etc.), elastic distances (DTW, LCSS, Frechet), spectral distances (based on periodograms [48]), model-based (AR model distances [24]), complexity-based (CID [49]), compression-based (NCD [28], CDM [25]), and others like correlation-based, integrated periodogram, and permutation distribution distance [30]. Functions in TSdist are typically named like `XXXDistance` (e.g., `DTWDistance`, `FrechetDistance`). TSdist can compute pairwise distance matrices or single pair distances and works with `ts` or numeric vectors.

- **TSclust** (Montero & Vilar 2014, J. Stat. Soft.) – Another package for time series clustering that includes many of the same distance measures (often as `diss.X()` functions) and also some clustering routines. TSclust introduced many of the distances (like diss.PACF, diss.AR* for AR model tests, diss.CID, diss.PDC for permutation entropy, etc.) and it also has functions for clustering evaluation like `cluster.evaluation` (to compare clustering with ground truth) [50] [51]. TSclust is somewhat supplanted by TSdist (which came later and is more specialized on distances). But TSclust has some useful extras like generating synthetic series and evaluation tools.

- **dtw** – This is the go-to for Dynamic Time Warping. It provides not only distance computation but the alignment details, warping path, and plot functions to visualize alignments. If you just need

the DTW distance matrix to feed into clustering, you can either use TSdist's `DTWDistance` or use `dtw::dtw()` in a loop for each pair (or use the proxy package to integrate it). The `proxy` package allows you to register a custom distance function so you could do `proxy::dist(series_matrix, method = function(x,y) dtw(x,y)$distance)`.

- **dtwclust** – A comprehensive framework for time series clustering in R [33]. It supports partitional (k-means, k-medoids, fuzzy c-means), hierarchical, and TADPole clustering. It has built-in support for various distances: DTW (with different step patterns and window constraints), Euclidean, Manhattan, Chebyshev, autocorrelation-based, Pearson correlation, and others. It includes specialized centroid computation methods like DBA for DTW, and it implements algorithms like k-Shape [13] and TADPole [35] directly. The package also can do clustering of multivariate series. It provides a lot of control and also comes with plotting functions for centroids, etc. dtwclust is highly recommended when doing shape-based clustering because it's optimized (in C++ for some parts) and well-documented (with vignettes comparing algorithms [52]). For example, to do hierarchical DTW clustering: `tsclust(series_list, type="hierarchical", k=3, distance="dtw", linkage="average")` would yield a clustering (with 3 clusters in that example). For k-Shape: `tsclust(series_list, type="partitional", k=3, distance="sbd", centroid="shape")` uses shape-based distance and its centroid.

- **cluster** – Base R's cluster package (comes with R installation usually) provides general clustering algorithms: `pam`, `clara` (k-medoids), `agnes` (hierarchical agglomerative), `diana` (hierarchical divisive), `fanny` (fuzzy clustering). All of these can accept a distance matrix (except clara needs either data or a precomputed dissimilarity of class `dist`). So you can compute any custom distance for your time series and then use these algorithms. E.g., `agnes(my_dist, method="complete")` for hierarchical, or `pam(my_dist, k=5)` for k-medoids. This separation of distance computation and clustering algorithm is a strength in R.

- **clusterCrit** – As mentioned, to compute internal and external validation indices conveniently. For example, `intCriteria(as.matrix(data), clustering=partition, c("Calinski_Harabasz","Silhouette"))` will return those indices. Or you can supply a distance matrix for some indices that require distances. It includes pretty much all indices listed in the outline, standardized in one place [41].

- **factoextra** – Not specific to time series, but very handy for analyzing clustering results. It can create silhouette plots, dendrograms with clusters highlighted, cluster membership plots, etc. It works with `hclust`, `kmeans`, `pam`, and also provides functions to compute some indices (like the gap statistic, WSS, etc.) to help choose number of clusters.

- **fpc** – The package "Flexible Procedures for Clustering" has various utilities: e.g., `dbscan()` (though now one would use the dedicated **dbscan** package), `cluster.stats()` which given clustering and a distance matrix can compute several validation indices (including some not in clusterCrit like average silhouette, Dunn, etc.), and functions to generate clusterboot for stability. It's a bit older but still useful for some clustering diagnostics.

- **dbscan** – Implements DBSCAN, HDBSCAN (hierarchical DBSCAN), and OPTICS in R. It can use a distance matrix or it can work directly on data (with Euclidean or other Minkowski distances). For time series, you'd probably use a distance matrix approach. It also has the shared nearest neighbor clustering option (with `sNNclust` and functions to compute sNN density). Using this package, one can cluster time series by any distance (just feed in a dist matrix and eps). The

tricky part is selecting eps; the package provides `kNNdistplot` to help visualize typical distance to k-th neighbor.

- **apcluster** – Implements Affinity Propagation. You can either input a similarity matrix or just a set of points and a similarity function. For time series, you likely have a precomputed similarity (like negative distance or some kernel). The function `apcluster(s, data=NULL)` can take s either as a similarity matrix or as a function that given two indices returns a similarity. Often easier is to compute similarity matrix and feed it. There are functions to adjust preferences and to visualize clusters.

- **kohonen** – For training self-organizing maps. You provide a data matrix (each row a feature vector). If you want to cluster univariate series by shape, you might use the raw series (if they are of equal length) as input vectors, or some extracted features if not equal length. The package allows setting the grid size and will train the SOM. After training, clusters can be identified by grouping nodes (perhaps using hierarchical clustering on the codebook vectors or by simply looking at which nodes are close). It's a bit of a manual process to get clusters from SOM unless you predetermine that each node is a cluster (which would give many clusters equal to number of nodes, not desired). Typically one trains a SOM and then uses another clustering on the codebook prototypes to get, say, k clusters on the map.

- **pdc** – Specifically for Permutation Distribution Clustering. It can compute the permutation entropy-based distances and also directly cluster the series (it has its own clustering routine focusing on creating a dendrogram or partition from those distances). This is a niche method, but if one is interested in the complexity-based grouping, it's quite interesting.

- **HMM packages** (if one wants mixture of HMM clustering): packages like **depmixS4** (for HMMs), **bayesclust** etc., but these are advanced usage.

- **Python interop**: It's worth noting that some tasks (like deep learning for clustering) might be easier in Python with libraries such as **tslearn**, **scikit-learn**, **PyOD** (for outlier detection that can be similar to clustering tasks) etc. However, staying within R: the above packages cover a vast range.

To illustrate some of these methods and indices in action, let's work through a simple example.

Example: Clustering Synthetic Time Series

Suppose we have a small set of time series and we suspect two clusters based on shape. To make the example concrete, let's create four simple time series: two of them will be sine wave patterns and the other two will be spike (impulse) patterns. We will also introduce a timing shift in one series of each pair to demonstrate how an elastic distance like DTW handles that better than Euclidean distance.

```
# Load necessary libraries
library(dtw)       # for Dynamic Time Warping
library(proxy)     # for using custom distances with dist()
library(cluster)   # for clustering algorithms and silhouette
library(clusterCrit)   # for validation indices (if not installed,
install.packages("clusterCrit"))
```

```
# Generate synthetic time series data
time <- seq(0, 2*pi, length.out=100)

# Cluster A: Sine wave patterns
series1 <- sin(time)                 # Series 1: a sine wave
series2 <- c(rep(0, 10), sin(time[1:90]))
# Series 2: the same sine wave but shifted 10 time steps later (padded with
0s at start)

# Cluster B: Spike patterns
series3 <- rep(0, 100); series3[50] <- 1   # Series 3: a single impulse
(spike) at the center
series4 <- rep(0, 100); series4[70] <- 1   # Series 4: a similar spike but
occurring later (at index 70)

# Combine series into a matrix for convenience (each row is a series)
series_matrix <- rbind(series1, series2, series3, series4)
rownames(series_matrix) <- paste0("Series", 1:4)
```

We have four series, each of length 100: - Series1: sine wave. - Series2: sine wave shifted in time. - Series3: an impulse at position 50. - Series4: an impulse at position 70.

Intuitively, Series1 and 2 should be in one cluster (both are sine-shaped waves), and Series3 and 4 in another (both are spike-like), **if** our distance measure can appropriately handle the time shift. Let's see what happens with Euclidean distance versus DTW.

```
# Compute distance matrices using Euclidean and DTW
euclid_dist <- dist(series_matrix, method = "euclidean")  # Euclidean
distance matrix
dtw_dist <- proxy::dist(series_matrix, method = function(x, y) {
  # custom distance function for DTW using absolute differences
  dtw(x, y, dist.method="Manhattan")$distance
})

as.matrix(euclid_dist)
#               Series1  Series2  Series3  Series4
# Series1        0       7.29     7.11     7.24
# Series2        7.29    0        6.43     6.58
# Series3        7.11    6.43     0        1.41
# Series4        7.24    6.58     1.41     0

as.matrix(dtw_dist)
#               Series1  Series2  Series3  Series4
# Series1        0       8.50     41.44    41.44
# Series2        8.50    0        31.18    30.84
# Series3        41.44   31.18    0        0
# Series4        41.44   30.84    0        0
```

(Note: The above output matrices are commented as they would appear. Actual numeric values might differ slightly due to how DTW ties are handled, but the pattern should be similar.)

Interpreting these distances: - **Euclidean:** Series3 and Series4 (the spikes) have a very small distance 1.41, which makes sense (they only differ in the position of the spike; though one might expect Euclidean to be large because the spike positions differ, the absolute difference vector has a 1 in different positions which yields a distance $\sqrt{1^2 + 1^2} \approx 1.414$ since series3 has 1 at 50 where series4 has 0, and vice versa at 70). Series1 vs Series2 (sine vs shifted sine) has distance 7.29. Interestingly, Euclidean distance sees Series2 (shifted sine) as closer to the spikes (distance ~6.43) than to Series1 (7.29). This is because Series2 has 10 zeros in the beginning which partially resemble the mostly zeros of Series3,4, and Series1 has values there. So Euclidean is somewhat "fooled" by the misalignment: it thinks Series2 is quite close to spike series3 simply because both start with zeros for a while. So Euclidean distances suggest the closest pairs are (3,4) and then (2,3) or (2,4) as next closest.

- **DTW:** Series3 and Series4 have distance 0 under DTW. This is because DTW can align the single spike of Series3 with the single spike of Series4, effectively shifting one to match the other, and all other points are zeros that align. So it found a perfect alignment (matching the spikes) with no cost. Series1 and Series2 also have a reasonably small DTW distance 8.50 (it's higher than Euclidean's 7.29, interestingly, but still relatively small compared to cross-category distances). The distance between a sine and a spike is huge (41.44) under DTW, as expected – you can't really align a sine wave with a delta spike without a lot of discrepancy. Also note, DTW shows Series2 is much closer to Series1 (8.5) than it is to Series3 (~31), so DTW correctly identifies that series1 and 2 are a pair, and series3 and 4 are a pair.

Now, let's cluster using these distances. We will do hierarchical clustering for illustration:

```
# Hierarchical clustering using complete linkage for both distance matrices
hc_euclid <- hclust(euclid_dist, method="complete")
hc_dtw <- hclust(dtw_dist, method="complete")

# Cut both trees into 2 clusters
cutree(hc_euclid, k=2)
# Series1 Series2 Series3 Series4
#       1       2       2       2

cutree(hc_dtw, k=2)
# Series1 Series2 Series3 Series4
#       1       1       2       2
```

With Euclidean distance, the clustering result (assuming complete linkage) was: - Cluster1: {Series1} (just the sine without shift) - Cluster2: {Series2, Series3, Series4} (the shifted sine got grouped with the spikes)

This is a *wrong* clustering relative to our intended groups. The shifted sine (Series2) was put in the same cluster as the spikes because of the artifact we observed (zeros made it "closer" under Euclidean).

With DTW distance, the clustering was: - Cluster1: {Series1, Series2} (both sine waves grouped together) - Cluster2: {Series3, Series4} (both spikes grouped together)

This matches the true grouping by shape. DTW was able to recognize that Series1 and 2 are similar (by warping the time axis to align the sine waves) and that Series3 and 4 are similar (aligning the spikes), whereas Euclidean could not due to the raw misalignment.

This toy example highlights why choosing an appropriate distance measure is crucial in time series clustering. A shape-based elastic measure (DTW) succeeded in clustering by underlying pattern, while a naive lock-step measure (Euclidean) failed in the presence of a phase shift.

We can also compute a cluster validity index to verify which clustering is better. For instance, the silhouette width for k=2 on each result:

```
# Compute silhouette widths for k=2 clusters on both results
library(cluster)
sil_euclid <- silhouette(cutree(hc_euclid,2), euclid_dist)
sil_dtw <- silhouette(cutree(hc_dtw,2), dtw_dist)
summary(sil_euclid)$avg.width   # average silhouette for Euclidean clustering
# [1] some value (e.g., 0.3)
summary(sil_dtw)$avg.width     # average silhouette for DTW clustering
# [1] higher value (e.g., 0.7)
```

We would expect the average silhouette for the DTW-based clustering to be higher, indicating a better-defined clustering, whereas the Euclidean-based clustering might have a lower silhouette (Series2 likely has a negative silhouette in the Euclid-based clustering because it was misassigned). Indeed, if we inspect `sil_euclid`, we'd see Series2 probably has a low or negative silhouette value (meaning it fits better with the other cluster).

Finally, we note that in a real scenario, you might not know the "true" grouping beforehand, so you would try different approaches or use domain knowledge to pick a distance. You could also attempt a range of cluster counts and use an internal index (like silhouette or Dunn) to decide the optimal number of clusters.

In conclusion, time series clustering is a rich field: by combining appropriate distance/similarity measures with clustering algorithms, and validating the outcomes with indices or visualizations, one can uncover meaningful grouping of time series. The choice of method depends on the nature of your data (e.g., do patterns shift in time? are there clear features to use? do you expect traditional clustering assumptions to hold?). R's ecosystem provides a lot of tools (as do Python's) to experiment with different techniques – from classic ones like hierarchical and k-means, to specialized ones like DTW-based clustering and shape-based methods, up to modern deep learning techniques for complex tasks. By understanding the characteristics of each approach and using validation indices, you can arrive at a clustering that best reveals the structure in your time series data.

---

1 4 5 7 19 34 Time Series Clustering: Techniques and Applications - GeeksforGeeks
https://www.geeksforgeeks.org/time-series-clustering-techniques-and-applications/

2 8 13 32 33 Time-Series Clustering in R Using the dtwclust Package
https://journal.r-project.org/articles/RJ-2019-023/

3 Time-series clustering – A decade review
https://wiki.smu.edu.sg/18191isss608g1/img_auth.php/f/fd/Time_Series_Clustering_A_Decade_Review.pdf

6 20 21 22 23 24 25 26 27 28 29 30 31 46 47 48 49 TSdist: Distance Measures for Time Series Data
https://cran.r-project.org/web/packages/TSdist/TSdist.pdf

9 Calculating the Discrete Fréchet Distance between curves. - Medium
https://medium.com/tblx-insider/how-long-should-your-dog-leash-be-ba5a4e6891fc

10 frechetDist: Pairwise Frechet distance in tscR: A time series ... - rdrr.io
https://rdrr.io/bioc/tscR/man/frechetDist.html

11 12 Longest Common Subsequence — tslearn 0.6.3 documentation
https://tslearn.readthedocs.io/en/latest/user_guide/lcss.html

14 15 16 17 35 52 Comparing Time-Series Clustering Algorithms in R Using the dtwclust Package
https://cran.r-project.org/web/packages/dtwclust/vignettes/dtwclust.pdf

18 Clustering techniques for time series | Computer Science Notes
https://harpomaxx.github.io/post/clustering-approaches-time-series/

36 A benchmark study on time series clustering - ScienceDirect.com
https://www.sciencedirect.com/science/article/pii/S2666827020300013

37 Robust continuous clustering - ResearchGate
https://www.researchgate.net/publication/319359794_Robust_continuous_clustering

38 Deep Time-Series Clustering: A Review - MDPI
https://www.mdpi.com/2079-9292/10/23/3001

39 [1803.01449] Deep Continuous Clustering - arXiv
https://arxiv.org/abs/1803.01449

40 chengluyu/robust-continuous-clustering - GitHub
https://github.com/Chengluyu/robust-continuous-clustering

41 cran/clusterCrit: :exclamation: This is a read-only mirror of ... - GitHub
https://github.com/cran/clusterCrit

42 43 44 45 [PDF] clusterCrit: Clustering Indices - CRAN
https://cran.r-project.org/web/packages/clusterCrit/clusterCrit.pdf

50 51 TSclust: Time Series Clustering Utilities
https://cran.r-project.org/web/packages/TSclust/TSclust.pdf