Package: dtwclust (via r-universe)

October 29, 2026

Type Package

Title Time Series Clustering Along with Optimizations for the Dynamic Time Warping Distance

Description Time series clustering along with optimized techniques related to the Dynamic Time Warping distance and its corresponding lower bounds. Implementations of partitional, hierarchical, fuzzy, k-Shape and TADPole clustering are available. Functionality can be easily extended with custom distance measures and centroid definitions. Implementations of DTW barycenter averaging, a distance based on global alignment kernels, and the soft-DTW distance and centroid routines are also provided. All included distance functions have custom loops optimized for the calculation of cross-distance matrices, including parallelization support. Several cluster validity indices are included.

Version 6.0.0

Depends R (>= 3.3.0), methods, proxy (>= 0.4-16), dtw

Imports parallel, stats, utils, clue, cluster, dplyr, flexclust, foreach, ggplot2, ggrepel, rlang, Matrix (>= 1.5-0), RSpectra, Rcpp, RcppParallel (>= 4.4.0), reshape2, shiny, shinyjs

LinkingTo Rcpp, RcppArmadillo, RcppParallel, RcppThread

Suggests doParallel, iterators, knitr, rmarkdown, testthat

Date 2024-07-20

Author Alexis Sarda-Espinosa

Maintainer Alexis Sarda <alexis.sarda@gmail.com>

BugReports https://github.com/asardaes/dtwclust/issues

License GPL-3

URL https://github.com/asardaes/dtwclust

NeedsCompilation yes

SystemRequirements GNU make

Encoding UTF-8

LazyData TRUE

RoxygenNote 7.3.1

VignetteBuilder knitr

Config/testthat/edition 3

Collate 'CENTROIDS-dba.R' 'CENTROIDS-pam.R' 'CENTROIDS-sdtw-cent.R'
'CENTROIDS-shape-extraction.R' 'CLUSTERING-all-cent2.R'
'CLUSTERING-compare-clusterings.R'
'CLUSTERING-cvi-evaluators.R' 'CLUSTERING-ddist2.R'
'CLUSTERING-partitional-fuzzy.R'
'CLUSTERING-repeat-clustering.R' 'CLUSTERING-tadpole.R'
'CLUSTERING-tsclust-controls.R' 'CLUSTERING-tsclust.R'
'DISTANCES-dtw-basic.R' 'DISTANCES-dtw-lb.R' 'DISTANCES-dtw2.R'
'DISTANCES-gak.R' 'DISTANCES-lb-improved.R'
'DISTANCES-lb-keogh.R' 'DISTANCES-sbd.R' 'DISTANCES-sdtw.R'
'GENERICS-cvi.R' 'RD-helpers.R' 'S4-Distmat.R'
'S4-DistmatLowerTriangular.R' 'S4-PairTracker.R'
'S4-SparseDistmat.R' 'S4-tsclustFamily.R'
'S4-TSClusters-classes.R' 'S4-TSClusters-methods.R'
'SHINY-interactive-clustering.R' 'SHINY-ssdtwclust.R'
'SHINY-utils.R' 'UTILS-as-methods.R' 'UTILS-compute-envelope.R'
'UTILS-data.R' 'UTILS-expressions.R' 'UTILS-globals-internal.R'
'UTILS-nccc.R' 'UTILS-reinterpolate.R' 'UTILS-rng.R'
'UTILS-tslist.R' 'UTILS-utils.R' 'UTILS-zscore.R' 'pkg.R'

Config/pak/sysreqs make

Repository https://cran.r-universe.dev

Date/Publication 2024-07-24 02:45:59 UTC

RemoteUrl https://github.com/cran/dtwclust

RemoteRef HEAD

RemoteSha 89d34105e1d3f2e4fca2ef9ed22c4fd1fd06cacc

Contents

dtwclust-package 3
as.matrix 5
compare_clusterings 5
compare_clusterings_configs 11
compute_envelope 13
cvi 14
cvi_evaluators 17
DBA 18
DistmatLowerTriangular-class 21
dtw_basic 22
dtw_lb 26

dtw2 . . . 29
dtwclustTimings . . . 30
GAK . . . 30
interactive_clustering . . . 33
lb_improved . . . 34
lb_keogh . . . 37
NCCc . . . 39
pam_cent . . . 40
pdc_configs . . . 41
reinterpolate . . . 42
repeat_clustering . . . 43
SBD . . . 44
sdtw . . . 46
sdtw_cent . . . 47
shape_extraction . . . 48
ssdtwclust . . . 50
TADPole . . . 52
tsclust . . . 54
tsclust-controls . . . 62
TSClusters-class . . . 65
tsclusters-methods . . . 67
tsclustFamily-class . . . 71
tslist . . . 73
uciCT . . . 74
zscore . . . 75

Index **76**

`dtwclust-package` *Time series clustering along with optimizations for the Dynamic Time Warping distance*

Description

Time series clustering with a wide variety of strategies and a series of optimizations specific to the Dynamic Time Warping (DTW) distance and its corresponding lower bounds (LBs).

Details

Many of the algorithms implemented in this package are specifically tailored to DTW, hence its name. However, the main clustering function is flexible so that one can test many different clustering approaches, using either the time series directly, or by applying suitable transformations and then clustering in the resulting space. Other implementations included in the package provide some alternatives to DTW.

DTW is a dynamic programming algorithm that tries to find the optimum warping path between two series. Over the years, several variations have appeared in order to make the procedure faster or more efficient. Please refer to the included references for more information, especially Giorgino (2009), which is a good practical introduction.

Most optimizations require equal dimensionality, which means time series should have equal length. DTW itself does not require this, but it is relatively expensive to compute. Other distance definitions may be used, or series could be reinterpolated to a matching length (Ratanamahatana and Keogh 2004).

The main clustering function and entry point for this package is `tsclust()`, with a convenience wrapper for multiple tests in `compare_clusterings()`, and a shiny app in `interactive_clustering()`. There is another less-general-purpose shiny app in `ssdtwclust()`.

Please note the random number generator is set to L'Ecuyer-CMRG when **dtwclust** is attached in an attempt to preserve reproducibility. You are free to change this afterwards if you wish (see `base::RNGkind()`), but **dtwclust** will always use L'Ecuyer-CMRG internally.

For more information, please read the included package vignettes, which can be accessed by typing `browseVignettes("dtwclust")`.

Note

This software package was developed independently of any organization or institution that is or has been associated with the author.

This package can be used without attaching it with `base::library()` with some caveats:

- The **methods** package must be attached. R usually does this automatically, but `utils::Rscript()` only does so in R versions 3.5.0 and above.
- If you want to use the **proxy** version of `dtw::dtw()` (e.g. for clustering), you have to attach the **dtw** package manually.

Be careful with reproducibility, R's random number generator is only changed session-wide if **dtwclust** is attached.

Author(s)

Alexis Sarda-Espinosa

References

Please refer to the package's vignette's references.

See Also

`tsclust()`, `compare_clusterings()`, `interactive_clustering()`, `ssdtwclust()`, `dtw_basic()`, `proxy::dist()`.

as.matrix *as.matrix*

Description

proxy exported a non-generic `as.matrix` function. This is to re-export the base version and add coercion methods for `pairdist` and `crossdist`.

Usage

```
as.matrix(x, ...)
```

Arguments

x, ...	See `base::as.matrix()`.

See Also

`base::as.matrix()`

compare_clusterings *Compare different clustering configurations*

Description

Compare many different clustering algorithms with support for parallelization.

Usage

```
compare_clusterings(
  series = NULL,
  types = c("p", "h", "f", "t"),
  configs = compare_clusterings_configs(types),
  seed = NULL,
  trace = FALSE,
  ...,
  score.clus = function(...) stop("No scoring"),
  pick.clus = function(...) stop("No picking"),
  shuffle.configs = FALSE,
  return.objects = FALSE,
  packages = character(0L),
  .errorhandling = "stop"
)
```

Arguments

series	A list of series, a numeric matrix or a data frame. Matrices and data frames are coerced to a list row-wise (see tslist()).
types	Clustering types. It must be any combination of (possibly abbreviated): "partitional", "hierarchical", "fuzzy", "tadpole."
configs	The list of data frames with the desired configurations to run. See pdc_configs() and compare_clusterings_configs().
seed	Seed for random reproducibility.
trace	Logical indicating that more output should be printed to screen.
...	Further arguments for tsclust(), score.clus or pick.clus.
score.clus	A function that gets the list of results (and ...) and scores each one. It may also be a named list of functions, one for each type of clustering. See Scoring section.
pick.clus	A function to pick the best result. See Picking section.
shuffle.configs	Randomly shuffle the order of configs, which can be useful to balance load when using parallel computation.
return.objects	Logical indicating whether the objects returned by tsclust() should be given in the result.
packages	A character vector with the names of any packages needed for any functions used (distance, centroid, preprocessing, etc.). The name "dtwclust" is added automatically. Relevant for parallel computation.
.errorhandling	This will be passed to foreach::foreach(). See Parallel section below.

Details

This function calls tsclust() with different configurations and evaluates the results with the provided functions. Parallel support is included. See the examples.

Parameters specified in configs whose values are NA will be ignored automatically.

The scoring and picking functions are for convenience, if they are not specified, the scores and pick elements of the result will be NULL.

See repeat_clustering() for when return.objects = FALSE.

Value

A list with:

- results: A list of data frames with the flattened configs and the corresponding scores returned by score.clus.
- scores: The scores given by score.clus.
- pick: The object returned by pick.clus.
- proc_time: The measured execution time, using base::proc.time().
- seeds: A list of lists with the random seeds computed for each configuration.

The cluster objects are also returned if return.objects = TRUE.

Parallel computation

The configurations for each clustering type can be evaluated in parallel (multi-processing) with the **foreach** package. A parallel backend can be registered, e.g., with **doParallel**.

If the `.errorhandling` parameter is changed to "pass" and a custom `score.clus` function is used, said function should be able to deal with possible error objects.

If it is changed to "remove", it might not be possible to attach the scores to the results data frame, or it may be inconsistent. Additionally, if `return.objects` is `TRUE`, the names given to the objects might also be inconsistent.

Parallelization can incur a lot of deep copies of data when returning the cluster objects, since each one will contain a copy of `datalist`. If you want to avoid this, consider specifying `score.clus` and setting `return.objects` to `FALSE`, and then using `repeat_clustering()`.

Scoring

The clustering results are organized in a *list of lists* in the following way (where only applicable `types` exist; first-level list names in bold):

- **partitional** - list with
  - Clustering results from first partitional config
  - etc.
- **hierarchical** - list with
  - Clustering results from first hierarchical config
  - etc.
- **fuzzy** - list with
  - Clustering results from first fuzzy config
  - etc.
- **tadpole** - list with
  - Clustering results from first tadpole config
  - etc.

If `score.clus` is a function, it will be applied to the available partitional, hierarchical, fuzzy and/or tadpole results via:

```
scores <- lapply(list_of_lists, score.clus, ...)
```

Otherwise, `score.clus` should be a list of functions with the same names as the list above, so that `score.clus$partitional` is used to score `list_of_lists$partitional` and so on (via `base::Map()`).

Therefore, the scores returned shall always be a list of lists with first-level names as above.

Picking

If `return.objects` is `TRUE`, the results' data frames and the list of TSClusters objects are given to `pick.clus` as first and second arguments respectively, followed by `....` Otherwise, `pick.clus` will receive only the data frames and the contents of `...` (since the objects will not be returned by the preceding step).

Limitations

Note that the configurations returned by the helper functions assign special names to preprocessing/distance/centroid arguments, and these names are used internally to recognize them.

If some of these arguments are more complex (e.g. matrices) and should *not* be expanded, consider passing them directly via the ellipsis (`...`) instead of using `pdc_configs()`. This assumes that said arguments can be passed to all functions without affecting their results.

The distance matrices (if calculated) are not re-used across configurations. Given the way the configurations are created, this shouldn't matter, because clusterings with arguments that can use the same distance matrix are already grouped together by `compare_clusterings_configs()` and `pdc_configs()`.

Author(s)

Alexis Sarda-Espinosa

See Also

`compare_clusterings_configs()`, `tsclust()`

Examples

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

# Define overall configuration
cfgs <- compare_clusterings_configs(
    types = c("p", "h", "f", "t"),
    k = 19L:20L,
    controls = list(
        partitional = partitional_control(
            iter.max = 30L,
            nrep = 1L
        ),
        hierarchical = hierarchical_control(
            method = "all"
        ),
        fuzzy = fuzzy_control(
            # notice the vector
            fuzziness = c(2, 2.5),
            iter.max = 30L
        ),
        tadpole = tadpole_control(
            # notice the vectors
            dc = c(1.5, 2),
            window.size = 19L:20L
        )
```

```
    ),
    preprocs = pdc_configs(
        type = "preproc",
        # shared
        none = list(),
        zscore = list(center = c(FALSE)),
        # only for fuzzy
        fuzzy = list(
            acf_fun = list()
        ),
        # only for tadpole
        tadpole = list(
            reinterpolate = list(new.length = 205L)
        ),
        # specify which should consider the shared ones
        share.config = c("p", "h")
    ),
    distances = pdc_configs(
        type = "distance",
        sbd = list(),
        fuzzy = list(
            L2 = list()
        ),
        share.config = c("p", "h")
    ),
    centroids = pdc_configs(
        type = "centroid",
        partitional = list(
            pam = list()
        ),
        # special name 'default'
        hierarchical = list(
            default = list()
        ),
        fuzzy = list(
            fcmdd = list()
        ),
        tadpole = list(
            default = list(),
            shape_extraction = list(znorm = TRUE)
        )
    )
)

# Number of configurations is returned as attribute
num_configs <- sapply(cfgs, attr, which = "num.configs")
cat("\nTotal number of configurations without considering optimizations:",
    sum(num_configs),
    "\n\n")

# Define evaluation functions based on CVI: Variation of Information (only crisp partition)
vi_evaluators <- cvi_evaluators("VI", ground.truth = CharTrajLabels)
score_fun <- vi_evaluators$score
```

```
pick_fun <- vi_evaluators$pick

# ============================================================================
# Short run with only fuzzy clustering
# ============================================================================

comparison_short <- compare_clusterings(CharTraj, types = c("f"), configs = cfgs,
                                        seed = 293L, trace = TRUE,
                                        score.clus = score_fun, pick.clus = pick_fun,
                                        return.objects = TRUE)

## Not run:
# ============================================================================
# Parallel run with all comparisons
# ============================================================================

require(doParallel)
registerDoParallel(cl <- makeCluster(detectCores()))

comparison_long <- compare_clusterings(CharTraj, types = c("p", "h", "f", "t"),
                                       configs = cfgs,
                                       seed = 293L, trace = TRUE,
                                       score.clus = score_fun,
                                       pick.clus = pick_fun,
                                       return.objects = TRUE)

# Using all external CVIs and majority vote
external_evaluators <- cvi_evaluators("external", ground.truth = CharTrajLabels)
score_external <- external_evaluators$score
pick_majority <- external_evaluators$pick

comparison_majority <- compare_clusterings(CharTraj, types = c("p", "h", "f", "t"),
                                           configs = cfgs,
                                           seed = 84L, trace = TRUE,
                                           score.clus = score_external,
                                           pick.clus = pick_majority,
                                           return.objects = TRUE)

# best results
plot(comparison_majority$pick$object)
print(comparison_majority$pick$config)

stopCluster(cl); registerDoSEQ()

# ============================================================================
# A run with only partitional clusterings
# ============================================================================

p_cfgs <- compare_clusterings_configs(
    types = "p", k = 19L:21L,
    controls = list(
        partitional = partitional_control(
            iter.max = 20L,
```

```
            nrep = 8L
        )
    ),
    preprocs = pdc_configs(
        "preproc",
        none = list(),
        zscore = list(center = c(FALSE, TRUE))
    ),
    distances = pdc_configs(
        "distance",
        sbd = list(),
        dtw_basic = list(window.size = 19L:20L,
                         norm = c("L1", "L2")),
        gak = list(window.size = 19L:20L,
                   sigma = 100)
    ),
    centroids = pdc_configs(
        "centroid",
        partitional = list(
            pam = list(),
            shape = list()
        )
    )
)

# Remove redundant (shape centroid always uses zscore preprocessing)
id_redundant <- p_cfgs$partitional$preproc == "none" &
    p_cfgs$partitional$centroid == "shape"
p_cfgs$partitional <- p_cfgs$partitional[!id_redundant, ]

# LONG! 30 minutes or so, sequentially
comparison_partitional <- compare_clusterings(CharTraj, types = "p",
                                              configs = p_cfgs,
                                              seed = 32903L, trace = TRUE,
                                              score.clus = score_fun,
                                              pick.clus = pick_fun,
                                              shuffle.configs = TRUE,
                                              return.objects = TRUE)

## End(Not run)
```

`compare_clusterings_configs` *Create clustering configurations.*

Description

Create configurations for `compare_clusterings()`

Usage

```
compare_clusterings_configs(
  types = c("p", "h", "f"),
  k = 2L,
  controls = NULL,
  preprocs = pdc_configs("preproc", none = list()),
  distances = pdc_configs("distance", dtw_basic = list()),
  centroids = pdc_configs("centroid", default = list()),
  no.expand = character(0L)
)
```

Arguments

types	Clustering types. It must be any combination of (possibly abbreviated): partitional, hierarchical, fuzzy, tadpole.
k	A numeric vector with one or more elements specifying the number of clusters to test.
controls	A named list of tsclust-controls. NULL means defaults. See details.
preprocs	Preprocessing configurations. See details.
distances	Distance configurations. See details.
centroids	Centroid configurations. See details.
no.expand	A character vector indicating parameters that should *not* be expanded between pdc_configs() configurations. See examples.

Details

Preprocessing, distance and centroid configurations are specified with the helper function pdc_configs(), refer to the examples in compare_clusterings() to see how this is used.

The controls list may be specified with the usual tsclust-controls functions. The names of the list must correspond to "partitional", "hierarchical", "fuzzy" or "tadpole" clustering. Again, please refer to the examples in compare_clusterings().

Value

A list for each clustering type, each of which includes a data frame with the computed and merged configurations. Each data frame has an extra attribute num.configs specifying the number of configurations.

Examples

```
# compare this with leaving no.expand empty
compare_clusterings_configs(
   distances = pdc_configs("d", dtw_basic = list(window.size = 1L:2L, norm = c("L1", "L2"))),
    centroids = pdc_configs("c", dba = list(window.size = 1L:2L, norm = c("L1", "L2"))),
    no.expand = c("window.size", "norm")
)
```

`compute_envelope` *Time series warping envelopes*

Description

This function computes the envelopes for DTW lower bound calculations with a Sakoe-Chiba band for a given univariate time series using the streaming algorithm proposed by Lemire (2009).

Usage

```
compute_envelope(x, window.size, error.check = TRUE)
```

Arguments

`x`	A univariate time series.
`window.size`	Window size for envelope calculation. See details.
`error.check`	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

The windowing constraint uses a centered window. The calculations expect a value in `window.size` that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, `window.size = 10`, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in `10(2) + 1 = 21` observations falling within the window.

Value

A list with two elements (lower and upper envelopes respectively): `lower` and `upper`.

Note

This envelope is calculated assuming a Sakoe-Chiba constraint for DTW.

References

Lemire D (2009). "Faster retrieval with a two-pass dynamic-time-warping lower bound ." *Pattern Recognition*, **42**(9), pp. 2169 - 2180. ISSN 0031-3203, doi:10.1016/j.patcog.2008.11.030, https://www.sciencedirect.com/science/article/pii/S0031320308004925.

Examples

```
data(uciCT)

H <- compute_envelope(CharTraj[[1L]], 18L)

matplot(do.call(cbind, H), type = "l", col = 2:3)
lines(CharTraj[[1L]])
```

---

cvi *Cluster validity indices*

Description

Compute different cluster validity indices (CVIs) of a given cluster partition, using the clustering distance measure and centroid function if applicable.

Usage

```
cvi(a, b = NULL, type = "valid", ..., log.base = 10)

## S4 method for signature 'matrix'
cvi(a, b = NULL, type = "valid", ..., log.base = 10)

## S4 method for signature 'PartitionalTSClusters'
cvi(a, b = NULL, type = "valid", ..., log.base = 10)

## S4 method for signature 'HierarchicalTSClusters'
cvi(a, b = NULL, type = "valid", ..., log.base = 10)

## S4 method for signature 'FuzzyTSClusters'
cvi(a, b = NULL, type = "valid", ..., log.base = 10)
```

Arguments

a	An object returned by tsclust(), for crisp partitions a vector that can be coerced to integers which indicate the cluster memberships, or the membership matrix for soft clustering.
b	If needed, a vector that can be coerced to integers which indicate the cluster memberships. The ground truth (if known) should be provided here.
type	Character vector indicating which indices are to be computed. See supported values below.
...	Arguments to pass to and from other methods.
log.base	Base of the logarithm to be used in the calculation of VI (see details).

Details

Clustering is commonly considered to be an unsupervised procedure, so evaluating its performance can be rather subjective. However, a great amount of effort has been invested in trying to standardize cluster evaluation metrics by using cluster validity indices (CVIs).

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

Note that even though a fuzzy partition can be changed into a crisp one, making it compatible with many of the existing crisp CVIs, there are also fuzzy CVIs tailored specifically to fuzzy clustering, and these may be more suitable in those situations. Fuzzy partitions usually have no ground truth associated with them, but there are exceptions depending on the task's goal.

Each index defines their range of values and whether they are to be minimized or maximized. In many cases, these CVIs can be used to evaluate the result of a clustering algorithm regardless of how the clustering works internally, or how the partition came to be.

Knowing which CVI will work best cannot be determined a priori, so they should be tested for each specific application. Usually, many CVIs are utilized and compared to each other, maybe using a majority vote to decide on a final result. Furthermore, it should be noted that many CVIs perform additional distance calculations when being computed, which can be very considerable if using DTW or GAK.

Value

The chosen CVIs.

External CVIs

- Crisp partitions (the first 4 are calculated via `flexclust::comPart()`)
  - `"RI"`: Rand Index (to be maximized).
  - `"ARI"`: Adjusted Rand Index (to be maximized).
  - `"J"`: Jaccard Index (to be maximized).
  - `"FM"`: Fowlkes-Mallows (to be maximized).
  - `"VI"`: Variation of Information (Meila (2003); to be minimized).
- Fuzzy partitions (based on Lei et al. (2017))
  - `"RI"`: Soft Rand Index (to be maximized).
  - `"ARI"`: Soft Adjusted Rand Index (to be maximized).
  - `"VI"`: Soft Variation of Information (to be minimized).
  - `"NMIM"`: Soft Normalized Mutual Information based on Max entropy (to be maximized).

Internal CVIs

The indices marked with an exclamation mark (!) calculate (or re-use if already available) the whole distance matrix between the series in the data. If you were trying to avoid this in the first place, then these CVIs might not be suitable for your application.

The indices marked with a question mark (?) depend on the extracted centroids, so bear that in mind if a hierarchical procedure was used and/or the centroid function has associated randomness (such as `shape_extraction()` with series of different length).

The indices marked with a tilde (~) require the calculation of a global centroid. Since `DBA()` and `shape_extraction()` (for series of different length) have some randomness associated, these indices might not be appropriate for those centroids.

- Crisp partitions
  - `"Sil"` (!): Silhouette index (Rousseeuw (1987); to be maximized).
  - `"D"` (!): Dunn index (Arbelaitz et al. (2013); to be maximized).
  - `"COP"` (!): COP index (Arbelaitz et al. (2013); to be minimized).
  - `"DB"` (?): Davies-Bouldin index (Arbelaitz et al. (2013); to be minimized).
  - `"DBstar"` (?): Modified Davies-Bouldin index (DB*) (Kim and Ramakrishna (2005); to be minimized).
  - `"CH"` (~): Calinski-Harabasz index (Arbelaitz et al. (2013); to be maximized).
  - `"SF"` (~): Score Function (Saitta et al. (2007); to be maximized; see notes).
- Fuzzy partitions (using the nomenclature from Wang and Zhang (2007))
  - `"MPC"`: to be maximized.
  - `"K"` (~): to be minimized.
  - `"T"`: to be minimized.
  - `"SC"` (~): to be maximized.
  - `"PBMF"` (~): to be maximized (see notes).

Additionally

- `"valid"`: Returns all valid indices depending on the type of `a` and whether `b` was provided or not.
- `"internal"`: Returns all internal CVIs. Only supported for TSClusters objects.
- `"external"`: Returns all external CVIs. Requires `b` to be provided.

Note

In the original definition of many internal and fuzzy CVIs, the Euclidean distance and a mean centroid was used. **The implementations here change this, making use of whatever distance/centroid was chosen during clustering**. However, some of the CVIs assume that the distances are symmetric, since cross-distance matrices are calculated and only the upper/lower triangulars are considered. A warning will be given if the matrices are not symmetric and the CVI assumes so.

Because of the above, calculating CVIs for clusterings made with `TADPole()` is a special case. Since TADPole uses 3 distances during its execution (DTW, LB_Keogh and Euclidean), it is not obvious which one should be used for the calculation of CVIs. Nevertheless, `dtw_basic()` is used by default.

The formula for the SF index in Saitta et al. (2007) does not correspond to the one in Arbelaitz et al. (2013). The one specified in the former is used here.

The formulas for the Silhouette index are not entirely correct in Arbelaitz et al. (2013), refer to Rousseeuw (1987) for the correct ones.

The formulas for the PBMF index are not entirely unambiguous in the literature, the ones given in Lin (2013) are used here.

References

Arbelaitz, O., Gurrutxaga, I., Muguerza, J., Perez, J. M., & Perona, I. (2013). An extensive comparative study of cluster validity indices. Pattern Recognition, 46(1), 243-256.

Kim, M., & Ramakrishna, R. S. (2005). New indices for cluster validity assessment. Pattern Recognition Letters, 26(15), 2353-2363.

Lei, Y., Bezdek, J. C., Chan, J., Vinh, N. X., Romano, S., & Bailey, J. (2017). Extending information-theoretic validity indices for fuzzy clustering. IEEE Transactions on Fuzzy Systems, 25(4), 1013-1018.

Lin, H. Y. (2013). Effective Feature Selection for Multi-class Classification Models. In Proceedings of the World Congress on Engineering (Vol. 3).

Meila, M. (2003). Comparing clusterings by the variation of information. In Learning theory and kernel machines (pp. 173-187). Springer Berlin Heidelberg.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics, 20, 53-65.

Saitta, S., Raphael, B., & Smith, I. F. (2007). A bounded index for cluster validity. In International Workshop on Machine Learning and Data Mining in Pattern Recognition (pp. 174-187). Springer Berlin Heidelberg.

Wang, W., & Zhang, Y. (2007). On fuzzy cluster validity indices. Fuzzy sets and systems, 158(19), 2095-2117.

Examples

```
cvi(CharTrajLabels, sample(CharTrajLabels), type = c("ARI", "VI"))
```

`cvi_evaluators` *Cluster comparison based on CVIs*

Description

Create evaluation functions for `compare_clusterings()`.

Usage

```
cvi_evaluators(type = "valid", fuzzy = FALSE, ground.truth = NULL)
```

Arguments

`type`	A character vector with options supported by `cvi()`.
`fuzzy`	Logical indicating whether to use fuzzy CVIs or not.
`ground.truth`	A vector that can be coerced to integers used for the calculation of external CVIs (passed as `b` to `cvi()`).

Details

Think of this as a factory for `compare_clusterings()` that creates functions that can be passed as its `score.clus` and `pick.clus` arguments. It is somewhat limited in scope because it depends on the cluster validity indices available in `cvi()` for scoring and performs *majority voting* for picking. They always assume that no errors occurred.

The scoring function takes the CVIs that are to be minimized and "inverts" them by taking their reciprocal so that maximization can be considered uniformly for the purpose of majority voting. Its ellipsis (`...`) is passed to `cvi()`.

The picking function returns the best configuration if `return.objects` is `FALSE`, or a list with the chosen TSClusters object and the corresponding configuration otherwise.

Refer to the examples in `compare_clusterings()`.

Value

A list with two functions: `score` and `pick`.

Note

To avoid ambiguity, if this function is used, configurations for both fuzzy and crisp clusterings should *not* be provided in the same call to `compare_clusterings()`. In such cases the scoring function may fail entirely, e.g. if it was created with `type = "valid"`.

DBA *DTW Barycenter Averaging*

Description

A global averaging method for time series under DTW (Petitjean, Ketterlin and Gancarski 2011).

Usage

```
DBA(
  X,
  centroid = NULL,
  ...,
  window.size = NULL,
  norm = "L1",
  max.iter = 20L,
  delta = 0.001,
  error.check = TRUE,
  trace = FALSE,
  mv.ver = "by-variable"
)

dba(
  X,
```

```
  centroid = NULL,
  ...,
  window.size = NULL,
  norm = "L1",
  max.iter = 20L,
  delta = 0.001,
  error.check = TRUE,
  trace = FALSE,
  mv.ver = "by-variable"
)
```

Arguments

X	A matrix or data frame where each row is a time series, or a list where each element is a time series. Multivariate series should be provided as a list of matrices where time spans the rows and the variables span the columns of each matrix.
centroid	Optionally, a time series to use as reference. Defaults to a random series of X if NULL. For multivariate series, this should be a matrix with the same characteristics as the matrices in X.
...	Further arguments for dtw_basic(). However, the following are already prespecified: window.size, norm (passed along), and backtrack.
window.size	Window constraint for the DTW calculations. NULL means no constraint. A slanted band is used.
norm	Norm for the local cost matrix of DTW. Either "L1" for Manhattan distance or "L2" for Euclidean distance.
max.iter	Maximum number of iterations allowed.
delta	At iteration i, if all(abs(centroid_{i} - centroid_{i-1}) < delta), convergence is assumed.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.
trace	If TRUE, the current iteration is printed to output.
mv.ver	Multivariate version to use. See below.

Details

This function tries to find the optimum average series between a group of time series in DTW space. Refer to the cited article for specific details on the algorithm.

If a given series reference is provided in centroid, the algorithm should always converge to the same result provided the elements of X keep the same values, although their order may change.

The windowing constraint uses a centered window. The calculations expect a value in window.size that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, window.size = 10, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in 10(2) + 1 = 21 observations falling within the window.

Value

The average time series.

Parallel Computing

Please note that running tasks in parallel does **not** guarantee faster computations. The overhead introduced is sometimes too large, and it's better to run tasks sequentially.

This function uses the RcppParallel package for parallelization. It uses all available threads by default (see `RcppParallel::defaultNumThreads()`), but this can be changed by the user with `RcppParallel::setThreadOptions()`.

An exception to the above is when it is called within a `foreach` parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

This function appears to be very sensitive to numerical inaccuracies if multi-threading is used in a **32 bit** installation. In such systems, consider limiting calculations to 1 thread.

Multivariate series

There are currently 2 versions of DBA implemented for multivariate series (see examples):

- If `mv.ver = "by-variable"`, then each variable of each series in `X` and `centroid` are extracted, and the univariate version of the algorithm is applied to each set of variables, binding the results by column. Therefore, the DTW backtracking is different for each variable.
- If `mv.ver = "by-series"`, then all variables are considered at the same time, so the DTW backtracking is computed based on each multivariate series as a whole. This version was implemented in version 4.0.0 of **dtwclust**, and it is faster, but not necessarily more correct.

Note

The indices of the DTW alignment are obtained by calling `dtw_basic()` with `backtrack = TRUE`.

References

Petitjean F, Ketterlin A and Gancarski P (2011). "A global averaging method for dynamic time warping, with applications to clustering." *Pattern Recognition*, **44**(3), pp. 678 - 693. ISSN 0031-3203, doi:10.1016/j.patcog.2010.09.013, https://www.sciencedirect.com/science/article/pii/S003132031000453X.

Examples

```
# Sample data
data(uciCT)

# Obtain an average for the first 5 time series
dtw_avg <- DBA(CharTraj[1:5], CharTraj[[1]], trace = TRUE)

# Plot
matplot(do.call(cbind, CharTraj[1:5]), type = "l")
points(dtw_avg)
```

```
# Change the provided order
dtw_avg2 <- DBA(CharTraj[5:1], CharTraj[[1]], trace = TRUE)

# Same result?
all.equal(dtw_avg, dtw_avg2)

## Not run:
# ================================================================================
# Multivariate versions
# ================================================================================

# sample centroid reference
cent <- CharTrajMV[[3L]]
# sample series
x <- CharTrajMV[[1L]]
# sample set of series
X <- CharTrajMV[1L:5L]

# the by-series version does something like this for each series and the centroid
alignment <- dtw_basic(x, cent, backtrack = TRUE)
# alignment$index1 and alginment$index2 indicate how to map x to cent (row-wise)

# the by-variable version treats each variable separately
alignment1 <- dtw_basic(x[,1L], cent[,1L], backtrack = TRUE)
alignment2 <- dtw_basic(x[,2L], cent[,2L], backtrack = TRUE)
alignment3 <- dtw_basic(x[,3L], cent[,3L], backtrack = TRUE)

# effectively doing:
X1 <- lapply(X, function(x) { x[,1L] })
X2 <- lapply(X, function(x) { x[,2L] })
X3 <- lapply(X, function(x) { x[,3L] })

dba1 <- dba(X1, cent[,1L])
dba2 <- dba(X2, cent[,2L])
dba3 <- dba(X3, cent[,3L])

new_cent <- cbind(dba1, dba2, dba3)

# sanity check
newer_cent <- dba(X, cent, mv.ver = "by-variable")
all.equal(newer_cent, new_cent, check.attributes = FALSE) # ignore names


## End(Not run)
```

DistmatLowerTriangular-class

Distance matrix's lower triangular

Description

Reference class that is used internally for PAM centroids when `pam.precompute = TRUE` and `pam.sparse = FALSE`. It contains Distmat.

Details

If you wish to, you can use this class to access `dist` elements with `[` as if it were a normal matrix. You can use methods::new passing the `dist` object in a `distmat` argument.

Fields

`distmat` The lower triangular.

Methods

`initialize(..., distmat, series, distance, control, error.check = TRUE)` Initialization based on needed parameters

Examples

```
dm <- new("DistmatLowerTriangular",
       distmat = proxy::dist(CharTraj[1:5], method = "gak", sigma = 5.5, window.size = 10L))

dm[2:3, 4:5]
```

`dtw_basic` *Basic DTW distance*

Description

This is a custom implementation of the DTW algorithm without all the functionality included in `dtw::dtw()`. Because of that, it should be faster, while still supporting the most common options.

Usage

```
dtw_basic(
  x,
  y,
  window.size = NULL,
  norm = "L1",
  step.pattern = dtw::symmetric2,
  backtrack = FALSE,
  normalize = FALSE,
  sqrt.dist = TRUE,
  ...,
  error.check = TRUE
)
```

Arguments

x, y	Time series. Multivariate series must have time spanning the rows and variables spanning the columns.
window.size	Size for slanted band window. NULL means no constraint.
norm	Norm for the LCM calculation, "L1" for Manhattan or "L2" for (squared) Euclidean. See notes.
step.pattern	Step pattern for DTW. Only symmetric1 or symmetric2 supported here. Note that these are *not* characters. See dtw::stepPattern.
backtrack	Also compute the warping path between series? See details.
normalize	Should the distance be normalized? Only supported for symmetric2.
sqrt.dist	Only relevant for norm = "L2", see notes.
...	Currently ignored.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

If backtrack is TRUE, the mapping of indices between series is returned in a list.

The windowing constraint uses a centered window. The calculations expect a value in window.size that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, window.size = 10, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in 10(2) + 1 = 21 observations falling within the window.

Value

The DTW distance. For backtrack = TRUE, a list with:

- distance: The DTW distance.
- index1: x indices for the matched elements in the warping path.
- index2: y indices for the matched elements in the warping path.

Proxy version

The version registered with proxy::dist() is custom (loop = FALSE in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see RcppParallel::defaultNumThreads()), but this can be changed by the user with RcppParallel::setThreadOptions().

An exception to the above is when it is called within a foreach parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - browseVignettes("dtwclust")

It also includes symmetric optimizations to calculate only half a distance matrix when appropriate—only one list of series should be provided in x. Starting with version 6.0.0, this optimization means that the function returns an array with the lower triangular values of the distance matrix, similar to what stats::dist() does; see DistmatLowerTriangular for a helper to access elements as it if

were a normal matrix. If you want to avoid this optimization, call proxy::dist by giving the same list of series in both `x` and `y`.

In order for symmetry to apply here, the following must be true: no window constraint is used (`window.size` is `NULL`) or, if one is used, all series have the same length.

Note

The elements of the local cost matrix are calculated by using either Manhattan or squared Euclidean distance. This is determined by the `norm` parameter. When the squared Euclidean version is used, the square root of the resulting DTW distance is calculated at the end (as defined in Ratanamahatana and Keogh 2004; Lemire 2009; see vignette references). This can be avoided by passing `FALSE` in `sqrt.dist`.

The DTW algorithm (and the functions that depend on it) might return different values in 32 bit installations compared to 64 bit ones.

An infinite distance value indicates that the constraints could not be fulfilled, probably due to a too small `window.size` or a very large length difference between the series.

Examples

```
## Not run:
# ============================================================================================
# Understanding multivariate DTW
# ============================================================================================

# The variables for each multivariate time series are:
# tip force, x velocity, and y velocity
A1 <- CharTrajMV[[1L]] # A character
B1 <- CharTrajMV[[6L]] # B character

# Let's extract univariate time series
A1_TipForce <- A1[,1L] # first variable (column)
A1_VelX <- A1[,2L] # second variable (column)
A1_VelY <- A1[,3L] # third variable (column)
B1_TipForce <- B1[,1L] # first variable (column)
B1_VelX <- B1[,2L] # second variable (column)
B1_VelY <- B1[,3L] # third variable (column)

# Looking at each variable independently:

# Just force
dtw_basic(A1_TipForce, B1_TipForce, norm = "L1", step.pattern = symmetric1)
# Corresponding LCM
proxy::dist(A1_TipForce, B1_TipForce, method = "L1")

# Just x velocity
dtw_basic(A1_VelX, B1_VelX, norm = "L1", step.pattern = symmetric1)
# Corresponding LCM
proxy::dist(A1_VelX, B1_VelX, method = "L1")

# Just y velocity
dtw_basic(A1_VelY, B1_VelY, norm = "L1", step.pattern = symmetric1)
```

```
# Corresponding LCM
proxy::dist(A1_VelY, B1_VelY, method = "L1")

# NOTES:
# In the previous examples there was one LCM for each *pair* of series.
# Additionally, each LCM has dimensions length(A1_*) x length(B1_*)

# proxy::dist won't return the LCM for multivariate series,
# but we can do it manually:
mv_lcm <- function(mvts1, mvts2) {
    # Notice how the number of variables (columns) doesn't come into play here
    num_obs1 <- nrow(mvts1)
    num_obs2 <- nrow(mvts2)

    lcm <- matrix(0, nrow = num_obs1, ncol = num_obs2)

    for (i in 1L:num_obs1) {
        for (j in 1L:num_obs2) {
            # L1 norm for ALL variables (columns).
            # Consideration: mvts1 and mvts2 MUST have the same number of variables
            lcm[i, j] <- sum(abs(mvts1[i,] - mvts2[j,]))
        }
    }

    # return
    lcm
}

# Let's say we start with only x velocity and y velocity for each character
mvts1 <- cbind(A1_VelX, A1_VelY)
mvts2 <- cbind(B1_VelX, B1_VelY)

# DTW distance
dtw_d <- dtw_basic(mvts1, mvts2, norm = "L1", step.pattern = symmetric1)
# Corresponding LCM
lcm <- mv_lcm(mvts1, mvts2) # still 178 x 174
# Sanity check
all.equal(
    dtw_d,
    dtw::dtw(lcm, step.pattern = symmetric1)$distance # supports LCM as input
)

# Now let's consider all variables for each character
mvts1 <- cbind(mvts1, A1_TipForce)
mvts2 <- cbind(mvts2, B1_TipForce)

# Notice how the next code is exactly the same as before,
# even though we have one extra variable now

# DTW distance
dtw_d <- dtw_basic(mvts1, mvts2, norm = "L1", step.pattern = symmetric1)
# Corresponding LCM
lcm <- mv_lcm(mvts1, mvts2) # still 178 x 174
```

```
# Sanity check
all.equal(
    dtw_d,
    dtw::dtw(lcm, step.pattern = symmetric1)$distance # supports LCM as input
)

# By putting things in a list,
# proxy::dist returns the *cross-distance matrix*, not the LCM
series_list <- list(mvts1, mvts2)
distmat <- proxy::dist(series_list, method = "dtw_basic",
                       norm = "L1", step.pattern = symmetric1)
# So this should be TRUE
all.equal(distmat[1L, 2L], dtw_d)

# NOTE: distmat is a 2 x 2 matrix, because there are 2 multivariate series.
# Each *cell* in distmat has a corresponding LCM (not returned by the function).
# Proof:
manual_distmat <- matrix(0, nrow = 2L, ncol = 2L)
for (i in 1L:nrow(manual_distmat)) {
    for (j in 1L:ncol(manual_distmat)) {
        lcm_cell <- mv_lcm(series_list[[i]], series_list[[j]]) # LCM for this pair
        manual_distmat[i, j] <- dtw::dtw(lcm_cell, step.pattern = symmetric1)$distance
    }
}
# TRUE
all.equal(
    as.matrix(distmat),
    manual_distmat
)

## End(Not run)
```

---

dtw_lb *DTW distance matrix guided by Lemire's improved lower bound*

Description

Calculation of a distance matrix with the Dynamic Time Warping (DTW) distance guided by Lemire's improved lower bound (LB_Improved).

Usage

```
dtw_lb(
  x,
  y = NULL,
  window.size = NULL,
  norm = "L1",
  error.check = TRUE,
  pairwise = FALSE,
```

```
  dtw.func = "dtw_basic",
  nn.margin = 1L,
  ...
)
```

Arguments

x, y	A matrix or data frame where rows are time series, or a list of time series.
window.size	Window size to use with the LB and DTW calculation. See details.
norm	Either "L1" for Manhattan distance or "L2" for Euclidean.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.
pairwise	Calculate pairwise distances?
dtw.func	Which function to use for the core DTW calculations, either "dtw" or "dtw_basic". See dtw::dtw() and dtw_basic().
nn.margin	Either 1 to search for nearest neighbors row-wise, or 2 to search column-wise. Only implemented for dtw.func = "dtw_basic".
...	Further arguments for dtw.func or lb_improved().

Details

This function first calculates an initial estimate of a distance matrix between two sets of time series using lb_improved() (the proxy::dist() version). Afterwards, it uses the estimate to calculate the corresponding true DTW distance between *only* the nearest neighbors of each series in x found in y, and it continues iteratively until no changes in the nearest neighbors occur.

If only x is provided, the distance matrix is calculated between all its time series, effectively returning a matrix filled with the LB_Improved values.

This could be useful in case one is interested in only the nearest neighbor of one or more series within a dataset.

The windowing constraint uses a centered window. The calculations expect a value in window.size that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, window.size = 10, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in 10(2) + 1 = 21 observations falling within the window.

Value

The distance matrix with class crossdist.

Parallel Computing

Please note that running tasks in parallel does **not** guarantee faster computations. The overhead introduced is sometimes too large, and it's better to run tasks sequentially.

This function uses the RcppParallel package for parallelization. It uses all available threads by default (see RcppParallel::defaultNumThreads()), but this can be changed by the user with RcppParallel::setThreadOptions().

An exception to the above is when it is called within a foreach parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

Note

This function uses a lower bound that is only defined for time series of equal length.

The `proxy::dist()` version simply calls this function.

A considerably large dataset is probably necessary before this is faster than using `dtw_basic()` with `proxy::dist()`. Also note that `lb_improved()` calculates warping envelopes for the series in `y`, so be careful with the provided order and `nn.margin` (see examples).

Author(s)

Alexis Sarda-Espinosa

References

Lemire D (2009). "Faster retrieval with a two-pass dynamic-time-warping lower bound ." *Pattern Recognition*, **42**(9), pp. 2169 - 2180. ISSN 0031-3203, doi:10.1016/j.patcog.2008.11.030, https://www.sciencedirect.com/science/article/pii/S0031320308004925.

See Also

`lb_keogh()`, `lb_improved()`

Examples

```
# Load data
data(uciCT)

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

# Calculate the DTW distance between a certain subset aided with the lower bound
system.time(d <- dtw_lb(data[1:5], data[6:50], window.size = 20L))

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

# Calculate the DTW distances between all elements (slower)
system.time(d2 <- proxy::dist(data[1:5], data[6:50], method = "DTW",
                              window.type = "sakoechiba", window.size = 20L))

# Nearest neighbors
NN2 <- apply(d2, 1L, which.min)

# Calculate the DTW distances between all elements using dtw_basic
# (might be faster, see notes)
system.time(d3 <- proxy::dist(data[1:5], data[6:50], method = "DTW_BASIC",
                              window.size = 20L))
```

```
# Nearest neighbors
NN3 <- apply(d3, 1L, which.min)

# Change order and margin for nearest neighbor search
# (usually fastest, see notes)
system.time(d4 <- dtw_lb(data[6:50], data[1:5],
                         window.size = 20L, nn.margin = 2L))

# Nearest neighbors *column-wise*
NN4 <- apply(d4, 2L, which.min)

# Same results?
identical(NN1, NN2)
identical(NN1, NN3)
identical(NN1, NN4)
```

---

dtw2 *DTW distance with L2 norm*

Description

Wrapper for the dtw::dtw() function using L2 norm for both the local cost matrix (LCM) creation as well as the final cost aggregation step.

Usage

```
dtw2(x, y, ...)
```

Arguments

x, y	A time series. A multivariate series should have time spanning the rows and variables spanning the columns.
...	Further arguments for dtw::dtw().

Details

The L-norms are used in two different steps by the DTW algorithm. First when creating the LCM, where the element (i, j) of the matrix is computed as the L-norm of $x_i^v - y_j^v$ for all variables v. Note that this means that, in case of multivariate series, they must have the same number of variables, and that univariate series will produce the same LCM regardless of the L-norm used. After the warping path is found by DTW, the final distance is calculated as the L-norm of all (i, j) elements of the LCM that fall on the warping path.

The dtw::dtw() function allows changing the norm by means of its `dist.method` parameter, but it only uses it when creating the LCM, and not when calculating the final aggregated cost, i.e. the DTW distance.

This wrapper simply returns the appropriate DTW distance using L2 norm (Euclidean norm). A `proxy::dist()` version is also registered.

The windowing constraint uses a centered window. The calculations expect a value in `window.size` that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, `window.size = 10`, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in `10(2) + 1 = 21` observations falling within the window.

Value

An object of class `dtw`.

`dtwclustTimings` *Results of timing experiments*

Description

This is the list with data frames containing the results of the timing experiments vignette included with **dtwclust**. See `browseVignettes("dtwclust")`.

Format

The results are organized into different data frames and saved in one list with nested lists. For more details, refer to the included vignette or the scripts available at https://github.com/asardaes/dtwclust/tree/master/timing-experiments.

Source

Refer to the timing experiments vignette.

`GAK` *Fast global alignment kernels*

Description

Distance based on (triangular) global alignment kernels.

Usage

```
GAK(
  x,
  y,
  ...,
  sigma = NULL,
  window.size = NULL,
  normalize = TRUE,
```

```
  error.check = TRUE
)

gak(
  x,
  y,
  ...,
  sigma = NULL,
  window.size = NULL,
  normalize = TRUE,
  error.check = TRUE
)
```

Arguments

x, y	Time series. A multivariate series should have time spanning the rows and variables spanning the columns.
...	Currently ignored.
sigma	Parameter for the Gaussian kernel's width. See details for the interpretation of NULL.
window.size	Parameterization of the constraining band (T in Cuturi (2011)). See details.
normalize	Normalize the result by considering diagonal terms.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

This function uses the Triangular Global Alignment Kernel (TGAK) described in Cuturi (2011). It supports series of different length and multivariate series, so long as the ratio of the series' lengths doesn't differ by more than 2 (or less than 0.5).

The window.size parameter is similar to the one used in DTW, so NULL signifies no constraint, and its value should be greater than 1 if used with series of different length.

The Gaussian kernel is parameterized by sigma. Providing NULL means that the value will be estimated by using the strategy mentioned in Cuturi (2011) with a constant of 1. This estimation is subject to **randomness**, so consider estimating the value once and re-using it (the estimate is returned as an attribute of the result). See the examples.

For more information, refer to the package vignette and the referenced article.

Value

The logarithm of the GAK if normalize = FALSE, otherwise 1 minus the normalized GAK. The value of sigma is assigned as an attribute of the result.

Proxy version

The version registered with proxy::dist() is custom (loop = FALSE in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see RcppParallel::defaultNumThreads()), but this can be changed by the user with RcppParallel::setThreadOptions().

An exception to the above is when it is called within a foreach parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - browseVignettes("dtwclust")

It also includes symmetric optimizations to calculate only half a distance matrix when appropriate—only one list of series should be provided in x. Starting with version 6.0.0, this optimization means that the function returns an array with the lower triangular values of the distance matrix, similar to what stats::dist() does; see DistmatLowerTriangular for a helper to access elements as it if were a normal matrix. If you want to avoid this optimization, call proxy::dist by giving the same list of series in both x and y.

Note

The estimation of sigma does *not* depend on window.size.

If normalize is set to FALSE, the returned value is **not** a distance, rather a similarity. The proxy::dist() version is thus always normalized. Use proxy::simil() with method set to "uGAK" if you want the unnormalized similarities.

A constrained unnormalized calculation (i.e. with window.size > 0 and normalize = FALSE) will return negative infinity if abs(NROW(x) - NROW(y)) > window.size. Since the function won't perform calculations in that case, it might be faster, but if this behavior is not desired, consider reinterpolating the time series (see reinterpolate()) or increasing the window size.

References

Cuturi, M. (2011). Fast global alignment kernels. In *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 929-936).

Examples

```
## Not run:
data(uciCT)

set.seed(832)
GAKd <- proxy::dist(zscore(CharTraj), method = "gak",
                    pairwise = TRUE, window.size = 18L)

# Obtained estimate of sigma
sigma <- attr(GAKd, "sigma")

# Use value for clustering
tsclust(CharTraj, k = 20L,
        distance = "gak", centroid = "shape",
        trace = TRUE,
        args = tsclust_args(dist = list(sigma = sigma,
                                        window.size = 18L)))
```

```
## End(Not run)

# Unnormalized similarities
proxy::simil(CharTraj[1L:5L], method = "ugak")
```

`interactive_clustering` *A shiny app for interactive clustering*

Description

Display a shiny user interface to do clustering based on the provided series.

Usage

```
interactive_clustering(series, ...)
```

Arguments

`series`	Time series in the formats accepted by `tsclust()`.
`...`	More arguments for `shiny::runApp()`.

Explore

This part of the app is simply to see some basic characteristics of the provided series and plot some of them. The field for integer IDs expects a valid R expression that specifies which of the `series` should be plotted. Multivariate series are plotted with each variable in a different facet.

Cluster

This part of the app wraps `tsclust()`, so you should be familiar with it. Some remarks:

- Specifying a custom centroid or hierarchical method expects the name of a function available in the R session (without quotes). Naturally, any required package should be loaded before calling `interactive_clustering`. For example, if you want to use `cluster::agnes()`, you should load **cluster** beforehand.
- A random seed of 0 means that it will be left as `NULL` when calling `tsclust()`.
- The input fields for Extra parameters (distance, centroid and ellipsis) expect a comma- separated sequence of key-value pairs. For example: `window.size = 10L, trace = TRUE`. You should be able to pass any variables available in the R session's global environment.
- Regarding plot parameters:
  - The `Clusters` field is like the integer IDs from the Explore section.

- The `Labels` field is passed to the plot method (see TSClusters-methods). You can specify several values like with the Extra parameters, e.g.: `nudge_x = 10`, `nudge_y = 1`. You can type an empty space to activate them with the defaults, and delete everything to hide them. Note that the location of the labels is random each time.

The plot area reacts to the plot parameters, but the actual clustering with `tsclust()` won't be executed until you click the `Cluster!` button. **The plot can take a couple of seconds to load!** Plotting multivariate series might generate warnings about missing values, they can be safely ignored.

Some of the control parameters are disabled when **dtwclust** detects them automatically.

The cross-distance matrix is cached so that it can be re-used when appropriate. The cached version is invalidated automatically when necessary.

Evaluate

This part of the app provides results of the current clustering. External CVIs can be calculated if the name of a variable with the ground truth is provided (see `cvi()`).

Note

Tracing is printed to the console.

Author(s)

Alexis Sarda-Espinosa

Examples

```
## Not run:
interactive_clustering(CharTrajMV)

## End(Not run)
```

`lb_improved` *Lemire's improved DTW lower bound*

Description

This function calculates an improved lower bound (LB) on the Dynamic Time Warp (DTW) distance between two time series. It uses a Sakoe-Chiba constraint.

Usage

```
lb_improved(
  x,
  y,
  window.size = NULL,
  norm = "L1",
  lower.env = NULL,
  upper.env = NULL,
  force.symmetry = FALSE,
  error.check = TRUE
)
```

Arguments

x	A time series (reference).
y	A time series with the same length as x (query).
window.size	Window size for envelope calculation. See details.
norm	Vector norm. Either "L1" for Manhattan distance or "L2" for Euclidean.
lower.env	Optionally, a pre-computed lower envelope for y can be provided (non-proxy version only). See compute_envelope().
upper.env	Optionally, a pre-computed upper envelope for y can be provided (non-proxy version only). See compute_envelope().
force.symmetry	If TRUE, a second lower bound is calculated by swapping x and y, and whichever result has a *higher* distance value is returned. The proxy version can only work if a square matrix is obtained, but use carefully.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

The reference time series should go in x, whereas the query time series should go in y.

If the envelopes are provided, they should be provided together. If either one is missing, both will be computed.

The windowing constraint uses a centered window. The calculations expect a value in window.size that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, window.size = 10, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in 10(2) + 1 = 21 observations falling within the window.

Value

The improved lower bound for the DTW distance.

Proxy version

The version registered with `proxy::dist()` is custom (`loop = FALSE` in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see `RcppParallel::defaultNumThreads()`), but this can be changed by the user with `RcppParallel::setThreadOptions()`.

An exception to the above is when it is called within a `foreach` parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

Note

The lower bound is only defined for time series of equal length and is **not** symmetric.

If you wish to calculate the lower bound between several time series, it would be better to use the version registered with the `proxy` package, since it includes some small optimizations. The convention mentioned above for references and queries still holds. See the examples.

The proxy version of `force.symmetry` should only be used when only `x` is provided or both `x` and `y` are identical. It compares the lower and upper triangular of the resulting distance matrix and forces symmetry in such a way that the tightest lower bound is obtained.

References

Lemire D (2009). "Faster retrieval with a two-pass dynamic-time-warping lower bound ." *Pattern Recognition*, **42**(9), pp. 2169 - 2180. ISSN 0031-3203, doi:10.1016/j.patcog.2008.11.030, https://www.sciencedirect.com/science/article/pii/S0031320308004925.

Examples

```
# Sample data
data(uciCT)

# Lower bound distance between two series
d.lbi <- lb_improved(CharTraj[[1]], CharTraj[[2]], window.size = 20)

# Corresponding true DTW distance
d.dtw <- dtw(CharTraj[[1]], CharTraj[[2]],
             window.type = "sakoechiba", window.size = 20)$distance

d.lbi <= d.dtw

# Calculating the LB between several time series using the 'proxy' package
# (notice how both argments must be lists)
D.lbi <- proxy::dist(CharTraj[1], CharTraj[2:5], method = "LB_Improved",
                     window.size = 20, norm = "L2")

# Corresponding true DTW distance
D.dtw <- proxy::dist(CharTraj[1], CharTraj[2:5], method = "dtw_basic",
                     norm = "L2", window.size = 20)

D.lbi <= D.dtw
```

lb_keogh *Keogh's DTW lower bound*

Description

This function calculates a lower bound (LB) on the Dynamic Time Warp (DTW) distance between two time series. It uses a Sakoe-Chiba constraint.

Usage

```
lb_keogh(
  x,
  y,
  window.size = NULL,
  norm = "L1",
  lower.env = NULL,
  upper.env = NULL,
  force.symmetry = FALSE,
  error.check = TRUE
)
```

Arguments

x	A time series (reference).
y	A time series with the same length as x (query).
window.size	Window size for envelope calculation. See details.
norm	Vector norm. Either "L1" for Manhattan distance or "L2" for Euclidean.
lower.env	Optionally, a pre-computed lower envelope for y can be provided (non-proxy version only). See compute_envelope().
upper.env	Optionally, a pre-computed upper envelope for y can be provided (non-proxy version only). See compute_envelope().
force.symmetry	If TRUE, a second lower bound is calculated by swapping x and y, and whichever result has a *higher* distance value is returned. The proxy version can only work if a square matrix is obtained, but use carefully.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

The reference time series should go in x, whereas the query time series should go in y.

If the envelopes are provided, they should be provided together. If either one is missing, both will be computed.

The windowing constraint uses a centered window. The calculations expect a value in window.size that represents the distance between the point considered and one of the edges of the window.

Therefore, if, for example, `window.size = 10`, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in `10(2) + 1 = 21` observations falling within the window.

Value

A list with:

- `d`: The lower bound of the DTW distance.
- `upper.env`: The time series of `y`'s upper envelope.
- `lower.env`: The time series of `y`'s lower envelope.

Proxy version

The version registered with `proxy::dist()` is custom (`loop = FALSE` in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see `RcppParallel::defaultNumThreads()`), but this can be changed by the user with `RcppParallel::setThreadOptions()`.

An exception to the above is when it is called within a `foreach` parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

Note

The lower bound is only defined for time series of equal length and is **not** symmetric.

If you wish to calculate the lower bound between several time series, it would be better to use the version registered with the `proxy` package, since it includes some small optimizations. The convention mentioned above for references and queries still holds. See the examples.

The proxy version of `force.symmetry` should only be used when only `x` is provided or both `x` and `y` are identical. It compares the lower and upper triangular of the resulting distance matrix and forces symmetry in such a way that the tightest lower bound is obtained.

References

Keogh E and Ratanamahatana CA (2005). "Exact indexing of dynamic time warping." *Knowledge and information systems*, **7**(3), pp. 358-386.

Examples

```
# Sample data
data(uciCT)

# Lower bound distance between two series
d.lbk <- lb_keogh(CharTraj[[1]], CharTraj[[2]], window.size = 20)$d

# Corresponding true DTW distance
d.dtw <- dtw(CharTraj[[1]], CharTraj[[2]],
             window.type = "sakoechiba", window.size = 20)$distance

d.lbk <= d.dtw
```

```
# Calculating the LB between several time series using the 'proxy' package
# (notice how both argments must be lists)
D.lbk <- proxy::dist(CharTraj[1], CharTraj[2:5], method = "LB_Keogh",
                     window.size = 20, norm = "L2")

# Corresponding true DTW distance
D.dtw <- proxy::dist(CharTraj[1], CharTraj[2:5], method = "dtw_basic",
                     norm = "L2", window.size = 20)

D.lbk <= D.dtw
```

NCCc *Cross-correlation with coefficient normalization*

Description

This function uses the FFT to compute the cross-correlation sequence between two series. They need *not* be of equal length.

Usage

```
NCCc(x, y, error.check = TRUE)
```

Arguments

x, y	Univariate time series.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Value

The cross-correlation sequence with length `length(x) + length(y) - 1L`.

References

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

See Also

SBD()

pam_cent *Centroid for partition around medoids*

Description

Extract the medoid time series based on a distance measure.

Usage

```
pam_cent(
  series,
  distance,
  ids = seq_along(series),
  distmat = NULL,
  ...,
  error.check = TRUE
)
```

Arguments

series	The time series in one of the formats accepted by tslist().
distance	A character indicating which distance to use. Only needed if distmat is NULL. The distance must be registered in proxy::pr_DB().
ids	Integer vector indicating which of the series should be considered.
distmat	Optionally, a pre-computed cross-distance matrix of *all* series.
...	Any extra parameters for the distance function that may be used.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

The medoid's index is determined by taking the $argmin$ of the distmat's row-sums (considering only the rows in ids). The distance matrix is calculated if needed.

Value

The medoid time series.

Examples

```
# Computes the distance matrix for all series
pam_cent(CharTraj, "dtw_basic", ids = 6L:10L, window.size = 15L) # series_id = 7L

# Computes the distance matrix for the chosen subset only
pam_cent(CharTraj[6L:10L], "dtw_basic", window.size = 15L) # series_id = 2L
```

pdc_configs *Helper function for preprocessing/distance/centroid configurations*

Description

Create preprocessing, distance and centroid configurations for `compare_clusterings_configs()`.

Usage

```
pdc_configs(
  type = c("preproc", "distance", "centroid"),
  ...,
  partitional = NULL,
  hierarchical = NULL,
  fuzzy = NULL,
  tadpole = NULL,
  share.config = c("p", "h", "f", "t")
)
```

Arguments

`type`	Which type of function is being targeted by this configuration.
`...`	Any number of named lists with functions and arguments that will be shared by all clusterings. See details.
`partitional`	A named list of lists with functions and arguments for partitional clusterings.
`hierarchical`	A named list of lists with functions and arguments for hierarchical clusterings.
`fuzzy`	A named list of lists with functions and arguments for fuzzy clusterings.
`tadpole`	A named list of lists with functions and arguments for TADPole clusterings.
`share.config`	A character vector specifying which clusterings should include the shared lists (the ones specified in `...`). It must be any combination of (possibly abbreviated): partitional, hierarchical, fuzzy, tadpole.

Details

The named lists are interpreted in the following way: the name of the list will be considered to be a function name, and the elements of the list will be the possible parameters for the function. Each function must have at least an empty list. The parameters may be vectors that specify different values to be tested.

For preprocessing, the special name `none` signifies no preprocessing.

For centroids, the special name `default` leaves the centroid unspecified.

Please see the examples in `compare_clusterings()` to see how this is used.

Value

A list for each clustering, each of which includes a data frame with the computed configurations.

`reinterpolate` *Wrapper for simple linear reinterpolation*

Description

This function is just a wrapper for the native function `stats::approx()` to do simple linear reinterpolation. It also supports matrices, data frames, and lists of time series.

Usage

```
reinterpolate(x, new.length, multivariate = FALSE)
```

Arguments

`x`	Data to reinterpolate. Either a vector, a matrix/data.frame where each row is to be reinterpolated, or a list of vectors/matrices.
`new.length`	Desired length of the output series.
`multivariate`	Is `x` a multivariate time series? It will be detected automatically if a list is provided in `x`.

Details

Multivariate series must have time spanning the rows and variables spanning the columns.

Value

Reinterpolated time series

Examples

```
data(uciCT)

# list of univariate series
series <- reinterpolate(CharTraj, 205L)

# list of multivariate series
series <- reinterpolate(CharTrajMV, 205L)

# single multivariate series
series <- reinterpolate(CharTrajMV[[1L]], 205L, TRUE)
```

repeat_clustering *Repeat a clustering configuration*

Description

Repeat a clustering made with compare_clusterings() in order to obtain the TSClusters object.

Usage

```
repeat_clustering(series, clusterings, config_id, ...)
```

Arguments

series	The same time series that were given to compare_clusterings().
clusterings	The list returned by compare_clusterings().
config_id	The character indicating which configuration should be re-computed. Obtained from the clusterings' results' data frames.
...	More arguments for tsclust() (e.g. trace).

Details

Since the purpose of compare_clusterings() is to test many configurations, it is desirable to set its return.objects parameter to FALSE in order to save RAM. This function can then be used to compute the clustering object for a specific config_id.

Value

A TSClusters object.

Limitations

If the preprocessing function is subject to randomness, the clustering will not be correctly re-created by this function, since compare_clusterings() applies all preprocessing before calling tsclust().

If any parameters were given to compare_clusterings() through its ellipsis, they should probably be given to this function too.

SBD *Shape-based distance*

Description

Distance based on coefficient-normalized cross-correlation as proposed by Paparrizos and Gravano (2015) for the k-Shape clustering algorithm.

Usage

```
SBD(x, y, znorm = FALSE, error.check = TRUE, return.shifted = TRUE)

sbd(x, y, znorm = FALSE, error.check = TRUE, return.shifted = TRUE)
```

Arguments

x, y	Univariate time series.
znorm	Logical. Should each series be z-normalized before calculating the distance?
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.
return.shifted	Logical. Should the shifted version of y be returned? See details.

Details

This distance works best if the series are *z-normalized*. If not, at least they should have appropriate amplitudes, since the values of the signals **do** affect the outcome.

If x and y do **not** have the same length, it would be best if the longer sequence is provided in y, because it will be shifted to match x. After matching, the series may have to be truncated or extended and padded with zeros if needed.

The output values lie between 0 and 2, with 0 indicating perfect similarity.

Value

For return.shifted = FALSE, the numeric distance value, otherwise a list with:

- dist: The shape-based distance between x and y.
- yshift: A shifted version of y so that it optimally matches x (based on NCCc()).

Proxy version

The version registered with proxy::dist() is custom (loop = FALSE in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see RcppParallel::defaultNumThreads()), but this can be changed by the user with RcppParallel::setThreadOptions().

An exception to the above is when it is called within a foreach parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

It also includes symmetric optimizations to calculate only half a distance matrix when appropriate—only one list of series should be provided in `x`. Starting with version 6.0.0, this optimization means that the function returns an array with the lower triangular values of the distance matrix, similar to what `stats::dist()` does; see DistmatLowerTriangular for a helper to access elements as it if were a normal matrix. If you want to avoid this optimization, call proxy::dist by giving the same list of series in both `x` and `y`.

In some situations, e.g. for relatively small distance matrices, the overhead introduced by the logic that computes only half the distance matrix can be bigger than just calculating the whole matrix.

Note

If you wish to calculate the distance between several time series, it would be better to use the version registered with the `proxy` package, since it includes some small optimizations. See the examples.

This distance is calculated with help of the Fast Fourier Transform, so it can be sensitive to numerical precision. Thus, this function (and the functions that depend on it) might return different values in 32 bit installations compared to 64 bit ones.

References

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

See Also

`NCCc()`, `shape_extraction()`

Examples

```
# load data
data(uciCT)

# distance between series of different lengths
sbd <- SBD(CharTraj[[1]], CharTraj[[100]], znorm = TRUE)$dist

# cross-distance matrix for series subset (notice the two-list input)
sbD <- proxy::dist(CharTraj[1:10], CharTraj[1:10], method = "SBD", znorm = TRUE)
```

`sdtw` *Soft-DTW distance*

Description

Soft-DTW distance measure as proposed in Cuturi and Blondel (2017).

Usage

```
sdtw(x, y, gamma = 0.01, ..., error.check = TRUE)
```

Arguments

`x`, `y`	Time series. Multivariate series must have time spanning the rows and variables spanning the columns.
`gamma`	Positive regularization parameter, with lower values resulting in less smoothing.
`...`	Currently ignored.
`error.check`	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

Unlike other distances, soft-DTW can return negative values, and `sdtw(x, x)` is not always equal to zero. Like DTW, soft-DTW does not fulfill the triangle inequality, but it is always symmetric.

Value

The Soft DTW distance.

Proxy version

The version registered with `proxy::dist()` is custom (`loop = FALSE` in proxy::pr_DB). The custom function handles multi-threaded parallelization directly with RcppParallel. It uses all available threads by default (see `RcppParallel::defaultNumThreads()`), but this can be changed by the user with `RcppParallel::setThreadOptions()`.

An exception to the above is when it is called within a `foreach` parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

It also includes symmetric optimizations to calculate only half a distance matrix when appropriate—only one list of series should be provided in `x`. Starting with version 6.0.0, this optimization means that the function returns an array with the lower triangular values of the distance matrix, similar to what `stats::dist()` does; see DistmatLowerTriangular for a helper to access elements as it if were a normal matrix. If you want to avoid this optimization, call proxy::dist by giving the same list of series in both `x` and `y`.

Note that, due to the fact that this distance is not always zero when a series is compared against itself, this optimization is likely problematic for soft-DTW, as the `dist` object will be handled by many functions as if it had only zeroes in the diagonal. An exception is `tsclust()` when using partitional clustering with PAM centroids—actual diagonal values will be calculated and considered internally in that case.

References

Cuturi, M., & Blondel, M. (2017). Soft-DTW: a Differentiable Loss Function for Time-Series. arXiv preprint arXiv:1703.01541.

`sdtw_cent` *Centroid calculation based on soft-DTW*

Description

Soft-DTW centroid function as proposed in Cuturi and Blondel (2017).

Usage

```
sdtw_cent(
  series,
  centroid = NULL,
  gamma = 0.01,
  weights = rep(1, length(series)),
  ...,
  error.check = TRUE,
  opts = list(algorithm = "NLOPT_LD_LBFGS", maxeval = 20L)
)
```

Arguments

`series`	A matrix or data frame where each row is a time series, or a list where each element is a time series. Multivariate series should be provided as a list of matrices where time spans the rows and the variables span the columns of each matrix.
`centroid`	Optionally, a time series to use as reference. Defaults to a random series of `series` if `NULL`. For multivariate series, this should be a matrix with the same characteristics as the matrices in `series`.
`gamma`	Positive regularization parameter, with lower values resulting in less smoothing.
`weights`	A vector of weights for each element of `series`.
`...`	Further arguments for the optimization backend (except `opts` for `nloptr`, `control` for `optim`, and `...` for both).
`error.check`	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

opts — List of options to pass to nloptr or stats::optim()'s control. The defaults in the function's formals are for nloptr, but the value will be adjusted for optim if needed.

Details

This function can delegate the optimization to the **nloptr** package. For that to happen, you must load it with either base::library() or base::loadNamespace(). If the aforementioned is not fulfilled, the function will delegate to stats::optim().

Value

The resulting centroid, with the optimization results as attributes (except for the returned centroid).

Parallel Computing

Please note that running tasks in parallel does **not** guarantee faster computations. The overhead introduced is sometimes too large, and it's better to run tasks sequentially.

This function uses the RcppParallel package for parallelization. It uses all available threads by default (see RcppParallel::defaultNumThreads()), but this can be changed by the user with RcppParallel::setThreadOptions().

An exception to the above is when it is called within a foreach parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - browseVignettes("dtwclust")

For unknown reasons, this function has returned different results (in the order of 1e-6) when using multi-threading in x64 Windows installations in comparison to other environments (using nloptr v1.0.4). Consider limiting the number of threads if you run into reproducibility problems.

References

Cuturi, M., & Blondel, M. (2017). Soft-DTW: a Differentiable Loss Function for Time-Series. arXiv preprint arXiv:1703.01541.

---

shape_extraction *Shape average of several time series*

Description

Time-series shape extraction based on optimal alignments as proposed by Paparrizos and Gravano (2015) for the k-Shape clustering algorithm.

Usage

```
shape_extraction(X, centroid = NULL, znorm = FALSE, ..., error.check = TRUE)
```

Arguments

X	A matrix or data frame where each row is a time series, or a list where each element is a time series. Multivariate series should be provided as a list of matrices where time spans the rows and the variables span the columns.
centroid	Optionally, a time series to use as reference. Defaults to a random series of X if NULL. For multivariate series, this should be a matrix with the same characteristics as the matrices in X. **It will be z-normalized**.
znorm	Logical flag. Should z-scores be calculated for X before processing?
...	Further arguments for zscore().
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

This works only if the series are *z-normalized*, since the output will also have this normalization.

The resulting centroid will have the same length as centroid if provided. Otherwise, there are two possibilities: if all series from X have the same length, all of them will be used as-is, and the output will have the same length as the series; if series have different lengths, a series will be chosen at random and used as reference. The output series will then have the same length as the chosen series.

This centroid computation is cast as an optimization problem called maximization of Rayleigh Quotient. It depends on the SBD() algorithm. See the cited article for more details.

Value

Centroid time series (z-normalized).

References

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

See Also

SBD(), zscore()

Examples

```
# Sample data
data(uciCT)

# Normalize desired subset
X <- zscore(CharTraj[1:5])

# Obtain centroid series
C <- shape_extraction(X)
```

```
# Result
matplot(do.call(cbind, X),
        type = "l", col = 1:5)
points(C)
```

`ssdtwclust` *A shiny app for semi-supervised DTW-based clustering*

Description

Display a shiny user interface that implements the approach in Dau et al. (2016).

Usage

```
ssdtwclust(series, ..., complexity = NULL)
```

Arguments

`series`	Time series in the formats accepted by `compare_clusterings()`.
`...`	More arguments for `shiny::runApp()`.
`complexity`	A function to calculate a constraint's complexity. See details in the Cluster section.

Details

The approach developed in Dau et al. (2016) argues that finding a good value of `window.size` for the DTW distance is very important, and suggests how to find one by using user-provided feedback. After clustering is done, a pair of series is presented at a time, and the user must annotate the pair as:

- Must link: the series should be in the same cluster.
- Cannot link: the series should *not* be in the same cluster.
- Skip: the choice is unclear.

After each step, a good value of the window size is suggested by evaluating which clusterings fulfill the constraint(s) so far, and how (see Dau et al. (2016) for more information), and performing a majority vote using the window sizes inferred from each constraint. The (main) procedure is thus interactive and can be abandoned at any point.

Explore

This part of the app is simply to see some basic characteristics of the provided series and plot some of them. The field for integer IDs expects a valid R expression that specifies which of the `series` should be plotted. Multivariate series are plotted with each variable in a different facet.

Cluster

This part of the app implements the main procedure by leveraging `compare_clusterings()`. The interface is similar to `interactive_clustering()`, so it's worth checking its documentation too. Since `compare_clusterings()` supports parallelization with `foreach::foreach()`, you can register a parallel backend before opening the shiny app, but you should pre-load the workers with the necessary packages and/or functions. See `parallel::clusterEvalQ()` and `parallel::clusterExport()`, as well as the examples below.

The range of window sizes is specified with a slider, and represents the size as a percentage of the shortest series' length. The `step` parameter indicates how spaced apart should the sizes be (parameter `'by'` in `base::seq()`). A 0-size window should only be used if all series have the same length. If the series have different lengths, using small window sizes can be problematic if the length differences are very big, see the notes in `dtw_basic()`.

A `window.size` should *not* be specified in the extra parameters, it will be replaced with the computed values based on the slider. Using `dba()` centroid is detected, and will use the same window sizes.

For partitional clusterings with many repetitions, and hierarchical clusterings with many linkage methods, the resulting partitions are aggregated by calling `clue::cl_medoid()` with the specified aggregation `method`.

By default, complexity of a constraint is calculated differently from what is suggested in Dau et al. (2016):

- Allocate a logical flag vector with length equal to the number of tested window sizes.
- For each window size, set the corresponding flag to `TRUE` if the constraint given by the user is fulfilled.
- Calculate complexity as: (number of sign changes in the vector) / (number of window sizes - 1L) / (maximum number of *contiguous* `TRUE` flags).

You can provide your own function in the `complexity` parameter. It will receive the flag vector as only input, and a single number is expected as a result.

The complexity threshold can be specified in the app. Any constraint whose complexity is higher than the threshold will not be considered for the majority vote. Constraints with a complexity of 0 are also ignored. An infinite complexity means that the constraint is never fulfilled by any clustering.

Evaluate

This section provides numerical results for reference. The latest results can be saved in the global environment, which includes clustering results, constraints so far, and the suggested window size. Since this includes everything returned by `compare_clusterings()`, you could also use `repeat_clustering()` afterwards.

The constraint plots depict if the constraints are fulfilled or not for the given window sizes, where 1 means it was fulfilled and 0 means it wasn't. An error about a zero-dimension viewport indicates the plot height is too small to fit the plots, so please increase the height.

Note

The optimization mentioned in section 3.4 of Dau et al. (2016) is also implemented here.

Tracing is printed to the console.

Author(s)

Alexis Sarda-Espinosa

References

Dau, H. A., Begum, N., & Keogh, E. (2016). Semi-supervision dramatically improves time series clustering under dynamic time warping. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (pp. 999-1008). ACM. https://sites.google.com/site/dtwclustering/

See Also

interactive_clustering(), compare_clusterings()

Examples

```
## Not run:
require(doParallel)
workers <- makeCluster(detectCores())
clusterEvalQ(workers, {
    library(dtwclust)
    RcppParallel::setThreadOptions(1L)
})
registerDoParallel(workers)
ssdtwclust(reinterpolate(CharTraj[1L:20L], 150L))

## End(Not run)
```

TADPole *TADPole clustering*

Description

Time-series Anytime Density Peaks Clustering as proposed by Begum et al. (2015).

Usage

```
TADPole(
  data,
  k = 2L,
  dc,
  window.size,
```

```
  error.check = TRUE,
  lb = "lbk",
  trace = FALSE
)

tadpole(
  data,
  k = 2L,
  dc,
  window.size,
  error.check = TRUE,
  lb = "lbk",
  trace = FALSE
)
```

Arguments

`data`	A matrix or data frame where each row is a time series, or a list where each element is a time series. Multivariate series are **not** supported.
`k`	The number of desired clusters. Can be a vector with several values.
`dc`	The cutoff distance(s). Can be a vector with several values.
`window.size`	Window size constraint for DTW (Sakoe-Chiba). See details.
`error.check`	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.
`lb`	Which lower bound to use, "lbk" for `lb_keogh()` or "lbi" for `lb_improved()`.
`trace`	Logical flag. If `TRUE`, more output regarding the progress is printed to screen.

Details

This function can be called either directly or through `tsclust()`.

TADPole clustering adopts a relatively new clustering framework and adapts it to time series clustering with DTW. See the cited article for the details of the algorithm.

Because of the way the algorithm works, it can be considered a kind of Partitioning Around Medoids (PAM). This means that the cluster centroids are always elements of the data. However, this algorithm is deterministic, depending on the value of `dc`.

The algorithm first uses the DTW's upper and lower bounds (Euclidean and LB_Keogh respectively) to find series with many close neighbors (in DTW space). Anything below the cutoff distance (`dc`) is considered a neighbor. Aided with this information, the algorithm then tries to prune as many DTW calculations as possible in order to accelerate the clustering procedure. The series that lie in dense areas (i.e. that have lots of neighbors) are taken as cluster centroids.

The algorithm relies on the DTW bounds, which are only defined for univariate time series of equal length.

Parallelization is supported in the following way:

- For multiple `dc` values, multi-processing with `foreach::foreach()`.

- The internal distance calculations use multi-threading with RcppParallel::RcppParallel.

The windowing constraint uses a centered window. The calculations expect a value in `window.size` that represents the distance between the point considered and one of the edges of the window. Therefore, if, for example, `window.size = 10`, the warping for an observation x_i considers the points between x_{i-10} and x_{i+10}, resulting in `10(2) + 1 = 21` observations falling within the window.

Value

A list with:

- `cl`: Cluster indices.
- `centroids`: Indices of the centroids.
- `distCalcPercentage`: Percentage of distance calculations that were actually performed.

For multiple k/dc values, a list of lists is returned, each internal list having the aforementioned elements.

Parallel Computing

Please note that running tasks in parallel does **not** guarantee faster computations. The overhead introduced is sometimes too large, and it's better to run tasks sequentially.

This function uses the RcppParallel package for parallelization. It uses all available threads by default (see `RcppParallel::defaultNumThreads()`), but this can be changed by the user with `RcppParallel::setThreadOptions()`.

An exception to the above is when it is called within a `foreach` parallel loop **made by dtwclust**. If the parallel workers do not have the number of threads explicitly specified, this function will default to 1 thread per worker. See the parallelization vignette for more information - `browseVignettes("dtwclust")`

References

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

`tsclust` *Time series clustering*

Description

This is the main function to perform time series clustering. See the details and the examples for more information, as well as the included package vignettes (which can be found by typing `browseVignettes("dtwclust")`). A convenience wrapper is available in `compare_clusterings()`, and a shiny app in `interactive_clustering()`.

Usage

```
tsclust(
  series = NULL,
  type = "partitional",
  k = 2L,
  ...,
  preproc = NULL,
  distance = "dtw_basic",
  centroid = ifelse(type == "fuzzy", "fcm", "pam"),
  control = do.call(paste0(type, "_control"), list()),
  args = tsclust_args(),
  seed = NULL,
  trace = FALSE,
  error.check = TRUE
)
```

Arguments

series	A list of series, a numeric matrix or a data frame. Matrices and data frames are coerced to a list row-wise (see tslist()).
type	What type of clustering method to use: "partitional", "hierarchical", "tadpole" or "fuzzy".
k	Number of desired clusters. It can be a numeric vector with different values.
...	Arguments to pass to preprocessing, centroid **and** distance functions (added to args). Also passed to method from hierarchical_control() if it happens to be a function, and to stats::hclust() if it contains the members parameter.
preproc	Function to preprocess data. Defaults to zscore() *only* if centroid = "shape", but will be replaced by a custom function if provided.
distance	A registered distance from proxy::dist(). Ignored for type = "tadpole".
centroid	Either a supported string, or an appropriate function to calculate centroids when using partitional/hierarchical/tadpole methods. See Centroids section.
control	An appropriate list of controls. See tsclust-controls.
args	An appropriate list of arguments for preprocessing, distance and centroid functions. See tsclust_args() and the examples.
seed	Random seed for reproducibility.
trace	Logical flag. If TRUE, more output regarding the progress is printed to screen.
error.check	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

Partitional and fuzzy clustering procedures use a custom implementation. Hierarchical clustering is done with stats::hclust() by default. TADPole clustering uses the TADPole() function. Specifying type = "partitional", preproc = zscore, distance = "sbd" and centroid = "shape" is equivalent to the k-Shape algorithm (Paparrizos and Gravano 2015).

The `series` may be provided as a matrix, a data frame or a list. Matrices and data frames are coerced to a list, both row-wise. Only lists can have series with different lengths or multiple dimensions. Most of the optimizations require series to have the same length, so consider reinterpolating them to save some time (see Ratanamahatana and Keogh 2004; `reinterpolate()`). No missing values are allowed.

In the case of multivariate time series, they should be provided as a list of matrices, where time spans the rows of each matrix and the variables span the columns (see CharTrajMV for an example). All included centroid functions should work with the aforementioned format, although `shape` is *not* recommended. Note that the `plot` method will simply append all dimensions (columns) one after the other.

Value

An object with an appropriate class from TSClusters.

If `control$nrep > 1` and a partitional procedure is used, `length(method) > 1` and hierarchical procedures are used, or `length(k) > 1`, a list of objects is returned.

Centroid Calculation

In the case of partitional/fuzzy algorithms, a suitable function should calculate the cluster centroids at every iteration. In this case, the centroids may also be time series. Fuzzy clustering uses the standard fuzzy c-means centroid by default.

In either case, a custom function can be provided. If one is provided, it will receive the following parameters with the shown names (examples for partitional clustering are shown in parentheses):

- `x`: The *whole* data list (`list(ts1, ts2, ts3)`)
- `cl_id`: An integer vector with length equal to the number of series in `data`, indicating which cluster a series belongs to (`c(1L, 2L, 2L)`)
- `k`: The desired number of total clusters (`2L`)
- `cent`: The current centroids in order, in a list (`list(centroid1, centroid2)`)
- `cl_old`: The membership vector of the *previous* iteration (`c(1L, 1L, 2L)`)
- The elements of `...` that match its formal arguments

In case of fuzzy clustering, the membership vectors (2nd and 5th elements above) are matrices with number of rows equal to amount of elements in the data, and number of columns equal to the number of desired clusters. Each row must sum to 1.

The other option is to provide a character string for the custom implementations. The following options are available:

- "mean": The average along each dimension. In other words, the average of all x_i^j among the j series that belong to the same cluster for all time points t_i.
- "median": The median along each dimension. Similar to mean.
- "shape": Shape averaging. By default, all series are z-normalized in this case, since the resulting centroids will also have this normalization. See `shape_extraction()` for more details.
- "dba": DTW Barycenter Averaging. See `DBA()` for more details.
- "sdtw_cent": Soft-DTW centroids, See `sdtw_cent()` for more details.

- "pam": Partition around medoids (PAM). This basically means that the cluster centroids are always one of the time series in the data. In this case, the distance matrix can be pre-computed once using all time series in the data and then re-used at each iteration. It usually saves overhead overall for small datasets (see tsclust-controls).
- "fcm": Fuzzy c-means. Only supported for fuzzy clustering and used by default in that case.
- "fcmdd": Fuzzy c-medoids. Only supported for fuzzy clustering. It **always** precomputes/uses the whole cross-distance matrix.

The `dba`, `shape` and `sdtw_cent` implementations check for parallelization. Note that only `shape`, `dba`, `sdtw_cent`, `pam` and `fcmdd` support series of different length. Also note that for `shape`, `dba` and `sdtw_cent`, this support has a caveat: the final centroids' length will depend on the length of those series that were randomly chosen at the beginning of the clustering algorithm. For example, if the series in the dataset have a length of either 10 or 15, 2 clusters are desired, and the initial choice selects two series with length of 10, the final centroids will have this same length.

As special cases, if hierarchical or tadpole clustering is used, you can provide a centroid function that takes a list of series as first input. It will also receive the contents of `args$cent` that match its formal arguments, and should return a single centroid series. These centroids are returned in the `centroids` slot. By default, the medoid of each cluster is extracted (similar to what `pam_cent()` does).

In the following cases, the `centroids` list will have an attribute `series_id` with an integer vector indicating which `series` were chosen as centroids:

- Partitional clustering using "pam" centroid.
- Fuzzy clustering using "fcmdd" centroid.
- Hierarchical clustering with the default centroid extraction.
- TADPole clustering with the default centroid extraction.

Distance Measures

The distance measure to be used with partitional, hierarchical and fuzzy clustering can be modified with the `distance` parameter. The supported option is to provide a string, which must represent a compatible distance registered with `proxy`'s `proxy::dist()`. Registration is done via `proxy::pr_DB()`, and extra parameters can be provided in `args$dist` (see the examples).

Note that you are free to create your own distance functions and register them. Optionally, you can use one of the following custom implementations (all registered with `proxy`):

- `"dtw"`: DTW, optionally with a Sakoe-Chiba/Slanted-band constraint. Done with `dtw::dtw()`.
- `"dtw2"`: DTW with L2 norm and optionally a Sakoe-Chiba/Slanted-band constraint. See `dtw2()`.
- `"dtw_basic"`: A custom version of DTW with less functionality, but faster. See `dtw_basic()`.
- `"dtw_lb"`: DTW with L1 or L2 norm and a Sakoe-Chiba constraint. Some computations are avoided by first estimating the distance matrix with Lemire's lower bound and then iteratively refining with DTW. See `dtw_lb()`. Not suitable for `pam.precompute = TRUE` nor hierarchical clustering.
- `"lbk"`: Keogh's lower bound for DTW with either L1 or L2 norm for the Sakoe-Chiba constraint. See `lb_keogh()`.

- "lbi": Lemire's lower bound for DTW with either L1 or L2 norm for the Sakoe-Chiba constraint. See lb_improved().
- "sbd": Shape-based distance. See sbd().
- "gak": Global alignment kernels. See gak().
- "sdtw": Soft-DTW. See sdtw().

Out of the aforementioned, only the distances based on DTW lower bounds *don't* support series of different length. The lower bounds are probably unsuitable for direct clustering unless series are very easily distinguishable.

If you know that the distance function is symmetric, and you use a hierarchical algorithm, or a partitional algorithm with PAM centroids, or fuzzy c-medoids, some time can be saved by calculating only half the distance matrix. Therefore, consider setting the symmetric control parameter to TRUE if this is the case (see tsclust-controls).

Preprocessing

It is strongly advised to use z-normalization in case of centroid = "shape", because the resulting series have this normalization (see shape_extraction()). Therefore, zscore() is the default in this case. The user can, however, specify a custom function that performs any transformation on the data, but the user must make sure that the format stays consistent, i.e. a list of time series.

Setting to NULL means no preprocessing (except for centroid = "shape"). A provided function will receive the data as first argument, followed by the contents of args$preproc that match its formal arguments.

It is convenient to provide this function if you're planning on using the stats::predict() generic (see also TSClusters-methods).

Repetitions

Due to their stochastic nature, partitional clustering is usually repeated several times with different random seeds to allow for different starting points. This function uses parallel::nextRNGStream() to obtain different seed streams for each repetition, utilizing the seed parameter (if provided) to initialize it. If more than one repetition is made, the streams are returned in an attribute called rng.

Multiple values of k can also be provided to get different partitions using any type of clustering.

Repetitions are greatly optimized when PAM centroids are used and the whole distance matrix is precomputed, since said matrix is reused for every repetition.

Parallel Computing

Please note that running tasks in parallel does **not** guarantee faster computations. The overhead introduced is sometimes too large, and it's better to run tasks sequentially.

The user can register a parallel backend, e.g. with the **doParallel** package, in order to attempt to speed up the calculations (see the examples). This relies on foreach::foreach(), i.e. it uses multi-processing.

Multi-processing is used in partitional and fuzzy clustering for multiple values of k and/or nrep (in partitional_control()). See TADPole() to know how it uses parallelization. For cross-distance matrix calculations, the parallelization strategy depends on whether the distance is included with **dtwclust** or not, see the caveats in tsclustFamily.

If you register a parallel backend and special packages must be loaded, provide their names in the `packages` element of `control`. Note that "dtwclust" is always loaded in each parallel worker, so that doesn't need to be included. Alternatively, you may want to pre-load **dtwclust** in each worker with `parallel::clusterEvalQ()`.

Note

The lower bounds are defined only for time series of equal length. They are **not** symmetric, and `DTW` is not symmetric in general.

Author(s)

Alexis Sarda-Espinosa

References

Please refer to the package vignette references (which can be loaded by typing `vignette("dtwclust")`).

See Also

TSClusters, TSClusters-methods, tsclustFamily, tsclust-controls, `compare_clusterings()`, `interactive_clustering()`, `ssdtwclust()`.

Examples

```
#' NOTE: More examples are available in the vignette. Here are just some miscellaneous
#' examples that might come in handy. They should all work, but some don't run
#' automatically.

# Load data
data(uciCT)

# ================================================================================
# Simple partitional clustering with Euclidean distance and PAM centroids
# ================================================================================

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

# Subset for speed
series <- series[1:20]
labels <- CharTrajLabels[1:20]

# Making many repetitions
pc.l2 <- tsclust(series, k = 4L,
                 distance = "L2", centroid = "pam",
                 seed = 3247, trace = TRUE,
                 control = partitional_control(nrep = 10L))

# Cluster validity indices
sapply(pc.l2, cvi, b = labels)
```

```
# ============================================================================
# Hierarchical clustering with Euclidean distance
# ============================================================================

# Re-use the distance matrix from the previous example (all matrices are the same)
# Use all available linkage methods for function hclust
hc.l2 <- tsclust(series, type = "hierarchical",
                 k = 4L, trace = TRUE,
                 control = hierarchical_control(method = "all",
                                                distmat = pc.l2[[1L]]@distmat))

# Plot the best dendrogram according to variation of information
plot(hc.l2[[which.min(sapply(hc.l2, cvi, b = labels, type = "VI"))]])

# ============================================================================
# Multivariate time series
# ============================================================================

# Multivariate series, provided as a list of matrices
mv <- CharTrajMV[1L:20L]

# Using GAK distance
mvc <- tsclust(mv, k = 4L, distance = "gak", seed = 390,
               args = tsclust_args(dist = list(sigma = 100)))

# Note how the variables of each series are appended one after the other in the plot
plot(mvc)

## Not run:
# ============================================================================
# This function is more verbose but allows for more explicit fine-grained control
# ============================================================================

tsc <- tsclust(series, k = 4L,
               distance = "gak", centroid = "dba",
               preproc = zscore, seed = 382L, trace = TRUE,
               control = partitional_control(iter.max = 30L),
               args = tsclust_args(preproc = list(center = FALSE),
                                   dist = list(window.size = 20L,
                                               sigma = 100),
                                   cent = list(window.size = 15L,
                                               norm = "L2",
                                               trace = TRUE)))

# ============================================================================
# Registering a custom distance with the 'proxy' package and using it
# ============================================================================

# Normalized asymmetric DTW distance
ndtw <- function(x, y, ...) {
    dtw::dtw(x, y, step.pattern = asymmetric,
             distance.only = TRUE, ...)$normalizedDistance
}
```

```
# Registering the function with 'proxy'
if (!pr_DB$entry_exists("nDTW"))
    proxy::pr_DB$set_entry(FUN = ndtw, names=c("nDTW"),
                           loop = TRUE, type = "metric", distance = TRUE,
                           description = "Normalized asymmetric DTW")

# Subset of (original) data for speed
pc.ndtw <- tsclust(series[-1L], k = 4L,
                   distance = "nDTW",
                   seed = 8319,
                   trace = TRUE,
                   args = tsclust_args(dist = list(window.size = 18L)))

# Which cluster would the first series belong to?
# Notice that newdata is provided as a list
predict(pc.ndtw, newdata = series[1L])

# ============================================================================================
# Custom hierarchical clustering
# ============================================================================================

require(cluster)

hc.diana <- tsclust(series, type = "h", k = 4L,
                    distance = "L2", trace = TRUE,
                    control = hierarchical_control(method = diana))

plot(hc.diana, type = "sc")

# ============================================================================================
# TADPole clustering
# ============================================================================================

pc.tadp <- tsclust(series, type = "tadpole", k = 4L,
                   control = tadpole_control(dc = 1.5,
                                             window.size = 18L))

# Modify plot, show only clusters 3 and 4
plot(pc.tadp, clus = 3:4,
     labs.arg = list(title = "TADPole, clusters 3 and 4",
                     x = "time", y = "series"))

# Saving and modifying the ggplot object with custom time labels
require(scales)
t <- seq(Sys.Date(), len = length(series[[1L]]), by = "day")
gpc <- plot(pc.tadp, time = t, plot = FALSE)

gpc + ggplot2::scale_x_date(labels = date_format("%b-%Y"),
                            breaks = date_breaks("2 months"))

# ============================================================================================
# Specifying a centroid function for prototype extraction in hierarchical clustering
```

```
# ============================================================================================

# Seed is due to possible randomness in shape_extraction when selecting a basis series
hc.sbd <- tsclust(CharTraj, type = "hierarchical",
                  k = 20L, distance = "sbd",
                  preproc = zscore, centroid = shape_extraction,
                  seed = 320L)

plot(hc.sbd, type = "sc")

# ============================================================================================
# Using parallel computation to optimize several random repetitions
# and distance matrix calculation
# ============================================================================================
require(doParallel)

# Create parallel workers
cl <- makeCluster(detectCores())
invisible(clusterEvalQ(cl, library(dtwclust)))
registerDoParallel(cl)

## Use constrained DTW and PAM
pc.dtw <- tsclust(CharTraj, k = 20L, seed = 3251, trace = TRUE,
                  args = tsclust_args(dist = list(window.size = 18L)))

## Use constrained DTW with DBA centroids
pc.dba <- tsclust(CharTraj, k = 20L, centroid = "dba",
                  seed = 3251, trace = TRUE,
                  args = tsclust_args(dist = list(window.size = 18L),
                                      cent = list(window.size = 18L)))

#' Using distance based on global alignment kernels
pc.gak <- tsclust(CharTraj, k = 20L,
                  distance = "gak",
                  centroid = "dba",
                  seed = 8319,
                  trace = TRUE,
                  control = partitional_control(nrep = 8L),
                  args = tsclust_args(dist = list(window.size = 18L),
                                      cent = list(window.size = 18L)))

# Stop parallel workers
stopCluster(cl)

# Return to sequential computations. This MUST be done if stopCluster() was called
registerDoSEQ()

## End(Not run)
```

`tsclust-controls` *Control parameters for clusterings with* `tsclust()`

Description

Control parameters for fine-grained control.

Usage

```
partitional_control(
  pam.precompute = TRUE,
  iter.max = 100L,
  nrep = 1L,
  symmetric = FALSE,
  packages = character(0L),
  distmat = NULL,
  pam.sparse = FALSE,
  version = 2L
)

hierarchical_control(
  method = "average",
  symmetric = FALSE,
  packages = character(0L),
  distmat = NULL
)

fuzzy_control(
  fuzziness = 2,
  iter.max = 100L,
  delta = 0.001,
  packages = character(0L),
  symmetric = FALSE,
  version = 2L,
  distmat = NULL
)

tadpole_control(dc, window.size, lb = "lbk")

tsclust_args(preproc = list(), dist = list(), cent = list())
```

Arguments

`pam.precompute`	Logical flag. Precompute the whole distance matrix once and reuse it on each iteration if using PAM centroids. Otherwise calculate distances at every iteration. See details.
`iter.max`	Integer. Maximum number of allowed iterations for partitional/fuzzy clustering.
`nrep`	Integer. How many times to repeat clustering with different starting points (i.e., different random seeds).
`symmetric`	Logical flag. Is the distance function symmetric? In other words, is `dist(x,y) == dist(y,x)`? If `TRUE`, only half the distance matrix needs to be computed. Automatically detected and overridden for the distances included in **dtwclust**.

packages	Character vector with the names of any packages required for custom proxy functions. Relevant for parallel computation, although since the distance entries are re-registered in each parallel worker if needed, this is probably useless, but just in case.
distmat	If available, the cross-distance matrix can be provided here. Only relevant for partitional with PAM centroids, fuzzy with FCMdd centroids, or hierarchical clustering.
pam.sparse	Attempt to use a sparse matrix for PAM centroids. See details.
version	Which version of partitional/fuzzy clustering to use. See details.
method	Character vector with one or more linkage methods to use in hierarchical procedures (see stats::hclust()), the character "all" to use all of the available ones, or a function that performs hierarchical clustering based on distance matrices (e.g. cluster::diana()). See details.
fuzziness	Numeric. Exponent used for fuzzy clustering. Commonly termed m in the literature.
delta	Numeric. Convergence criterion for fuzzy clustering.
dc	The cutoff distance for the TADPole algorithm.
window.size	The window.size specifically for the TADPole algorithm.
lb	The lower bound to use with TADPole. Either "lbk" or "lbi".
preproc	A list of arguments for a preprocessing function to be used in tsclust().
dist	A list of arguments for a distance function to be used in tsclust().
cent	A list of arguments for a centroid function to be used in tsclust().

Details

The functions essentially return their function arguments in a classed list, although some checks are performed.

Regarding parameter version: the first version of partitional/fuzzy clustering implemented in the package always performed an extra iteration, which is unnecessary. Use version 1 to mimic this previous behavior.

Partitional

When pam.precompute = FALSE, using pam.sparse = TRUE defines a sparse matrix (refer to Matrix::sparseMatrix()) and updates it every iteration (except for "dtw_lb" distance). For most cases, precomputing the whole distance matrix is still probably faster. See the timing experiments in browseVignettes("dtwclust").

Parallel computations for PAM centroids have the following considerations:

- If pam.precompute is TRUE, both distance matrix calculations and repetitions are done in parallel, regardless of pam.sparse.
- If pam.precompute is FALSE and pam.sparse is TRUE, repetitions are done sequentially, so that the distance calculations can be done in parallel and the sparse matrix updated iteratively.
- If both pam.precompute and pam.sparse are FALSE, repetitions are done in parallel, and each repetition performs distance calculations sequentially, but the distance matrix cannot be updated iteratively.

Hierarchical

There are some limitations when using a custom hierarchical function in `method`: it will receive the lower triangular of the distance matrix as first argument (see `stats::as.dist()`) and the result should support the `stats::as.hclust()` generic. This functionality was added with the **cluster** package in mind, since its functions follow this convention, but other functions could be used if they are adapted to work similarly.

TADPole

When using TADPole, the `dist` argument list includes the `window.size` and specifies `norm = "L2"`.

`TSClusters-class` *Class definition for* `TSClusters` *and derived classes*

Description

Formal S4 classes for time-series clusters. See class hierarchy and slot organization at the **bottom**.

Details

The base class is `TSClusters`. The 3 classes that inherit from it are: `PartitionalTSClusters`, `HierarchicalTSClusters` and `FuzzyTSClusters`.

`HierarchicalTSClusters` also contain `stats::hclust()` as parent class.

Package **clue** is supported, but generics from **flexclust** are not. See also TSClusters-methods.

Slots

`call` The function call.

`family` An object of class tsclustFamily.

`control` An appropriate control object for `tsclust()`. See tsclust-controls.

`datalist` The provided data in the form of a list, where each element is a time series.

`type` A string indicating one of the supported clustering types of `tsclust()`.

`distance` A string indicating the distance used.

`centroid` A string indicating the centroid used.

`preproc` A string indicating the preprocessing used.

`k` Integer indicating the number of desired clusters.

`cluster` Integer vector indicating which cluster a series belongs to (crisp partition). For fuzzy clustering, this is based on **distance**, not on `fcluster`. For hierarchical, this is obtained by calling `stats::cutree()` with the given value of `k`.

`centroids` A list with the centroid time series.

`distmat` If computed, the cross-distance matrix.

`proctime` Time during function execution, as measured with `base::proc.time()`.

`dots` The contents of the original call's ellipsis (...).

`args` The contents of the original call's `args` parameter. See `tsclust_args()`.

`seed` The random seed that was used.

`iter` The number of iterations used.

`converged` A logical indicating whether the function converged.

`clusinfo` A data frame with two columns: `size` indicates the number of series each cluster has, and `av_dist` indicates, for each cluster, the average distance between series and their respective centroids (crisp partition).

`cldist` A column vector with the distance between each series in the data and its corresponding centroid (crisp partition).

`method` A string indicating which hierarchical method was used.

`fcluster` Numeric matrix that contains membership of fuzzy clusters. It has one row for each series and one column for each cluster. The rows must sum to 1. Only relevant for fuzzy clustering.

TSClusters

The base class contains the following slots:

- `call`
- `family`
- `control`
- `datalist`
- `type`
- `distance`
- `centroid`
- `preproc`
- `k`
- `cluster`
- `centroids`
- `distmat`
- `proctime`
- `dots`
- `args`
- `seed`

PartitionalTSClusters

This class adds the following slots to the base class:

- `iter`
- `converged`
- `clusinfo`
- `cldist`

HierarchicalTSClusters

This class adds the following slots to the base class:

- `method`
- `clusinfo`
- `cldist`

FuzzyTSClusters

This class adds the following slots to the base class:

- `iter`
- `converged`
- `fcluster`

See Also

TSClusters-methods

---

tsclusters-methods *Methods for* TSClusters

Description

Methods associated with TSClusters and derived objects.

Usage

```
## S4 method for signature 'TSClusters'
initialize(.Object, ..., override.family = TRUE)

## S4 method for signature 'TSClusters'
show(object)

## S3 method for class 'TSClusters'
update(object, ..., evaluate = TRUE)

## S4 method for signature 'TSClusters'
update(object, ..., evaluate = TRUE)

## S3 method for class 'TSClusters'
predict(object, newdata = NULL, ...)

## S4 method for signature 'TSClusters'
predict(object, newdata = NULL, ...)
```

```
## S3 method for class 'TSClusters'
plot(
  x,
  y,
  ...,
  clus = seq_len(x@k),
  labs.arg = NULL,
  series = NULL,
  time = NULL,
  plot = TRUE,
  type = NULL,
  labels = NULL
)

## S4 method for signature 'TSClusters,missing'
plot(
  x,
  y,
  ...,
  clus = seq_len(x@k),
  labs.arg = NULL,
  series = NULL,
  time = NULL,
  plot = TRUE,
  type = NULL,
  labels = NULL
)
```

Arguments

`.Object`	A `TSClusters` prototype. You *shouldn't* use this, see Initialize section and the examples.
`...`	For `initialize`, any valid slots. For `plot`, passed to `ggplot2::geom_line()` for the plotting of the *cluster centroids*, or to `stats::plot.hclust()`; see Plotting section and the examples. For `update`, any supported argument. Otherwise ignored.
`override.family`	Logical. Attempt to substitute the default family with one that conforms to the provided elements? See Initialize section.
`object, x`	An object that inherits from TSClusters as returned by `tsclust()`.
`evaluate`	Logical. Defaults to `TRUE` and evaluates the updated call, which will result in a new `TSClusters` object. Otherwise, it returns the unevaluated call.
`newdata`	New data to be assigned to a cluster. It can take any of the supported formats of `tsclust()`. Note that for multivariate series, this means that it **must** be a list of matrices, even if the list has only one matrix.
`y`	Ignored.
`clus`	A numeric vector indicating which clusters to plot.

labs.arg	A list with arguments to change the title and/or axis labels. See the examples and ggplot2::labs() for more information.
series	Optionally, the data in the same format as it was provided to tsclust().
time	Optional values for the time axis. If series have different lengths, provide the time values of the longest series.
plot	Logical flag. You can set this to FALSE in case you want to save the ggplot object without printing anything to screen
type	What to plot. NULL means default. See details.
labels	Whether to include labels in the plot (not for dendrogram plots). See details and note that this is subject to **randomness**.

Details

The update method takes the original function call, replaces any provided argument and optionally evaluates the call again. Use evaluate = FALSE if you want to get the unevaluated call. If no arguments are provided, the object is updated to a new version if necessary (this is due to changes in the internal functions of the package, here for backward compatibility).

Value

The plot method returns a gg object (or NULL for dendrogram plot) invisibly.

Initialize

The initialize method is used when calling methods::new(). The family slot can be substituted with an appropriate one if certain elements are provided by the user. The initialize methods of derived classes also inherit the family and can use it to calculate other slots. In order to get a fully functional object, at least the following slots should be provided:

- type: "partitional", "hierarchical", "fuzzy" or "tadpole".
- datalist: The data in one of the supported formats.
- centroids: The time series centroids in one of the supported formats.
- cluster: The cluster indices for each series in the datalist.
- control*: A tsclust-controls object with the desired parameters.
- distance*: A string indicating the distance that should be used.
- centroid*: A string indicating the centroid to use (only necessary for partitional clustering).

*Necessary when overriding the default family for the calculation of other slots, CVIs or prediction. Maybe not always needed, e.g. for plotting.

Prediction

The predict generic can take the usual newdata argument. If NULL, the method simply returns the obtained cluster indices. Otherwise, a nearest-neighbor classification based on the centroids obtained from clustering is performed:

1. newdata is preprocessed with object@family@preproc using the parameters in object@args$preproc.

2. A cross-distance matrix between the processed series and `object@centroids` is computed with `object@family@dist` using the parameters in `object@args$dist`.
3. For non-fuzzy clustering, the series are assigned to their nearest centroid's cluster. For fuzzy clustering, the fuzzy membership matrix for the series is calculated. In both cases, the function in `object@family@cluster` is used.

Plotting

The plot method uses the `ggplot2` plotting system (see `ggplot2::ggplot()`).

The default depends on whether a hierarchical method was used or not. In those cases, the dendrogram is plotted by default; you can pass any extra parameters to `stats::plot.hclust()` via the ellipsis (`...`).

Otherwise, the function plots the time series of each cluster along with the obtained centroid. The default values for cluster centroids are: `linetype = "dashed"`, `linewidth = 1.5`, `colour = "black"`, `alpha = 0.5`. You can change this by means of the ellipsis (`...`).

You can choose what to plot with the `type` parameter. Possible options are:

- `"dendrogram"`: Only available for hierarchical clustering.
- `"series"`: Plot the time series divided into clusters without including centroids.
- `"centroids"`: Plot the obtained centroids only.
- `"sc"`: Plot both series and centroids

In order to enable labels on the (non-dendrogram) plot, you have to select an option that plots the series and at least provide an empty list in the `labels` argument. This list can contain arguments for `ggrepel::geom_label_repel()` and will be passed along. The following are set by the plot method if they are not provided:

- `"mapping"`: set to aes(x = t, y = value, label = label)
- `"data"`: a data frame with as many rows as series in the `datalist` and 4 columns:
  - `t`: x coordinate of the label for each series.
  - `value`: y coordinate of the label for each series.
  - `cl`: index of the cluster to which the series belongs (i.e. `x@cluster`).
  - `label`: the label for the given series (i.e. `names(x@datalist)`).

You can provide your own data frame if you want, but it must have those columns and, even if you override `mapping`, the `cl` column must have that name. The method will attempt to spread the labels across the plot, but note that this is **subject to randomness**, so be careful if you need reproducibility of any commands used after plotting (see examples).

If created, the function returns the gg object invisibly, in case you want to modify it to your liking. You might want to look at `ggplot2::ggplot_build()` if that's the case.

If you want to free the scale of the X axis, you can do the following:

```
plot(x, plot = FALSE) + facet_wrap(~cl, scales = "free")
```

For more complicated changes, you're better off looking at the source code at https://github.com/asardaes/dtwclust/blob/master/R/S4-TSClusters-methods.R and creating your own plotting function.

Examples

```
data(uciCT)

# Assuming this was generated by some clustering procedure
centroids <- CharTraj[seq(1L, 100L, 5L)]
cluster <- unclass(CharTrajLabels)

pc_obj <- new("PartitionalTSClusters",
              type = "partitional", datalist = CharTraj,
              centroids = centroids, cluster = cluster,
              distance = "sbd", centroid = "dba",
              control = partitional_control(),
              args = tsclust_args(cent = list(window.size = 8L, norm = "L2")))

fc_obj <- new("FuzzyTSClusters",
              type = "fuzzy", datalist = CharTraj,
              centroids = centroids, cluster = cluster,
              distance = "sbd", centroid = "fcm",
              control = fuzzy_control())

show(fc_obj)


## Not run:
plot(pc_obj, type = "c", linetype = "solid",
     labs.arg = list(title = "Clusters' centroids"))

set.seed(15L)
plot(pc_obj, labels = list(nudge_x = -5, nudge_y = 0.2),
     clus = c(1L,4L))

## End(Not run)
```

`tsclustFamily-class` *Class definition for* `tsclustFamily`

Description

Formal S4 class with a family of functions used in `tsclust()`.

Details

The custom implementations also handle parallelization.

Since the distance function makes use of **proxy**, it also supports any extra `proxy::dist()` parameters in `...`.

The prototype includes the `cluster` function for partitional methods, as well as a pass-through `preproc` function. The initializer expects a control from tsclust-controls. See more below.

Slots

`dist` The function to calculate the distance matrices.

`allcent` The function to calculate centroids on each iteration.

`cluster` The function used to assign a series to a cluster.

`preproc` The function used to preprocess the data (relevant for `stats::predict()`).

Distance function

The family's dist() function works like `proxy::dist()` but supports parallelization and optimized symmetric calculations. If you like, you can use the function more or less directly, but provide a control argument when creating the family (see examples). However, bear in mind the following considerations.

- The second argument is called `centroids` (inconsistent with `proxy::dist()`).
- If `control$distmat` is *not* `NULL`, the function will try to subset it.
- If `control$symmetric` is `TRUE`, `centroids` is `NULL`, *and* there is no argument `pairwise` that is `TRUE`, only half the distance matrix will be computed.

Note that all distances implemented as part of **dtwclust** have custom proxy loops that use multi-threading independently of **foreach**, so see their respective documentation to see what optimizations apply to each one.

For distances *not* included in **dtwclust**, the computation can be in parallel using multi-processing with `foreach::foreach()`. If you install and load or attach (see `base::library()` or `base::loadNamespace()`) the **bigmemory** package, the function will take advantage of said package when all of the following conditions are met, reducing the overhead of data copying across processes:

- `control$symmetric` is `TRUE`
- `centroids` is `NULL`
- `pairwise` is `FALSE` or `NULL`
- The distance was registered in proxy::pr_DB with `loop = TRUE`
- A parallel backend with more than 1 worker has been registered with **foreach**

This symmetric, parallel case makes chunks for parallel workers, but they are not perfectly balanced, so some workers might finish before the others.

Centroid function

The default partitional allcent() function is a closure with the implementations of the included centroids. The ones for `DBA()`, `shape_extraction()` and `sdtw_cent()` can use multi-process parallelization with `foreach::foreach()`. Its formal arguments are described in the Centroid Calculation section from `tsclust()`.

Note

This class is meant to group together the relevant functions, but they are **not** linked with each other automatically. In other words, neither `dist` nor `allcent` apply `preproc`. They essentially don't know of each other's existence.

See Also

dtw_basic(), dtw_lb(), gak(), lb_improved(), lb_keogh(), sbd(), sdtw().

Examples

```
## Not run:
data(uciCT)
# See "GAK" documentation
fam <- new("tsclustFamily", dist = "gak")

# This is done with symmetric optimizations, regardless of control$symmetric
crossdist <- fam@dist(CharTraj, window.size = 18L)

# This is done without symmetric optimizations, regardless of control$symmetric
crossdist <- fam@dist(CharTraj, CharTraj, window.size = 18L)

# For non-dtwclust distances, symmetric optimizations only apply
# with an appropriate control AND a single data argument:
fam <- new("tsclustFamily", dist = "dtw",
           control = partitional_control(symmetric = TRUE))
fam@dist(CharTraj[1L:5L])

# If you want the fuzzy family, use fuzzy = TRUE
ffam <- new("tsclustFamily", control = fuzzy_control(), fuzzy = TRUE)

## End(Not run)
```

---

tslist *Coerce matrices or data frames to a list of time series*

Description

Change a matrix or data frame to a list of univariate time series

Usage

```
tslist(series, simplify = FALSE)
```

Arguments

series	A matrix or data frame where each row is a time series.
simplify	Coerce all series in the resulting list to either matrix (multivariate) or numeric (univariate).

Details

Almost all functions in **dtwclust** work internally with lists of time series. If you want to avoid constant coercion, create a list of time series once by calling this function.

For matrices and data frames, each **row** is considered as one time series. A list input is simply passed through.

Value

A list of time series.

Note

The function assumes that matrix-like objects can be first coerced via `base::as.matrix()`, so that the result can be indexed with `series[i, ]`.

No consistency checks are performed by this function.

---

uciCT *Subset of character trajectories data set*

Description

Subset: only 5 examples of each considered character. See details.

Format

Lists with 100 elements. Each element is a time series. Labels included as factor vector.

Details

Quoting the source:

"Multiple, labelled samples of pen tip trajectories recorded whilst writing individual characters. All samples are from the same writer, for the purposes of primitive extraction. Only characters with a single pen-down segment were considered."

The subset included in `CharTraj` has only 5 examples of the X velocity for each character. A vector with labels is also loaded in `CharTrajLabels`.

The subset included in `CharTrajMV` has 5 examples too, but includes tip force as well as X and Y velocity. Each element of the list is a multivariate series with 3 variables.

Please note that even though both `CharTraj` and `CharTrajMV` have the same series names, the actual series in each subset are **not** the same, i.e., `CharTraj$A.V1` is not in `CharTrajMV$A.V1`.

Source

https://archive.ics.uci.edu/ml/datasets/Character+Trajectories

zscore *Wrapper for z-normalization*

Description

Wrapper for function `base::scale()` that returns zeros instead of `NaN`. It also supports matrices, data frames, and lists of time series.

Usage

```
zscore(
  x,
  ...,
  multivariate = FALSE,
  keep.attributes = FALSE,
  error.check = TRUE
)
```

Arguments

`x`	Data to normalize. Either a vector, a matrix/data.frame where each row is to be normalized, or a list of vectors/matrices.
`...`	Further arguments to pass to `base::scale()`.
`multivariate`	Is `x` a multivariate time series? It will be detected automatically if a list is provided in `x`.
`keep.attributes`	Should the mean and standard deviation returned by `base::scale()` be preserved?
`error.check`	Logical indicating whether the function should try to detect inconsistencies and give more informative errors messages. Also used internally to avoid repeating checks.

Details

Multivariate series must have time spanning the rows and variables spanning the columns.

Value

Normalized data in the same format as provided.

Index

aes, *70*
as.matrix, 5

base::as.matrix(), *5*, *74*
base::library(), *4*, *48*, *72*
base::loadNamespace(), *48*, *72*
base::Map(), *7*
base::proc.time(), *6*, *65*
base::RNGkind(), *4*
base::scale(), *75*
base::seq(), *51*

CharTraj *(*uciCT*)*, 74
CharTrajLabels *(*uciCT*)*, 74
CharTrajMV, *56*
CharTrajMV *(*uciCT*)*, 74
clue::cl_medoid(), *51*
cluster::agnes(), *33*
cluster::diana(), *64*
compare_clusterings, 5
compare_clusterings(), *4*, *11, 12*, *17, 18*, *41*, *43*, *50–52*, *54*, *59*
compare_clusterings_configs, 11
compare_clusterings_configs(), *6*, *8*, *41*
compute_envelope, 13
compute_envelope(), *35*, *37*
cvi, 14
cvi(), *17, 18*, *34*
cvi,FuzzyTSClusters *(*cvi*)*, 14
cvi,FuzzyTSClusters-method *(*cvi*)*, 14
cvi,HierarchicalTSClusters *(*cvi*)*, 14
cvi,HierarchicalTSClusters-method *(*cvi*)*, 14
cvi,matrix-method *(*cvi*)*, 14
cvi,PartitionalTSClusters *(*cvi*)*, 14
cvi,PartitionalTSClusters-method *(*cvi*)*, 14
cvi_evaluators, 17

DBA, 18
dba *(*DBA*)*, 18
DBA(), *16*, *56*, *72*
dba(), *51*
Distmat, *22*
DistmatLowerTriangular, *23*, *32*, *45, 46*
DistmatLowerTriangular *(*DistmatLowerTriangular-class*)*, 21
DistmatLowerTriangular-class, 21
dtw2, 29
dtw2(), *57*
dtw::dtw(), *4*, *22*, *27*, *29*, *57*
dtw::stepPattern, *23*
dtw_basic, 22
dtw_basic(), *4*, *16*, *19, 20*, *27, 28*, *51*, *57*, *73*
dtw_lb, 26
dtw_lb(), *57*, *73*
dtwclust *(*dtwclust-package*)*, 3
dtwclust-package, 3
dtwclustTimings, 30

flexclust::comPart(), *15*
foreach, *20*, *23*, *28*, *32*, *36*, *38*, *45, 46*, *48*, *54*
foreach::foreach(), *6*, *51*, *53*, *58*, *72*
fuzzy_control *(*tsclust-controls*)*, 62
FuzzyTSClusters *(*TSClusters-class*)*, 65
FuzzyTSClusters-class *(*TSClusters-class*)*, 65

GAK, 30
gak *(*GAK*)*, 30
gak(), *58*, *73*
ggplot2::geom_line(), *68*
ggplot2::ggplot(), *70*
ggplot2::ggplot_build(), *70*
ggplot2::labs(), *69*
ggrepel::geom_label_repel(), *70*

hierarchical_control *(*tsclust-controls*)*, 62

hierarchical_control(), *55*
HierarchicalTSClusters *(TSClusters-class)*, 65
HierarchicalTSClusters-class *(TSClusters-class)*, 65

initialize,TSClusters *(tsclusters-methods)*, 67
initialize,TSClusters-method *(tsclusters-methods)*, 67
interactive_clustering, 33
interactive_clustering(), *4*, *51*, *52*, *54*, *59*

lb_improved, 34
lb_improved(), *27*, *28*, *53*, *58*, *73*
lb_keogh, 37
lb_keogh(), *28*, *53*, *57*, *73*

Matrix::sparseMatrix(), *64*
methods::new, *22*
methods::new(), *69*

NCCc, 39
NCCc(), *44*, *45*

package, *4*
pam_cent, 40
pam_cent(), *57*
parallel::clusterEvalQ(), *51*, *59*
parallel::clusterExport(), *51*
parallel::nextRNGStream(), *58*
partitional_control *(tsclust-controls)*, 62
partitional_control(), *58*
PartitionalTSClusters *(TSClusters-class)*, 65
PartitionalTSClusters-class *(TSClusters-class)*, 65
pdc_configs, 41
pdc_configs(), *6*, *8*, *12*
plot,TSClusters,missing *(tsclusters-methods)*, 67
plot,TSClusters,missing-method *(tsclusters-methods)*, 67
plot.TSClusters *(tsclusters-methods)*, 67
predict,TSClusters *(tsclusters-methods)*, 67
predict,TSClusters-method *(tsclusters-methods)*, 67
predict.TSClusters *(tsclusters-methods)*, 67
proxy::dist, *24*, *32*, *45*, *46*
proxy::dist(), *4*, *23*, *27*, *28*, *30*, *32*, *36*, *38*, *44*, *46*, *55*, *57*, *71*, *72*
proxy::pr_DB, *23*, *32*, *36*, *38*, *44*, *46*, *72*
proxy::pr_DB(), *40*, *57*
proxy::simil(), *32*

RcppParallel, *20*, *23*, *27*, *32*, *36*, *38*, *44*, *46*, *48*, *54*
RcppParallel::defaultNumThreads(), *20*, *23*, *27*, *32*, *36*, *38*, *44*, *46*, *48*, *54*
RcppParallel::RcppParallel, *54*
RcppParallel::setThreadOptions(), *20*, *23*, *27*, *32*, *36*, *38*, *44*, *46*, *48*, *54*
reinterpolate, 42
reinterpolate(), *32*, *56*
repeat_clustering, 43
repeat_clustering(), *6*, *7*, *51*

SBD, 44
sbd *(SBD)*, 44
SBD(), *39*, *49*
sbd(), *58*, *73*
sdtw, 46
sdtw(), *58*, *73*
sdtw_cent, 47
sdtw_cent(), *56*, *72*
shape_extraction, 48
shape_extraction(), *16*, *45*, *56*, *58*, *72*
shiny::runApp(), *33*, *50*
show,TSClusters *(tsclusters-methods)*, 67
show,TSClusters-method *(tsclusters-methods)*, 67
ssdtwclust, 50
ssdtwclust(), *4*, *59*
stats::approx(), *42*
stats::as.dist(), *65*
stats::as.hclust(), *65*
stats::cutree(), *65*
stats::dist(), *23*, *32*, *45*, *46*
stats::hclust(), *55*, *64*, *65*
stats::optim(), *48*
stats::plot.hclust(), *68*, *70*
stats::predict(), *58*, *72*

TADPole, 52
tadpole *(TADPole)*, 52

TADPole(), *16*, *55*, *58*
tadpole_control *(*tsclust-controls*)*, 62
tsclust, 54
tsclust(), *4*, *6*, *8*, *14*, *33, 34*, *43*, *47*, *53*, *62*, *64*, *65*, *68, 69*, *71, 72*
tsclust-controls, *12*, *55*, *57–59*, 62, *65*, *69*, *71*
tsclust_args *(*tsclust-controls*)*, 62
tsclust_args(), *55*, *66*
TSClusters, *7*, *16*, *18*, *43*, *56*, *59*, *67, 68*
TSClusters *(*TSClusters-class*)*, 65
TSClusters-class, 65
TSClusters-methods, *34*, *58, 59*, *65*, *67*
TSClusters-methods *(*tsclusters-methods*)*, 67
tsclusters-methods, 67
tsclustFamily, *58, 59*, *65*
tsclustFamily *(*tsclustFamily-class*)*, 71
tsclustFamily-class, 71
tslist, 73
tslist(), *6*, *40*, *55*

uciCT, 74
ucict *(*uciCT*)*, 74
update,TSClusters *(*tsclusters-methods*)*, 67
update,TSClusters-method *(*tsclusters-methods*)*, 67
update.TSClusters *(*tsclusters-methods*)*, 67
utils::Rscript(), *4*

zscore, 75
zscore(), *49*, *55*, *58*