Package 'rgudhi'

January 20, 2023

Title An Interface to the GUDHI Library for Topological Data Analysis

Version 0.1.0

Description Provides an interface to the GUDHI library which is a generic open source C++ library, with a Python interface, for topological data analysis (TDA) and higher dimensional geometry understanding. The library offers state-of-the-art data structures and algorithms to construct simplicial complexes and compute persistent homology.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.3

URL https://github.com/LMJL-Alea/rgudhi, https://lmjl-alea.github.io/rgudhi/

BugReports https://github.com/lmjl-alea/rgudhi/issues

Config/reticulate list(packages = list(list(package = ``scikit-learn'', version = ``1.2.0''), list(package = ``gudhi'', version = ``3.7.1'')))

Imports cli, curl, dplyr, fs, ggplot2, purrr, R6, Rdpack (>= 2.4.0), reticulate, rlang, tibble, withr

Suggests covr, testthat (>= 3.0.0), vdiffr

Config/testthat/edition 3

RdMacros Rdpack

NeedsCompilation no

Author Aymeric Stamm [aut, cre] (<https://orcid.org/0000-0002-8725-3654>), GUDHI Editorial Board [ctb] (https://gudhi.inria.fr/people/)

Maintainer Aymeric Stamm <aymeric.stamm@cnrs.fr>

Repository CRAN

Date/Publication 2023-01-19 23:30:08 UTC

R topics documented:

`AffinityPropagation` *Performs clustering according to the affinity propagation algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.AffinityPropagation.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `AffinityPropagation`

Methods

Public methods:

- `AffinityPropagation$new()`
- `AffinityPropagation$clone()`

Method `new()`**:** The AffinityPropagation class constructor.

Usage:

```
AffinityPropagation$new(
  damping = 0.5,
  max_iter = 200L,
  convergence_iter = 15L,
  copy = TRUE,
  preference = NULL,
  affinity = c("euclidean", "precomputed"),
  verbose = FALSE,
  random_state = NULL
)
```

Arguments:

`damping` A numeric value specifying the damping factor in the range $[0.5, 1.0)$ which is the extent to which the current value is maintained relative to incoming values (weighted `1 - damping`). This avoids numerical oscillations when updating these values (messages). Defaults to `0.5`.

`max_iter` An integer value specifying the maximum number of iterations. Defaults to `200L`.

`convergence_iter` An integer value specifying the number of iterations with no change in the number of estimated clusters that stops the convergence. Defaults to `15L`.

`copy` A boolean value specifying whether to make a copy of input data. Defaults to `TRUE`.

`preference` A numeric value or numeric vector specifying the preferences for each point. Points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, i.e. of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities. Defaults to `NULL`.

`affinity` A string specifying the affinity to use. At the moment `"precomputed"` and `"euclidean"` are supported. `"euclidean"` uses the negative squared euclidean distance between points. Defaults to `"euclidean"`.

`verbose` A boolean value specifying whether to be verbose. Defaults to `FALSE`.

`random_state` An integer value specifying the seed of the random generator. Defaults to `NULL` which uses current time. Set it to a fixed integer for reproducible results across function calls.

Returns: An object of class AffinityPropagation.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AffinityPropagation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- Brendan J. Frey and Delbert Dueck (2007). *Clustering by Passing Messages Between Data Points*, Science.

Examples

```
cl <- AffinityPropagation$new()
```

`AgglomerativeClustering` *Performs clustering according to the agglomerative algorithm*

Description

Recursively merges pair of clusters of sample data; uses linkage distance. This is a wrapper around the Python class sklearn.cluster.AgglomerativeClustering.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `AgglomerativeClustering`

Methods

Public methods:

- AgglomerativeClustering$new()
- AgglomerativeClustering$clone()

Method `new()`**:** The AgglomerativeClustering class constructor.

Usage:

```
AgglomerativeClustering$new(
  n_clusters = 2L,
  affinity = c("euclidean", "l1", "l2", "manhattan", "cosine", "precomputed"),
  memory = NULL,
  connectivity = NULL,
  compute_full_tree = "auto",
  linkage = c("ward", "complete", "average", "single"),
  distance_threshold = NULL,
  compute_distances = FALSE
)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters to find. It must be `NULL` if `distance_threshold` is not `NULL`. Defaults to `2L`.

`affinity` A string specifying the metric used to compute the linkage. Can be `"euclidean"`, `"l1"`, `"l2"`, `"manhattan"`, `"cosine"` or `"precomputed"`. If `linkage` is `"ward"`, only `"euclidean"` is accepted. If `"precomputed"`, a distance matrix (instead of a similarity matrix) is needed as input for the `$fit()` method. Defaults to `"euclidean"`.

`memory` A string specifying the path to the caching directory. Defaults to `NULL` in which case no caching is done.

`connectivity` Either a numeric matrix or an object of class stats::dist or an object coercible into a function by `rlang::as_function()` specifying for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a function that transforms the data into a connectivity matrix. Defaults to `NULL`, i.e., the hierarchical clustering algorithm is unstructured.

`compute_full_tree` Either a boolean value or the `"auto"` string specifying whether to prematurely stop the construction of the tree at `n_clusters`. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `TRUE` if `distance_threshold` is not `NULL`. Defaults to `"auto"`, which is equivalent to `TRUE` when `distance_threshold` is not `NULL` or that `n_clusters` is inferior to the maximum between `100` and `0.02 * n_samples`. Otherwise, `"auto"` is equivalent to `FALSE`.

`linkage` A string specifying which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

- `ward`: minimizes the variance of the clusters being merged;
- `average`: uses the average of the distances of each observation of the two sets;
- `complete`: uses the maximum of the distances between all observations of the two sets.

- `single`: uses the minimum of the distances between all observations of the two sets. Defaults to `"ward"`.

`distance_threshold` A numeric value specifying the linkage distance threshold above which clusters will not be merged. If not `NULL`, `n_clusters` must be `NULL` and `compute_full_tree` must be `TRUE`. Defaults to `NULL`.

`compute_distances` A boolean value specifying whether to compute distances between clusters even if `distance_threshold` is not used. This can be used to make dendrogram visualization, but introduces a computational and memory overhead. Defaults to `FALSE`.

Returns: An object of class AgglomerativeClustering.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AgglomerativeClustering$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- AgglomerativeClustering$new()
```

`AlphaComplex` *R6 Class for Alpha Complex*

Description

AlphaComplex is a simplicial complex constructed from the finite cells of a Delaunay Triangulation.

Details

The filtration value of each simplex is computed as the square of the circumradius of the simplex if the circumsphere is empty (the simplex is then said to be Gabriel), and as the minimum of the filtration values of the codimension 1 cofaces that make it not Gabriel otherwise. All simplices that have a filtration value strictly greater than a given alpha squared value are not inserted into the complex.

Super class

`rgudhi::PythonClass` -> `AlphaComplex`

Methods

Public methods:

- AlphaComplex$new()
- AlphaComplex$create_simplex_tree()
- AlphaComplex$get_point()
- AlphaComplex$clone()

Method `new()`**:** `AlphaComplex` constructor.

Usage:

```
AlphaComplex$new(points, precision = "safe")
```

Arguments:

`points` Either a `n x d` matrix or a length-`n` list of `d`-dimensional vectors or a file with extension `.off`.

`precision` A string specifying the alpha complex precision. Can be one of `"fast"`, `"safe"` or `"exact"`. Defaults to `"safe"`.

Returns: A `AlphaComplex` object storing the Alpha complex.

Method `create_simplex_tree()`**:** Generates a simplex tree from the Delaunay triangulation.

Usage:

```
AlphaComplex$create_simplex_tree(
  max_alpha_square = Inf,
  default_filtration_value = FALSE
)
```

Arguments:

`max_alpha_square` A numeric value specifying the maximum alpha square threshold the simplices shall not exceed. Default is set to `Inf`, and there is very little point using anything else since it does not save time.

`default_filtration_value` A boolean specifying whether filtration values should not be computed and will be set to `NaN` (`default_filtration_value = TRUE`). Defaults to `FALSE` (which means compute the filtration values).

Returns: A `SimplexTree` object storing the computed simplex tree.

Method `get_point()`**:** This function returns the point corresponding to a given vertex from the SimplexTree.

Usage:

```
AlphaComplex$get_point(vertex)
```

Arguments:

`vertex` An integer value specifying the desired vertex.

Returns: A numeric vector storing the point corresponding to the input vertex.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AlphaComplex$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Vincent Rouvreau

See Also

Other filtrations and reconstructions: RipsComplex, TangentialComplex, WitnessComplex

Examples

```
Xl <- seq_circle(10)
Xm <- Reduce(rbind, Xl, init = numeric())
acm <- AlphaComplex$new(points = Xm)
acl <- AlphaComplex$new(points = Xl)
acl
```

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
```

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
ac$get_point(1)
```

Atol *Vector Representation: Atol*

Description

Computes measure vectorization (e.g. point clouds, persistence diagrams, etc.) after a quantisation step according to the Atol algorithm (Royer et al. 2021).

References:

Royer M, Chazal F, Levrard C, Umeda Y, Ike Y (2021). "ATOL: measure vectorization for automatic topologically-oriented learning." In *International Conference on Artificial Intelligence and Statistics*, 1000–1008. PMLR.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::VectorRepresentationStep -> Atol

Methods

Public methods:

- `Atol$new()`
- `Atol$clone()`

Method `new()`**:** The `Atol` constructor.

Usage:

```
Atol$new(
  quantiser,
  weighting_method = c("cloud", "iidproba"),
  contrast = c("gaussian", "laplacian", "indicator")
)
```

Arguments:

`quantiser` An object of class `BaseClustering` specifying any clustering algorithm from the **sklearn.cluster** module. It will be fitted when the `$fit()` method is called.

`weighting_method` A string specifying the constant generic function for weighting the measure points. Choices are either `"cloud"` or `"iidproba"`. Defaults to `"cloud"`, i.e. the measure is seen as a point cloud. This will have no impact if weights are provided along with measures all the way, i.e. at `$fit()` and `$transform()` calls, through the optional argument `sample_weight`.

`contrast` A string specifying the constant function for evaluating proximity of a measure with respect to centers. Choices are either `"gaussian"` or `"laplacian"` or `"indicator"`. Defaults to `"gaussian"` (see page 3 in Royer et al. (2021)).

Returns: An object of class `Atol`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Atol$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
km <- KMeans$new(n_clusters = 2, random_state = 202006)
vr <- Atol$new(quantiser = km)
```

```
# vr$apply(dgm) # TODO: needs fix in python
vr$fit_transform(list(dgm))
```

`BettiCurve` *Vector Representation: Betti Curve*

Description

Computes Betti curves from persistence diagrams. There are several modes of operation: with a given resolution (with or without a `sample_range`), with a predefined grid, and with none of the previous. With a predefined grid, the class computes the Betti numbers at those grid points. Without a predefined grid, if the resolution is set to `NULL`, it can be fit to a list of persistence diagrams and produce a grid that consists of (at least) the filtration values at which at least one of those persistence diagrams changes Betti numbers, and then compute the Betti numbers at those grid points. In the latter mode, the exact Betti curve is computed for the entire real line. Otherwise, if the resolution is given, the Betti curve is obtained by sampling evenly using either the given `sample_range` or based on the persistence diagrams.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `BettiCurve`

Methods

Public methods:

- `BettiCurve$new()`
- `BettiCurve$clone()`

Method `new()`**:** The `BettiCurve` constructor.

Usage:

```
BettiCurve$new(
  resolution = 100,
  sample_range = rep(NA, 2),
  predefined_grid = NULL
)
```

Arguments:

`resolution` An integer value specifying the number of sample for the piecewise constant function. Defaults to `100L`.

`sample_range` A length-2 numeric vector specifying the minimum and maximum of the piecewise constant function domain, of the form $[x_{\min}, x_{\max}]$. Defaults to `rep(NA, 2)`. It is the interval on which samples will be drawn evenly. If one of the values is `NA`, it can be computed from the persistence diagrams with the `$fit()` method.

`predefined_grid` A numeric vector specifying a predefined grid of points at which to compute the Betti curves. Must be strictly ordered. Infinities are ok. If set to `NULL` (default), and resolution is given, the grid will be uniform from $x_{\min}$ to $x_{\max}$ in `resolution` steps, otherwise a grid will be computed that captures all changes in Betti numbers in the provided data.

Returns: An object of class `BettiCurve`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BettiCurve$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
bc <- BettiCurve$new()
bc$apply(dgm)
bc$fit_transform(list(dgm))
```

Birch *Performs clustering according to the Birch algorithm*

Description

It is a memory-efficient, online-learning algorithm provided as an alternative to MiniBatchKMeans. It constructs a tree data structure with the cluster centroids being read off the leaf. These can be either the final cluster centroids or can be provided as input to another clustering algorithm such as AgglomerativeClustering. This is a wrapper around the Python class sklearn.cluster.Birch.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `Birch`

Methods

Public methods:

- Birch$new()
- Birch$clone()

Method `new()`**:** The Birch class constructor.

Usage:

```
Birch$new(
  threshold = 0.5,
  branching_factor = 50L,
  n_clusters = 3L,
  compute_labels = TRUE,
  copy = TRUE
)
```

Arguments:

`threshold` A numeric value specifying the upper bound of the radius of the subcluster obtained by merging a new sample and the closest subcluster. Otherwise a new subcluster is started. Setting this value to be very low promotes splitting and vice-versa. Defaults to `0.5`.

`branching_factor` An integer value specifying the maximum number of CF subclusters in each node. If a new sample enters such that the number of subclusters exceeds the `branching_factor` then that node is splitted into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes.

`n_clusters` Either an integer value or an object of class BaseClustering specifying the number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples.

- `NULL`: the final clustering step is not performed and the subclusters are returned as they are;
- an object of class BaseClustering: the model is fit treating the subclusters as new samples and the initial data is mapped to the label of the closest subcluster;
- integer value: the model fit is AgglomerativeClustering with `n_clusters` set to be equal to the integer value. Defaults to `3L`.

`compute_labels` A boolean value specifying whether to compute labels for each fit. Defaults to `TRUE`.

`copy` A boolean value specifying whether to make a copy of the given data. If set to `FALSE`, the initial data will be overwritten. Defaults to `TRUE`.

Returns: An object of class Birch.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Birch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- Tian Zhang, Raghu Ramakrishnan, Maron Livny (1996). *BIRCH: An efficient data clustering method for large databases*, https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf.
- Roberto Perdisci J. *Birch - Java implementation of BIRCH clustering algorithm*, https://code.google.com/archive/p/jbirch.

Examples

```
cl <- Birch$new()
```

`BirthPersistenceTransform` *Preprocessing: Birth Persistence Transform*

Description

This is a class for the affine transformation $(x, y) \mapsto (x, y - x)$ to be applied on persistence diagrams.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::PreprocessingStep` -> `BirthPersistenceTransform`

Methods

Public methods:

- `BirthPersistenceTransform$new()`
- `BirthPersistenceTransform$clone()`

Method `new()`**:** The `BirthPersistenceTransform` constructor.

Usage:

```
BirthPersistenceTransform$new()
```

Returns: An object of class `BirthPersistenceTransform`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BirthPersistenceTransform$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
  ac <- AlphaComplex$new(points = X)
  st <- ac$create_simplex_tree()
  dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
  bpt <- BirthPersistenceTransform$new()
  bpt$apply(dgm)
  bpt$fit_transform(list(dgm))
```

BisectingKMeans *Performs clustering according to the bisecting k-means algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.BisectingKMeans.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `BisectingKMeans`

Methods

Public methods:

- `BisectingKMeans$new()`
- `BisectingKMeans$clone()`

Method `new()`**:** The BisectingKMeans class constructor.

Usage:

```
BisectingKMeans$new(
  n_clusters = 2L,
  init = c("k-means++", "random"),
  n_init = 10L,
  max_iter = 300L,
  tol = 1e-04,
  verbose = 0L,
  random_state = NULL,
  copy_x = TRUE,
  algorithm = c("lloyd", "elkan"),
  bisecting_strategy = c("biggest_inertia", "largest_cluster")
)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters to form as well as the number of centroids to generate. Defaults to `2L`.

`init` Either a string or a numeric matrix of shape $\mathrm{n_clusters} \times \mathrm{n_features}$ specifying the method for initialization. If a string, choices are:

- `"k-means++"`: selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contribution to the overall inertia. This technique speeds up convergence, and is theoretically proven to be $\mathcal{O}(\log(k))$-optimal. See the description of `n_init` for more details;
- `"random"`: chooses `n_clusters` observations (rows) at random from data for the initial centroids. Defaults to `"k-means++"`.

`n_init` An integer value specifying the number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of `n_init` consecutive runs in terms of inertia. Defaults to `10L`.

`max_iter` An integer value specifying the maximum number of iterations of the k-means algorithm for a single run. Defaults to `300L`.

`tol` A numeric value specifying the relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence. Defaults to `1e-4`.

`verbose` An integer value specifying the level of verbosity. Defaults to `0L` which is equivalent to no verbose.

`random_state` An integer value specifying the initial seed of the random number generator. Defaults to `NULL` which uses the current timestamp.

`copy_x` A boolean value specifying whether the original data is to be modified. When precomputing distances it is more numerically accurate to center the data first. If `copy_x` is `TRUE`, then the original data is not modified. If `copy_x` is `FALSE`, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean. Note that if the original data is not C-contiguous, a copy will be made even if `copy_x` is `FALSE`. If the original data is sparse, but not in CSR format, a copy will be made even if `copy_x` is `FALSE`. Defaults to `TRUE`.

`algorithm` A string specifying the k-means algorithm to use. The classical EM-style algorithm is `"lloyd"`. The `"elkan"` variation can be more efficient on some datasets with well-defined clusters, by using the triangle inequality. However it's more memory-intensive due to the allocation of an extra array of shape $\mathrm{n_samples} \times \mathrm{n_clusters}$. Defaults to `"lloyd"`.

`bisecting_strategy` A string specifying how bisection should be performed. Choices are:

- `"biggest_inertia"`: means that it will always check all calculated cluster for cluster with biggest SSE (Sum of squared errors) and bisect it. This approach concentrates on precision, but may be costly in terms of execution time (especially for larger amount of data points).
- `"largest_cluster"`: means that it will always split cluster with largest amount of points assigned to it from all clusters previously calculated. That should work faster than picking by SSE and may produce similar results in most cases. Defaults to `"biggest_inertia"`.

Returns: An object of class BisectingKMeans.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BisectingKMeans$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- BisectingKMeans$new()
```

`BottleneckDistance` *Metrics: Bottleneck Distance*

Description

Computes the bottleneck distance matrix from a list of persistence diagrams.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::MetricStep` -> `BottleneckDistance`

Methods

Public methods:

- `BottleneckDistance$new()`
- `BottleneckDistance$clone()`

Method `new()`**:** The `BottleneckDistance` constructor.

Usage:

```
BottleneckDistance$new(epsilon = NULL, n_jobs = 1)
```

Arguments:

`epsilon` A numeric value specifying the absolute (additive) error tolerated on the distance. Defaults to `NULL`, in which case the smallest positive float is used.

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1L`.

Returns: An object of class `BottleneckDistance`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BottleneckDistance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
dis <- BottleneckDistance$new()
dis$apply(dgm, dgm)
dis$fit_transform(list(dgm))
```

ComplexPolynomial *Vector Representation: Complex Polynomial*

Description

Computes complex polynomials from a list of persistence diagrams. The persistence diagram points are seen as the roots of some complex polynomial, whose coefficients are returned in a complex vector. See https://link.springer.com/chapter/10.1007%2F978-3-319-23231-7_27 for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `ComplexPolynomial`

Methods

Public methods:

- `ComplexPolynomial$new()`
- `ComplexPolynomial$clone()`

Method `new()`**:** The `ComplexPolynomial` constructor.

Usage:

```
ComplexPolynomial$new(polynomial_type = c("R", "S", "T"), threshold = 10)
```

Arguments:

`polynomial_type` A string specifying the Type of complex polynomial that is going to be computed (explained in https://link.springer.com/chapter/10.1007%2F978-3-319-23231-7_27). Choices are `c("R", "S", "T")`. Defaults to `"R"`.

`threshold` An integer value specifying the number of coefficients. This is the dimension of the complex vector of coefficients, i.e. the number of coefficients corresponding to the largest degree terms of the polynomial. If `-1`, this threshold is computed from the list of persistence diagrams by considering the one with the largest number of points and using the dimension of its corresponding complex vector of coefficients as threshold. Defaults to `10L`.

Returns: An object of class `ComplexPolynomial`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ComplexPolynomial$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
cp <- ComplexPolynomial$new()
cp$apply(dgm)
cp$fit_transform(list(dgm))
```

CubicalComplex *R6 Class for Cubical Complex*

Description

The CubicalComplex is an example of a structured complex useful in computational mathematics (specially rigorous numerics) and image analysis.

Super class

`rgudhi::PythonClass` -> `CubicalComplex`

Methods

Public methods:

- `CubicalComplex$new()`
- `CubicalComplex$betti_numbers()`
- `CubicalComplex$cofaces_of_persistence_pairs()`
- `CubicalComplex$compute_persistence()`
- `CubicalComplex$dimension()`
- `CubicalComplex$num_simplices()`
- `CubicalComplex$persistence()`

- CubicalComplex$persistence_intervals_in_dimension()
- CubicalComplex$persistent_betti_numbers()
- CubicalComplex$clone()

Method `new()`**:** Constructor from either `top_dimensional_cells` (and possibly `dimensions`) or from a Perseus-style file name.

Usage:

```
CubicalComplex$new(
  perseus_file,
  top_dimensional_cells,
  dimensions = NULL,
  py_class = NULL
)
```

Arguments:

`perseus_file` A character string specifying the path to a Perseus-style file name.

`top_dimensional_cells` Either a numeric vector (in which case, `dimensions` should be provided as well) or a multidimensional array specifying cell filtration values.

`dimensions` An integer vector specifying the number of top dimensional cells. Defaults to `NULL`.

`py_class` An existing `CubicalComplex` Python class. Defaults to `NULL` which uses the Python class constructor instead.

Returns: A new `CubicalComplex` object.

Method `betti_numbers()`**:** This function returns the Betti numbers of the complex.

Usage:

```
CubicalComplex$betti_numbers()
```

Details: The `$betti_numbers()` method always returns `[1, 0, 0, ...]` as infinity filtration cubes are not removed from the complex.

Returns: An integer vector storing the Betti numbers.

Method `cofaces_of_persistence_pairs()`**:** A persistence interval is described by a pair of cells, one that creates the feature and one that kills it. The filtration values of those 2 cells give coordinates for a point in a persistence diagram, or a bar in a barcode. Structurally, in the cubical complexes provided here, the filtration value of any cell is the minimum of the filtration values of the maximal cells that contain it. Connecting persistence diagram coordinates to the corresponding value in the input (i.e. the filtration values of the top-dimensional cells) is useful for differentiation purposes.

Usage:

```
CubicalComplex$cofaces_of_persistence_pairs()
```

Details: This function returns a list of pairs of top-dimensional cells corresponding to the persistence birth and death cells of the filtration. The cells are represented by their indices in the input list of top-dimensional cells (and not their indices in the internal data structure that includes non-maximal cells). Note that when two adjacent top-dimensional cells have the same filtration value, we arbitrarily return one of the two when calling the function on one of their common faces.

Returns: The top-dimensional cells/cofaces of the positive and negative cells, together with the corresponding homological dimension, in two lists of integer arrays. The first list contains the regular persistence pairs, grouped by dimension. It contains numpy arrays of shape `[number_of_persistence_points, 2]`. The indices of the arrays in the list correspond to the homological dimensions, and the integers of each row in each array correspond to: `(index of positive top-dimension`
The second list contains the essential features, grouped by dimension. It contains numpy arrays of shape `[number_of_persistence_points, 1]`. The indices of the arrays in the list correspond to the homological dimensions, and the integers of each row in each array correspond to: `(index of positive top-dimensional cell)`.

Method `compute_persistence()`**:** This method computes the persistence of the complex, so it can be accessed through `$persistent_betti_numbers()`, `$persistence_intervals_in_dimension()`, etc. It is equivalent to the `$persistence()` method when you do not want the list `$persistence()` returns.

Usage:

```
CubicalComplex$compute_persistence(
  homology_coeff_field = 11,
  min_persistence = 0
)
```

Arguments:

`homology_coeff_field` An integer value specifying the homology coefficient field. Must be a prime number. Defaults to `11L`. Maximum is `46337L`.

`min_persistence` A numeric value specifying the minimum persistence value to take into account (strictly greater than `min_persistence`). Defaults to `0.0`. Set `min_persistence = -1.0` to see all values.

Returns: The updated `CubicalComplex` class itself invisibly.

Method `dimension()`**:** This function returns the dimension of the complex.

Usage:

```
CubicalComplex$dimension()
```

Returns: An integer value giving the complex dimension.

Method `num_simplices()`**:** This function returns the number of all cubes in the complex.

Usage:

```
CubicalComplex$num_simplices()
```

Returns: An integer value giving the number of all cubes in the complex.

Method `persistence()`**:** This function computes and returns the persistence of the complex.

Usage:

```
CubicalComplex$persistence(homology_coeff_field = 11, min_persistence = 0)
```

Arguments:

`homology_coeff_field` An integer value specifying the homology coefficient field. Must be a prime number. Defaults to `11L`. Maximum is `46337L`.

`min_persistence` A numeric value specifying the minimum persistence value to take into account (strictly greater than `min_persistence`). Defaults to `0.0`. Set `min_persistence = -1.0` to see all values.

Returns: A tibble listing all persistence feature summarised by 3 variables: dimension, birth and death.

Method persistence_intervals_in_dimension(): This function returns the persistence intervals of the complex in a specific dimension.

Usage:

```
CubicalComplex$persistence_intervals_in_dimension(dimension)
```

Arguments:

dimension An integer value specifying the desired dimension.

Returns: A tibble storing the persistence intervals by row.

Method persistent_betti_numbers(): This function returns the persistent Betti numbers of the complex.

Usage:

```
CubicalComplex$persistent_betti_numbers(from_value, to_value)
```

Arguments:

from_value A numeric value specifying the persistence birth limit to be added in the numbers (persistent birth <= from_value).

to_value A numeric value specifying the persistence death limit to be added in the numbers (persistent death > to_value).

Returns: An integer vector storing the persistent Betti numbers.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
CubicalComplex$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Pawel Dlotko

See Also

Other data structures for cell complexes: PeriodicCubicalComplex, SimplexTree

Examples

```
n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc


n <- 10
```

```
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$compute_persistence()$betti_numbers()


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$compute_persistence()$cofaces_of_persistence_pairs()


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$dimension()


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$num_simplices()


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$persistence()


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$compute_persistence()$persistence_intervals_in_dimension(0)


n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
cc <- CubicalComplex$new(top_dimensional_cells = X)
cc$compute_persistence()$persistent_betti_numbers(0, 1)
```

DBSCAN *Performs clustering according to the DBSCAN algorithm*

Description

DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density. This is a wrapper around the Python class sklearn.cluster.DBSCAN.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::BaseClustering -> DBSCAN

Methods

Public methods:

- DBSCAN$new()
- DBSCAN$clone()

Method new(): The DBSCAN class constructor.

Usage:

```
DBSCAN$new(
  eps = 0.5,
  min_samples = 5L,
  metric = "euclidean",
  metric_params = NULL,
  algorithm = c("auto", "ball_tree", "kd_tree", "brute"),
  leaf_size = 30L,
  p = 2L,
  n_jobs = 1L
)
```

Arguments:

eps A numeric value specifying the maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function. Defaults to 0.5.

min_samples An integer value specifying the number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself. Defaults to 5L.

metric Either a string or an object coercible into a function via rlang::as_function() specifying the metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by sklearn.metrics.pairwise_distances for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse graph, in which case only *nonzero* elements may be considered neighbors for DBSCAN. Defaults to "euclidean".

metric_params A named list specifying additional parameters to be passed on to the metric function. Defaults to NULL.

algorithm A string specifying the algorithm to be used by the sklearn.neighbors.NearestNeighbors module to compute pointwise distances and find nearest neighbors. Choices are "auto", "ball_tree", "kd_tree" or "brute". Defaults to "auto".

leaf_size An integer value specifying the leaf size passed to sklearn.neighbors.BallTree or sklearn.neighbors.KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30L.

p An integer value specifying the power of the Minkowski metric to be used to calculate distance between points. Defaults to 2L.

`n_jobs` An integer value specifying the number of parallel jobs to run. Defaults to `1L`.

Returns: An object of class DBSCAN.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DBSCAN$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

- Ester, M., H. P. Kriegel, J. Sander, and X. Xu (1996). *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*, In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231.
- Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). *DBSCAN revisited, revisited: why and how you should (still) use DBSCAN*, ACM Transactions on Database Systems (TODS), **42**(3), p. 19.

Examples

```
cl <- DBSCAN$new()
```

`DiagramScaler` *Preprocessing: Diagram Scaler*

Description

This is a class for preprocessing persistence diagrams with a given list of scalers, such as those included in **scikit-learn**.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::PreprocessingStep` -> `DiagramScaler`

Methods

Public methods:

- `DiagramScaler$new()`
- `DiagramScaler$clone()`

Method `new()`**:** The `DiagramScaler` constructor.

Usage:

```
DiagramScaler$new(use = FALSE, scalers = list())
```

Arguments:

`use` A boolean value specifying whether to use the class. Defaults to `FALSE`.

`scalers` A list of scalers to be fit on the persistence diagrams. Defaults to `list()` which is an empty list. Each element of the list is a length-2 base::list with two elements:

- the first one is a list of coordinates;
- the second one is an instantiated scaler class. Choices are MaxAbsScaler, MinMaxScaler, RobustScaler or StandardScaler.

Returns: An object of class `DiagramScaler`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DiagramScaler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramScaler$new()
ds$apply(dgm)
ds$fit_transform(list(dgm))
```

DiagramSelector *Preprocessing: Diagram Selector*

Description

This is a class for extracting finite or essential points in persistence diagrams.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::PreprocessingStep` -> `DiagramSelector`

Methods

Public methods:

- DiagramSelector$new()
- DiagramSelector$clone()

Method `new()`**:** The DiagramSelector constructor.

Usage:

```
DiagramSelector$new(
  use = FALSE,
  limit = Inf,
  point_type = c("finite", "essential")
)
```

Arguments:

`use` A boolean value specifying whether to use the class. Defaults to `FALSE`.

`limit` A numeric value specifying the second coordinate value which is the criterion for being an essential point. Defaults to ∞.

`point_type` A string specifying the type of the points that are going to be extracted. Choices are either "`finite`" or "`essential`". Defaults to "`finite`".

Returns: An object of class DiagramSelector.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DiagramSelector$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new()
ds$apply(dgm)
ds$fit_transform(list(dgm))
```

Entropy *Vector Representation: Entropy*

Description

Computes persistence entropy. Persistence entropy is a statistic for persistence diagrams inspired from Shannon entropy. This statistic can also be used to compute a feature vector, called the entropy summary function. See https://arxiv.org/pdf/1803.08304.pdf for more details. Note that a previous implementation was contributed by Manuel Soriano-Trigueros.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `Entropy`

Methods

Public methods:

- `Entropy$new()`
- `Entropy$clone()`

Method `new()`**:** The `Entropy` constructor.

Usage:

```
Entropy$new(
  mode = c("scalar", "vector"),
  normalized = TRUE,
  resolution = 100,
  sample_range = rep(NA_real_, 2)
)
```

Arguments:

`mode` A string specifying which entropy to compute: either `"scalar"` for computing the entropy statistic, or `"vector"` for computing the entropy summary function. Defaults to `"scalar"`.

`normalized` A boolean value specifying whether to normalize the entropy summary function. Defaults to `TRUE`. Used only if `mode == "vector"`.

`resolution` An integer value specifying the grid size for the entropy summary function. Defaults to `100L`. Used only if `mode == "vector"`.

`sample_range` A length-2 numeric vector specifying the domain for the entropy summary function, of the form $[x_{\min}, x_{\max}]$. Defaults to `rep(NA, 2)`. It is the interval on which samples will be drawn evenly. If one of the values is `NA`, it can be computed from the persistence diagrams with the `$fit()` method. Used only if `mode == "vector"`.

Returns: An object of class `Entropy`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Entropy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
ent <- Entropy$new()
ent$apply(dgm)
ent$fit_transform(list(dgm))
```

FeatureAgglomeration *Performs clustering according to the feature agglomeration algorithm*

Description

Recursively merges pair of clusters of features. This is a wrapper around the Python class sklearn.cluster.FeatureAgglomeratio

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `FeatureAgglomeration`

Methods

Public methods:

- `FeatureAgglomeration$new()`
- `FeatureAgglomeration$clone()`

Method `new()`**:** The FeatureAgglomeration class constructor.

Usage:

```
FeatureAgglomeration$new(
  n_clusters = 2L,
  affinity = c("euclidean", "l1", "l2", "manhattan", "cosine", "precomputed"),
  memory = NULL,
  connectivity = NULL,
  compute_full_tree = "auto",
  linkage = c("ward", "complete", "average", "single"),
  pooling_func = rowMeans,
  distance_threshold = NULL,
  compute_distances = FALSE
)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters to find. Defaults to `2L`.

`affinity` A string or an object coercible into a function via `rlang::as_function()` specifying the metric used to compute the linkage. If a string, choices are `"euclidean"`, `"l1"`, `"l2"`, `"manhattan"`, `"cosine"` or `"precomputed"`. If linkage is `"ward"`, only `"euclidean"` is accepted. Defaults to `"euclidean"`.

`memory` A string specifying path to the caching directory for storing the computation of the tree. Defaults to `NULL` in which case no caching is done.

`connectivity` A numeric matrix or an object coercible into a function via `rlang::as_function()` specifying the connectivity matrix. Defines for each feature the neighboring features following a given structure of the data. This can be a connectivity matrix itself or a function that transforms the data into a connectivity matrix, such as derived from sklearn.neighbors.kneighbors_graph(). Defaults to `NULL` in which case the hierarchical clustering algorithm is unstructured.

`compute_full_tree` The string `"auto"` or a boolean value specifying whether to stop early the construction of the tree at `n_clusters`. This is useful to decrease computation time if the number of clusters is not small compared to the number of features. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `TRUE` if `distance_threshold` is not `NULL`. Defaults to `"auto"`, which is equivalent to `TRUE` when `distance_threshold` is not `NULL` or when `n_clusters` is inferior to `max(100, 0.02 * n_samples)` and to `FALSE` otherwise.

`linkage` A string specifying which linkage criterion to use. The linkage criterion determines which distance to use between sets of features. The algorithm will merge the pairs of cluster that minimize this criterion:

- `"ward"`: minimizes the variance of the clusters being merged;
- `"complete"`: maximum linkage uses the maximum distances between all features of the two sets;
- `"average"`: uses the average of the distances of each feature of the two sets;
- `"single"`: uses the minimum of the distances between all features of the two sets.

`pooling_func` An object coercible into a function via `rlang::as_function()` specifying the aggregation method to combine the values of agglomerated features into a single value. It should take as input an array of shape $M \times N$ and the optional argument `axis = 1`, and reduce it to an array of shape M. Defaults to base::rowMeans.

`distance_threshold` A numeric value specifying the linkage distance threshold above which clusters will not be merged. If not `NULL`, `n_clusters` must be `NULL` and `compute_full_tree` must be `TRUE`. Defaults to `NULL`.

`compute_distances` A boolean value specifying whether to compute distances between clusters even if `distance_threshold` is not used. This can be used to make dendrogram visualization, but introduces a computational and memory overhead. Defaults to `FALSE`.

Returns: An object of class FeatureAgglomeration.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FeatureAgglomeration$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- FeatureAgglomeration$new()
```

`fetch` *Remote Data Sets*

Description

A collection of function for fetching famous remote data sets.

Usage

```
fetch_bunny(data_folder, accept_license = FALSE)

fetch_spiral_2d(data_folder)

clear_data_home(data_folder)
```

Arguments

`data_folder`	A string specifying a location for storing data ot be used with GUDHI.
`accept_license`	A boolean specifying whether the user accepts the file `LICENSE` and prevents from printing the corresponding license terms. Defaults to `FALSE`.

Value

A numeric array storing the points of the corresponding data set.

Stanford bunny dataset

The `fetch_bunny()` function returns a numeric array of shape 35947×3.

`spiral_2d` dataset

The `fetch_spiral_2d()` function returns a numeric array of shape $114, 562 \times 2$.

Examples

```
b <- withr::with_tempdir({fetch_bunny(getwd())})
s <- withr::with_tempdir({fetch_spiral_2d(getwd())})
```

KMeans *Performs clustering according to the k-means algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.KMeans.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `KMeans`

Methods

Public methods:

- `KMeans$new()`
- `KMeans$clone()`

Method `new()`**:** The KMeans class constructor.

Usage:

```
KMeans$new(
  n_clusters = 2L,
  init = c("k-means++", "random"),
  n_init = 10L,
  max_iter = 300L,
  tol = 1e-04,
  verbose = 0L,
  random_state = NULL,
  copy_x = TRUE,
  algorithm = c("lloyd", "elkan")
)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters to form as well as the number of centroids to generate. Defaults to `2L`.

`init` Either a string or a numeric matrix of shape $\mathrm{n}_{\mathrm{clusters}} \times \mathrm{n}_{\mathrm{features}}$ specifying the method for initialization. If a string, choices are:

- `"k-means++"`: selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contribution to the overall inertia. This technique speeds up convergence, and is theoretically proven to be $\mathcal{O}(\log(k))$-optimal. See the description of `n_init` for more details;
- `"random"`: chooses `n_clusters` observations (rows) at random from data for the initial centroids.

Defaults to `"k-means++"`.

`n_init` An integer value specifying the number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of `n_init` consecutive runs in terms of inertia. Defaults to `10L`.

`max_iter` An integer value specifying the maximum number of iterations of the k-means algorithm for a single run. Defaults to `300L`.

`tol` A numeric value specifying the relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence. Defaults to `1e-4`.

`verbose` An integer value specifying the level of verbosity. Defaults to `0L` which is equivalent to no verbose.

`random_state` An integer value specifying the initial seed of the random number generator. Defaults to `NULL` which uses the current timestamp.

`copy_x` A boolean value specifying whether the original data is to be modified. When precomputing distances it is more numerically accurate to center the data first. If `copy_x` is `TRUE`, then the original data is not modified. If `copy_x` is `FALSE`, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean. Note that if the original data is not C-contiguous, a copy will be made even if `copy_x` is `FALSE`. If the original data is sparse, but not in CSR format, a copy will be made even if `copy_x` is `FALSE`. Defaults to `TRUE`.

`algorithm` A string specifying the k-means algorithm to use. The classical EM-style algorithm is `"lloyd"`. The `"elkan"` variation can be more efficient on some datasets with well-defined clusters, by using the triangle inequality. However it's more memory-intensive due to the allocation of an extra array of shape $\mathrm{n}_{\mathrm{samples}} \times \mathrm{n}_{\mathrm{clusters}}$. Defaults to `"lloyd"`.

Returns: An object of class KMeans.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
KMeans$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- KMeans$new()
```

`Landscape` *Vector Representation: Landscape*

Description

Computes persistence landscapes from a list of persistence diagrams. A persistence landscape is a collection of 1D piecewise-linear functions computed from the rank function associated to the persistence diagram. These piecewise-linear functions are then sampled evenly on a given range and the corresponding vectors of samples are concatenated and returned. See http://jmlr.org/papers/v16/bubenik15a.html for more details.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::VectorRepresentationStep -> Landscape

Methods

Public methods:

- Landscape$new()
- Landscape$clone()

Method new(): The Landscape constructor.

Usage:

```
Landscape$new(
  num_landscapes = 5,
  resolution = 100,
  sample_range = rep(NA_real_, 2)
)
```

Arguments:

num_landscapes An integer value specifying the number of piecewise linear functions to output. Defaults to 5L.

resolution An integer value specifying the grid size for the landscapes. Defaults to 100L.

sample_range A length-2 numeric vector specifying the domain for the entropy summary function, of the form $[x_{\min}, x_{\max}]$. Defaults to rep(NA, 2). It is the interval on which samples will be drawn evenly. If one of the values is NA, it can be computed from the persistence diagrams with the $fit() method.

Returns: An object of class Landscape.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Landscape$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
```

```
lds <- Landscape$new()
lds$apply(dgm)
lds$fit_transform(list(dgm))
```

`MaxAbsScaler` *Scales each feature by its maximum absolute value*

Description

This estimator scales and translates each feature individually such that the maximal absolute value of each feature in the training set will be 1.0. It does not shift/center the data, and thus does not destroy any sparsity.

This scaler can also be applied to sparse CSR or CSC matrices.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseScaler` -> `MaxAbsScaler`

Methods

Public methods:

- `MaxAbsScaler$new()`
- `MaxAbsScaler$clone()`

Method `new()`**:** The MaxAbsScaler class constructor.

Usage:

```
MaxAbsScaler$new(copy = TRUE)
```

Arguments:

`copy` A boolean value specifying whether to perform in-place scaling and avoid a copy (if the input is already a numpy array). Defaults to `TRUE`.

Returns: An object of class MaxAbsScaler.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MaxAbsScaler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
mas <- MaxAbsScaler$new()
```

MeanShift *Performs clustering according to the mean shift algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.MeanShift.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `MeanShift`

Methods

Public methods:

- `MeanShift$new()`
- `MeanShift$clone()`

Method `new()`**:** The MeanShift class constructor.

Usage:

```
MeanShift$new(
  bandwidth = NULL,
  seeds = NULL,
  bin_seeding = FALSE,
  min_bin_freq = 1L,
  cluster_all = TRUE,
  n_jobs = 1L,
  max_iter = 300L
)
```

Arguments:

`bandwidth` A numeric value specifying the bandwidth used in the RBF kernel. If `NULL`, the bandwidth is estimated using sklearn.cluster.estimate_bandwidth(). Defaults to `NULL`.

`seeds` A numeric matrix of shape $n_{\text{samples}} \times n_{\text{features}}$ specifying the seeds used to initialize kernels. If `NULL`, the seeds are calculated by `sklearn.cluster.get_bin_seeds()` with bandwidth as the grid size and default values for other parameters. Defaults to `NULL`.

`bin_seeding` A boolean value specifying whether initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to `TRUE` will speed up the algorithm because fewer seeds will be initialized. Defaults to `FALSE`. Ignored if `seeds` is not `NULL`.

`min_bin_freq` An integer value specifying the minimal size of bins. To speed up the algorithm, accept only those bins with at least `min_bin_freq` points as seeds. Defaults to `1L`.

`cluster_all` A boolean value specifying whether all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If `FALSE`, then orphans are given cluster label `-1`. Defaults to `TRUE`.

`n_jobs` An integer value specifying the number of jobs to use for the computation. This works by computing each of the `n_init` runs in parallel. Defaults to `1L`.

`max_iter` An integer value specifying the maximum number of iterations per seed point before the clustering operation terminates (for that seed point) if it has not yet converged. Defaults to `300L`.

Returns: An object of class MeanShift.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MeanShift$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- MeanShift$new()
```

`MiniBatchKMeans` *Performs clustering according to the mini-batch k-means algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.MiniBatchKMeans.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `MiniBatchKMeans`

Methods

Public methods:

- `MiniBatchKMeans$new()`
- `MiniBatchKMeans$clone()`

Method `new()`**:** The MiniBatchKMeans class constructor.

Usage:

```
MiniBatchKMeans$new(
  n_clusters = 2L,
  init = c("k-means++", "random"),
  n_init = 10L,
  max_iter = 300L,
  tol = 1e-04,
  verbose = 0L,
```

```
  random_state = NULL,
  batch_size = 1024L,
  compute_labels = TRUE,
  max_no_improvement = 10L,
  init_size = NULL,
  reassignment_ratio = 0.01
)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters to form as well as the number of centroids to generate. Defaults to `2L`.

`init` Either a string or a numeric matrix of shape $\mathrm{n_clusters} \times \mathrm{n_features}$ specifying the method for initialization. If a string, choices are:

- `"k-means++"`: selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contribution to the overall inertia. This technique speeds up convergence, and is theoretically proven to be $\mathcal{O}(\log(k))$-optimal. See the description of `n_init` for more details;
- `"random"`: chooses `n_clusters` observations (rows) at random from data for the initial centroids. Defaults to `"k-means++"`.

`n_init` An integer value specifying the number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of `n_init` consecutive runs in terms of inertia. Defaults to `10L`.

`max_iter` An integer value specifying the maximum number of iterations of the k-means algorithm for a single run. Defaults to `300L`.

`tol` A numeric value specifying the relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence. Defaults to `1e-4`.

`verbose` An integer value specifying the level of verbosity. Defaults to `0L` which is equivalent to no verbose.

`random_state` An integer value specifying the initial seed of the random number generator. Defaults to `NULL` which uses the current timestamp.

`batch_size` An integer value specifying the size of the mini-batches. For faster computations, you can set the `batch_size` greater than 256 * number of cores to enable parallelism on all cores. Defaults to `1024L`.

`compute_labels` A boolean value specifying whether to compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit. Defaults to `TRUE`.

`max_no_improvement` An integer value specifying how many consecutive mini batches that does not yield an improvement on the smoothed inertia should be used to call off the algorithm. To disable convergence detection based on inertia, set `max_no_improvement` to `NULL`. Defaults to `10L`.

`init_size` An integer value specifying the number of samples to randomly sample for speeding up the initialization (sometimes at the expense of accuracy): the only algorithm is initialized by running a batch KMeans on a random subset of the data. This needs to be larger than `n_clusters`. If `NULL`, the heuristic is `init_size = 3 * batch_size` if `3 * batch_size < n_clusters`, else `init_size = 3 * n_clusters`. Defaults to `NULL`.

`reassignment_ratio` A numeric value specifying the fraction of the maximum number of counts for a center to be reassigned. A higher value means that low count centers are more

easily reassigned, which means that the model will take longer to converge, but should converge in a better clustering. However, too high a value may cause convergence issues, especially with a small batch size. Defaults to `0.01`.

Returns: An object of class MiniBatchKMeans.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MiniBatchKMeans$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- MiniBatchKMeans$new()
```

`MinMaxScaler` *Transforms features by scaling each feature to a given range*

Description

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where `min, max = feature_range`.

This transformation is often used as an alternative to zero mean, unit variance scaling.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseScaler` -> `MinMaxScaler`

Methods

Public methods:

- `MinMaxScaler$new()`
- `MinMaxScaler$clone()`

Method `new()`**:** The MinMaxScaler class constructor.

Usage:

```
MinMaxScaler$new(feature_range = c(0, 1), copy = TRUE, clip = FALSE)
```

Arguments:

`feature_range` A length-2 numeric vector specifying the desired range of transformed data. Defaults to `c(0, 1)`.

`copy` A boolean value specifying whether to perform in-place scaling and avoid a copy (if the input is already a numpy array). Defaults to `TRUE`.

`clip` A boolean value specifying whether to clip transformed values of held-out data to provided `feature_range`. Defaults to `FALSE`.

Returns: An object of class MinMaxScaler.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MinMaxScaler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
mms <- MinMaxScaler$new()
```

`OPTICS` *Performs clustering according to the OPTICS algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.OPTICS.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `OPTICS`

Methods

Public methods:

- `OPTICS$new()`
- `OPTICS$clone()`

Method `new()`**:** The OPTICS class constructor.

Usage:

```
OPTICS$new(
  min_samples = 5L,
  max_eps = Inf,
 metric = c("minkowski", "cityblock", "cosine", "euclidean", "l1", "l2", "manhattan",
  "braycurtis", "canberra", "chebyshev", "correlation", "dice", "hamming", "jaccard",
```

```
    "kulsinski", "mahalanobis", "rogerstanimoto", "russellrao", "seuclidean",
    "sokalmichener", "sokalsneath", "sqeuclidean", "yule"),
  p = 2L,
  metric_params = NULL,
  cluster_method = c("xi", "dbscan"),
  eps = NULL,
  xi = 0.05,
  predecessor_correction = TRUE,
  min_cluster_size = NULL,
  algorithm = c("auto", "ball_tree", "kd_tree", "brute"),
  leaf_size = 30L,
  memory = NULL,
  n_jobs = 1L
)
```

Arguments:

`min_samples` Either an integer value greater than 1 or a numeric value between 0 and 1 specifying the number of samples in a neighborhood for a point to be considered as a core point. Also, up and down steep regions can't have more than `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). Defaults to `5L`.

`max_eps` A numeric value specifying the maximum distance between two samples for one to be considered as in the neighborhood of the other. Reducing `max_eps` will result in shorter run times. Defaults to `Inf`.

`metric` Either a string or an object coercible into a function via `rlang::as_function()` specifying the metric to use for distance computation. If `metric` is a function, it is called on each pair of instances (rows) and the resulting value recorded. The function should take two numeric vectors as input and return one numeric value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string. If metric is `"precomputed"`, `X` is assumed to be a distance matrix and must be square. Valid string values for metric are:

- from **sklearn.metrics**: `"cityblock"`, `"cosine"`, `"euclidean"`, `"l1"`, `"l2"`, `"manhattan"`;
- from **scipy.spatial.distance**: `"braycurtis"`, `"canberra"`, `"chebyshev"`, `"correlation"`, `"dice"`, `"hamming"`, `"jaccard"`, `"kulsinski"`, `"mahalanobis"`, `"minkowski"`, `"rogerstanimoto"`, `"russellrao"`, `"seuclidean"`, `"sokalmichener"`, `"sokalsneath"`, `"sqeuclidean"`, `"yule"`.

Defaults to `"minkowski"`.

`p` An integer value specifying the power for the Minkowski metric. When `p = 1`, this is equivalent to using the Manhattan distance (ℓ_1). When `p = 2`, this is equivalent to using the Euclidean distance (ℓ_2). For arbitrary p, the Minkowski distance (ℓ_p) is used. Defaults to `2L`.

`metric_params` A named list specifying additional arguments for the metric function. Defaults to `NULL`.

`cluster_method` A string specifying the extraction method used to extract clusters using the calculated reachability and ordering. Possible values are `"xi"` and `"dbscan"`. Defaults to `"xi"`.

`eps` A numeric value specifying the maximum distance between two samples for one to be considered as in the neighborhood of the other. Defaults to `max_eps`. Used only when

`cluster_method == "dbscan"`.

`xi` A numeric value in $[0, 1]$ specifying the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most `1 - xi`. Used only when `cluster_method == "xi"`. Defaults to `0.05`.

`predecessor_correction` A boolean value specifying whether to correct clusters according to the predecessors calculated by OPTICS (Schubert and Gertz 2018). This parameter has minimal effect on most data sets. Used only when `cluster_method == "xi"`. Defaults to `TRUE`.

`min_cluster_size` Either an integer value > 1 or a numeric value in $[0, 1]$ specifying the minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `NULL`, the value of `min_samples` is used instead. Used only when `cluster_method == "xi"`. Defaults to `NULL`.

`algorithm` A string specifying the algorithm used to compute the nearest neighbors. Choices are `c("auto", "ball_tree", "kd_tree", "brute")`. Defaults to `"auto"` which will attempt to decide the most appropriate algorithm based on the values passed to fit method. Note: fitting on sparse input will override the setting of this parameter, using `algorithm == "brute"`.

`leaf_size` An integer value specifying the leaf size passed to `BallTree` or `KDTree`. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to `30L`

`memory` A string specifying the path to the caching directory into which caching the output of the computation of the tree. Defaults to `NULL` in which case no caching is done.

`n_jobs` An integer value specifying the number of parallel jobs to run for neighbors search. Defaults to `1L`. A value of `-1L` means using all processors.

Returns: An object of class OPTICS.

References:

Schubert E, Gertz M (2018). "Improving the cluster structure extracted from optics plots." In *LWDA*.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
OPTICS$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- OPTICS$new()
```

Padding *Preprocessing: Padding*

Description

This is a class for padding a list of persistence diagrams with dummy points, so that all persistence diagrams end up with the same number of points.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::PreprocessingStep` -> `Padding`

Methods

Public methods:

- `Padding$new()`
- `Padding$clone()`

Method `new()`**:** The `Padding` constructor.

Usage:

```
Padding$new(use = FALSE)
```

Arguments:

`use` A boolean value specifying whether to use the class. Defaults to `FALSE`.

Returns: An object of class `Padding`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Padding$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
pad <- Padding$new()
pad$apply(dgm)
pad$fit_transform(list(dgm))
```

PeriodicCubicalComplex

R6 Class for Periodic Cubical Complex

Description

The PeriodicCubicalComplex class is an example of a structured complex useful in computational mathematics (specially rigorous numerics) and image analysis.

Super classes

rgudhi::PythonClass -> rgudhi::CubicalComplex -> PeriodicCubicalComplex

Methods

Public methods:

- PeriodicCubicalComplex$new()
- PeriodicCubicalComplex$clone()

Method new(): Constructor from either top_dimensional_cells (and possibly dimensions) or from a Perseus-style file name.

Usage:

```
PeriodicCubicalComplex$new(
  perseus_file,
  top_dimensional_cells,
  periodic_dimensions,
  dimensions = NULL,
  py_class = NULL
)
```

Arguments:

perseus_file A character string specifying the path to a Perseus-style file name.

top_dimensional_cells Either a numeric vector (in which case, dimensions should be provided as well) or a multidimensional array specifying cell filtration values.

periodic_dimensions A logical vector specifying the periodicity value of the top dimensional cells.

dimensions An integer vector specifying the number of top dimensional cells. Defaults to NULL.

py_class An existing PeriodicCubicalComplex Python class. Defaults to NULL which uses the Python class constructor instead.

Returns: A new PeriodicCubicalComplex object.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
PeriodicCubicalComplex$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Pawel Dlotko

See Also

Other data structures for cell complexes: CubicalComplex, SimplexTree

Examples

```
n <- 10
X <- cbind(seq(0, 1, len = n), seq(0, 1, len = n))
pcc <- PeriodicCubicalComplex$new(
  top_dimensional_cells = X,
  periodic_dimensions = c(TRUE, FALSE)
)
pcc
```

PersistenceFisherDistance

Metrics: Persistence Fisher Distance

Description

Computes the persistence Fisher distance matrix from a list of persistence diagrams. The persistence Fisher distance is obtained by computing the original Fisher distance between the probability distributions associated to the persistence diagrams given by convolving them with a Gaussian kernel. See http://papers.nips.cc/paper/8205-persistence-fisher-kernel-a-riemannian-manifold-kernel-for-persistence-diagrams for more details.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::MetricStep -> PersistenceFisherDistance

Methods

Public methods:

- PersistenceFisherDistance$new()
- PersistenceFisherDistance$clone()

Method new(): The PersistenceFisherDistance constructor.

Usage:

```
PersistenceFisherDistance$new(bandwidth = 1, kernel_approx = NULL, n_jobs = 1)
```

Arguments:

bandwidth A numeric value specifying the bandwidth of the Gaussian kernel applied to the persistence Fisher distance. Defaults to 1.0.

`kernel_approx` A Python class specifying the kernel approximation class used to speed up computation. Defaults to `NULL`. Common kernel approximations classes can be found in the **scikit-learn** library (such as `RBFSampler` for instance).

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1L`.

Returns: An object of class `PersistenceFisherDistance`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceFisherDistance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
dis <- PersistenceFisherDistance$new()
dis$apply(dgm, dgm)
dis$fit_transform(list(dgm))
```

PersistenceFisherKernel

Kernel Representation: Persistence Fisher Kernel

Description

Computes the persistence Fisher kernel matrix from a list of persistence diagrams. The persistence Fisher kernel is computed by exponentiating the corresponding persistence Fisher distance with a Gaussian kernel. See papers.nips.cc/paper/8205-persistence-fisher-kernel-a-riemannian-manifold-kernel-for-persistence-diagrams for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::KernelRepresentationStep` -> `PersistenceFisherKernel`

Methods

Public methods:

- PersistenceFisherKernel$new()
- PersistenceFisherKernel$clone()

Method `new()`**:** The `PersistenceFisherKernel` constructor.

Usage:

```
PersistenceFisherKernel$new(
  bandwidth_fisher = 1,
  bandwidth = 1,
  kernel_approx = NULL,
  n_jobs = 1
)
```

Arguments:

`bandwidth_fisher` A numeric value specifying the bandwidth of the Gaussian kernel used to turn persistence diagrams into probability distributions by the `PersistenceFisherDistance` class. Defaults to `1.0`.

`bandwidth` A numeric value specifying the bandwidth of the Gaussian kernel applied to the persistence Fisher distance. Defaults to `1.0`.

`kernel_approx` A Python class specifying the kernel approximation class used to speed up computation. Defaults to `NULL`. Common kernel approximations classes can be found in the **scikit-learn** library (such as `RBFSampler` for instance).

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1`.

Returns: An object of class `PersistenceFisherKernel`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceFisherKernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
pfk <- PersistenceFisherKernel$new()
```

```
pfk$apply(dgm, dgm)
pfk$fit_transform(list(dgm))
```

`PersistenceImage` *Vector Representation: Persistence Image*

Description

Computes persistence images from a list of persistence diagrams. A persistence image is a 2D function computed from a persistence diagram by convolving the diagram points with a weighted Gaussian kernel. The plane is then discretized into an image with pixels, which is flattened and returned as a vector. See http://jmlr.org/papers/v18/16-337.html for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `PersistenceImage`

Methods

Public methods:

- `PersistenceImage$new()`
- `PersistenceImage$clone()`

Method `new()`**:** The `PersistenceImage` constructor.

Usage:

```
PersistenceImage$new(
  bandwidth = 1,
  weight = ~1,
  resolution = c(20, 20),
  im_range = rep(NA_real_, 4)
)
```

Arguments:

`bandwidth` A numeric value specifying the bandwidth of the Gaussian kernel. Defaults to `1.0`.

`weight` A function or a formula coercible into a function via `rlang::as_function()` specifying the weight function for the persistence diagram points. Defaults to the constant function `~ 1`. This function must be defined on 2D points, i.e. lists or arrays of the form $[p_x, p_y]$.

`resolution` An length-1 integer vector specifying the size (in pixels) of the persistence image. Defaults to `rep(20L, 2)`.

`im_range` A length-4 numeric vector specifying the two-dimensional domain for the persistence image, of the form $[x_{\min}, y_{\min}, x_{\max}, y_{\max}]$. Defaults to `rep(NA, 4)`. If one of the values is `NA`, it can be computed from the persistence diagrams with the `$fit()` method.

Returns: An object of class `PersistenceImage`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceImage$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
pei <- PersistenceImage$new()
pei$apply(dgm)
pei$fit_transform(list(dgm))
```

PersistenceScaleSpaceKernel

Kernel Representation: Persistence Scale-Space Kernel

Description

Computes the persistence scale space kernel matrix from a list of persistence diagrams. The persistence scale space kernel is computed by adding the symmetric to the diagonal of each point in each persistence diagram, with negative weight, and then convolving the points with a Gaussian kernel. See https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Reininghaus_A_Stable_Multi-Scale_2015_CVPR_paper.pdf for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::KernelRepresentationStep` -> `PersistenceScaleSpaceKernel`

Methods

Public methods:

- `PersistenceScaleSpaceKernel$new()`
- `PersistenceScaleSpaceKernel$clone()`

Method `new()`**:** The `PersistenceScaleSpaceKernel` constructor.

Usage:

```
PersistenceScaleSpaceKernel$new(
  bandwidth = 1,
  kernel_approx = NULL,
  n_jobs = 1
)
```

Arguments:

`bandwidth` A numeric value specifying the bandwidth of the Gaussian kernel with which persistence diagrams will be convolved. Defaults to `1.0`.

`kernel_approx` A Python class specifying the kernel approximation class used to speed up computation. Defaults to `NULL`. Common kernel approximations classes can be found in the **scikit-learn** library (such as `RBFSampler` for instance).

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1`.

Returns: An object of class `PersistenceScaleSpaceKernel`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceScaleSpaceKernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
pssk <- PersistenceScaleSpaceKernel$new()
pssk$apply(dgm, dgm)
pssk$fit_transform(list(dgm))
```

PersistenceSlicedWassersteinKernel

Kernel Representation: Persistence Sliced Wasserstein Kernel

Description

Computes the sliced Wasserstein kernel matrix from a list of persistence diagrams. The sliced Wasserstein kernel is computed by exponentiating the corresponding sliced Wasserstein distance with a Gaussian kernel. See http://proceedings.mlr.press/v70/carriere17a.html for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::KernelRepresentationStep` -> `PersistenceSlicedWassersteinKernel`

Methods

Public methods:

- `PersistenceSlicedWassersteinKernel$new()`
- `PersistenceSlicedWassersteinKernel$clone()`

Method `new()`**:** The `PersistenceSlicedWassersteinKernel` constructor.

Usage:

```
PersistenceSlicedWassersteinKernel$new(
  num_directions = 10,
  bandwidth = 1,
  n_jobs = 1
)
```

Arguments:

`num_directions` An integer value specifying the number of lines evenly sampled from $[-\pi/2, \pi/2]$ in order to approximate and speed up the kernel computation. Defaults to `10L`.

`bandwidth` A numeric value specifying the bandwidth of the Gaussian kernel with which persistence diagrams will be convolved. Defaults to `1.0`.

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1`.

Returns: An object of class `PersistenceSlicedWassersteinKernel`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceSlicedWassersteinKernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
pswk <- PersistenceSlicedWassersteinKernel$new()
pswk$apply(dgm, dgm)
pswk$fit_transform(list(dgm))
```

PersistenceWeightedGaussianKernel

Kernel Representation: Persistence Weighted Gaussian Kernel

Description

Computes the persistence weighted Gaussian kernel matrix from a list of persistence diagrams. The persistence weighted Gaussian kernel is computed by convolving the persistence diagram points with weighted Gaussian kernels. See http://proceedings.mlr.press/v48/kusano16.html for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::KernelRepresentationStep` -> `PersistenceWeightedGaussianKernel`

Methods

Public methods:

- `PersistenceWeightedGaussianKernel$new()`
- `PersistenceWeightedGaussianKernel$clone()`

Method `new()`**:** The `PersistenceWeightedGaussianKernel` constructor.

Usage:

```
PersistenceWeightedGaussianKernel$new(
  bandwidth = 1,
  weight = ~1,
  kernel_approx = NULL,
  n_jobs = 1
)
```

Arguments:

`bandwidth` A numeric value specifying the bandwidth of the Gaussian kernel with which persistence diagrams will be convolved. Defaults to `1.0`.

`weight` A function or a formula coercible into a function via `rlang::as_function()` specifying the weight function for the persistence diagram points. Defaults to the constant function `~ 1`. This function must be defined on 2D points, i.e. lists or arrays of the form $[p_x, p_y]$.

`kernel_approx` A Python class specifying the kernel approximation class used to speed up computation. Defaults to `NULL`. Common kernel approximations classes can be found in the **scikit-learn** library (such as `RBFSampler` for instance).

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1`.

Returns: An object of class `PersistenceWeightedGaussianKernel`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PersistenceWeightedGaussianKernel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
pwgk <- PersistenceWeightedGaussianKernel$new()
pwgk$apply(dgm, dgm)
pwgk$fit_transform(list(dgm))
```

---

persistence_diagram *Persistence Diagram*

Description

A collection of function to manipulate a persistence diagram as an object of class persistence_diagram. A persistence_diagram is a tibble::tibble with a `birth` variable and at least one of `death` or `lifetime` variables.

Usage

```
as_persistence_diagram(x)

is_persistence_diagram(x)
```

Arguments

x	An object coercible into a persistence_diagram object.

Value

An object of class persistence_diagram.

---

plot *Visualization of persistence diagrams*

Description

The `plot.persistence_diagram()` function only displays the plot while the `autoplot.persistence_diagram()` in addition outputs the ggplot2::ggplot object. If one wishes to control graphical properties, it is recommended to use the latter to retrieve the ggplot2::ggplot object and modify and/or add layers as desired.

Usage

```
## S3 method for class 'persistence_diagram'
plot(
  x,
  dimension = NULL,
  alpha = 0.6,
  max_intervals = 20000,
  legend = FALSE,
  greyblock = TRUE,
  type = c("barcode", "diagram"),
  ...
)

## S3 method for class 'persistence_diagram'
autoplot(
  x,
  dimension = NULL,
  alpha = 0.6,
  max_intervals = 20000,
  legend = FALSE,
  greyblock = TRUE,
  n = 10L,
  type = c("barcode", "diagram", "density"),
```

```
  ...
)
```

Arguments

x	An object of class persistence_diagram.
dimension	An integer value specifying the homology dimension to visualize. Defaults to NULL in which case the dimension is retrieved directly in the persistence_diagram object.
alpha	A numeric value between 0 and 1 specifying the transparency of points and lines in the plot. Defaults to 0.6.
max_intervals	An integer value specifying the maximal number of intervals to display. Selected intervals are those with the longest lifetime. Set it to 0 to see them all. Defaults 20000L.
legend	A boolean value specifying whether to display the legend about the homology dimension(s). Defaults to FALSE.
greyblock	A boolean value specifying whether to display a grey lower triangle in the diagram representation for nicer output. Defaults to TRUE.
type	A string specifyfing the type of representation. Choices are "barcode", "diagram" or "density". Defaults to "barcode".
...	Other parameters to be passed on to next methods.
n	An integer value specifying the number of bins for plotting the diagram as a density. Defaults to 10L.

Value

For plot.persistence_diagram(), NULL. Otherwise a ggplot2::ggplot object.

ProminentPoints *Preprocessing: Prominent Points*

Description

This is a class or removing points that are close or far from the diagonal in persistence diagrams. If persistence diagrams are 2-column tibble::tibbles (i.e. persistence diagrams with ordinary features), points are ordered and thresholded by distance-to-diagonal. If persistence diagrams are 1-column tibble::tibbles (i.e. persistence diagrams with essential features), points are not ordered and thresholded by first coordinate.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::PreprocessingStep -> ProminentPoints

Methods

Public methods:

- ProminentPoints$new()
- ProminentPoints$clone()

Method `new()`**:** The `ProminentPoints` constructor.

Usage:

```
ProminentPoints$new(
  use = FALSE,
  num_pts = 10,
  threshold = -1,
  location = c("upper", "lower")
)
```

Arguments:

`use` A boolean value specifying whether to use the class. Defaults to `FALSE`.

`num_pts` An integer value specifying the cardinality threshold. Defaults to `10L`. If `location == "upper"`, keeps the top `num_pts` points that are the farthest away from the diagonal. If `location == "lower"`, keeps the top `num_pts` points that are the closest to the diagonal.

`threshold` A numeric value specifying the distance-to-diagonal threshold. Defaults to `-1.0`. If `location == "upper"`, keeps the points that are at least at a distance threshold from the diagonal. If `location == "lower"`, keeps the points that are at most at a distance threshold from the diagonal.

`location` A string specifying whether to keep the points that are far away (`"upper"`) or close (`"lower"`) to the diagonal. Defaults to `"upper"`.

Returns: An object of class `ProminentPoints`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProminentPoints$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
pp <- ProminentPoints$new()
pp$apply(dgm)
pp$fit_transform(list(dgm))
```

RipsComplex *R6 Class for Rips Complex*

Description

The data structure is a one skeleton graph, or Rips graph, containing edges when the edge length is less or equal to a given threshold. Edge length is computed from a user given point cloud with a given distance function, or a distance matrix.

Super class

`rgudhi::PythonClass` -> `RipsComplex`

Methods

Public methods:

- `RipsComplex$new()`
- `RipsComplex$create_simplex_tree()`
- `RipsComplex$clone()`

Method `new()`**:** `RipsComplex` constructor.

Usage:

```
RipsComplex$new(data, max_edge_length = NULL, sparse = NULL)
```

Arguments:

`data` Either a `n x d` matrix or a length-`n` list of `d`-dimensional vectors or a distance matrix stored as a `dist` object.

`max_edge_length` A numeric value specifying the Rips value.

`sparse` A numeric value specifying the approximation parameter epsilon for buidling a sparse Rips complex. Defaults to `NULL` which builds an exact Rips complex.

Returns: A `RipsComplex` object storing the Rips complex.

Method `create_simplex_tree()`**:**

Usage:

```
RipsComplex$create_simplex_tree(max_dimension)
```

Arguments:

`max_dimension` An integer value specifying the maximal dimension which the Rips complex will be expanded to.

Returns: A `SimplexTree` object storing the computed simplex tree.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RipsComplex$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Clément Maria, Pawel Dlotko, Vincent Rouvreau, Marc Glisse, Yuichi Ike

See Also

Other filtrations and reconstructions: AlphaComplex, TangentialComplex, WitnessComplex

Examples

```
X <- seq_circle(10)
rc1 <- RipsComplex$new(data = X, max_edge_length = 1)
Xm <- Reduce(rbind, X, init = numeric())
rc2 <- RipsComplex$new(data = Xm, max_edge_length = 1)
D <- dist(Xm)
rc3 <- RipsComplex$new(data = D)


X <- seq_circle(10)
rc <- RipsComplex$new(data = X, max_edge_length = 1)
st <- rc$create_simplex_tree(1)
```

RobustScaler *Scales features using statistics that are robust to outliers*

Description

This scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Median and interquartile range are then stored to be used on later data using the `$transform()` method.

Standardization of a dataset is a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean / variance in a negative way. In such cases, the median and the interquartile range often give better results.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseScaler` -> `RobustScaler`

Methods

Public methods:

- RobustScaler$new()
- RobustScaler$clone()

Method `new()`**:** The RobustScaler class constructor.

Usage:

```
RobustScaler$new(
  with_centering = TRUE,
  with_scaling = TRUE,
  quantile_range = c(25, 75),
  copy = TRUE,
  unit_variance = FALSE
)
```

Arguments:

`with_centering` A boolean value specifying whether to center the data before scaling. This will cause transform to raise an exception when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory. Defaults to `TRUE`.

`with_scaling` A boolean value specifying whether to scale the data to interquartile range. Defaults to `TRUE`.

`quantile_range` A length-2 numeric vector specifying the quantile range used to calculate `scale_`. Defaults to `c(25.0, 75.0)`.

`copy` A boolean value specifying whether to perform in-place scaling and avoid a copy (if the input is already a numpy array). Defaults to `TRUE`.

`unit_variance` A boolean value specifying whether to scale data so that normally distributed features have a variance of 1. In general, if the difference between the x-values of $q_{\max}$ and $q_{\min}$ for a standard normal distribution is greater than 1, the data set will be scaled down. If less than 1, the data set will be scaled up. Defaults to `FALSE`.

Returns: An object of class RobustScaler.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RobustScaler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
rs <- RobustScaler$new()
```

seq_circle *Circular Sequence Generation*

Description

Generates a sequence of 2D points evenly spaced on the unit circle.

Usage

```
seq_circle(n)
```

Arguments

n	An integer value specifying the number of points in the sequence.

Value

A base::list of length-2 numeric vectors storing 2D points evenly spaced on the unit circle.

Examples

```
seq_circle(10)
```

Silhouette *Vector Representation: Silhouette*

Description

Computes persistence silhouettes from a list of persistence diagrams. A persistence silhouette is computed by taking a weighted average of the collection of 1D piecewise-linear functions given by the persistence landscapes, and then by evenly sampling this average on a given range. Finally, the corresponding vector of samples is returned. See https://arxiv.org/abs/1312.0308 for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `Silhouette`

Methods

Public methods:

- `Silhouette$new()`
- `Silhouette$clone()`

Method `new()`**:** The `Silhouette` constructor.

Usage:

```
Silhouette$new(weight = ~1, resolution = 100, sample_range = rep(NA_real_, 2))
```

Arguments:

`weight` A function or a formula coercible into a function via `rlang::as_function()` specifying the weight function for the persistence diagram points. Defaults to the constant function `~ 1`. This function must be defined on 2D points, i.e. lists or arrays of the form $[p_x, p_y]$.

`resolution` An length-1 integer vector specifying the size (in pixels) of the persistence image. Defaults to `rep(20L, 2)`.

`sample_range` A length-2 numeric vector specifying the domain for the entropy summary function, of the form $[x_{\min}, x_{\max}]$. Defaults to `rep(NA, 2)`. It is the interval on which samples will be drawn evenly. If one of the values is `NA`, it can be computed from the persistence diagrams with the `$fit()` method.

Returns: An object of class `Silhouette`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Silhouette$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
sil <- Silhouette$new()
sil$apply(dgm) # TO DO: fix gd because it does not set sample_range automatically
sil$fit_transform(list(dgm))
```

`SimplexTree` *R6 Class for Simplex Tree*

Description

The simplex tree is an efficient and flexible data structure for representing general (filtered) simplicial complexes. The data structure is described in Boissonnat and Maria (2014).

Details

This class is a filtered, with keys, and non contiguous vertices version of the simplex tree.

References:

Boissonnat J, Maria C (2014). "The simplex tree: An efficient data structure for general simplicial complexes." *Algorithmica*, **70**(3), 406–427.

Super class

`rgudhi::PythonClass` -> `SimplexTree`

Methods

Public methods:

- `SimplexTree$new()`
- `SimplexTree$set_is_flag()`
- `SimplexTree$assign_filtration()`
- `SimplexTree$betti_numbers()`
- `SimplexTree$collapse_edges()`
- `SimplexTree$compute_persistence()`
- `SimplexTree$dimension()`
- `SimplexTree$expansion()`
- `SimplexTree$extend_filtration()`
- `SimplexTree$extended_persistence()`
- `SimplexTree$filtration()`
- `SimplexTree$find()`
- `SimplexTree$flag_persistence_generators()`
- `SimplexTree$get_boundaries()`
- `SimplexTree$get_cofaces()`
- `SimplexTree$get_filtration()`
- `SimplexTree$get_simplices()`
- `SimplexTree$get_skeleton()`
- `SimplexTree$get_star()`
- `SimplexTree$insert()`
- `SimplexTree$lower_star_persistence_generators()`
- `SimplexTree$make_filtration_non_decreasing()`
- `SimplexTree$num_simplices()`
- `SimplexTree$num_vertices()`
- `SimplexTree$persistence()`
- `SimplexTree$persistence_intervals_in_dimension()`
- `SimplexTree$persistence_pairs()`
- `SimplexTree$persistent_betti_numbers()`
- `SimplexTree$prune_above_filtration()`
- `SimplexTree$remove_maximal_simplex()`

- SimplexTree$reset_filtration()
- SimplexTree$set_dimension()
- SimplexTree$upper_bound_dimension()
- SimplexTree$write_persistence_diagram()
- SimplexTree$clone()

Method new(): The SimplexTree class constructor.

Usage:

```
SimplexTree$new(py_class = NULL)
```

Arguments:

py_class A Python SimplexTree class object. Defaults to NULL which uses the Python class constructor instead.

Returns: A new SimplexTree object.

Method set_is_flag(): This function sets the internal field m_IsFlag which records whether the simplex tree is a flag complex (i.e. has been generated by a Rips complex).

Usage:

```
SimplexTree$set_is_flag(val)
```

Arguments:

val A boolean specifying whether the simplex tree is a flag complex.

Details: The SimplexTree class initializes the m_IsFlag field to FALSE by default and this method specifically allows to overwrite this default value.

Returns: The updated SimplexTree class itself invisibly.

Method assign_filtration(): This function assigns a new filtration value to a given N-simplex.

Usage:

```
SimplexTree$assign_filtration(simplex, filtration)
```

Arguments:

simplex An integer vector representing the N-simplex in the form of a list of vertices.

filtration A numeric value specifying the new filtration value.

Details: Beware that after this operation, the structure may not be a valid filtration anymore, a simplex could have a lower filtration value than one of its faces. Callers are responsible for fixing this (with more calls to the $assign_filtration() method or a call to the $make_filtration_non_decreasing() method for instance) before calling any function that relies on the filtration property, such as persistence().

Returns: The updated SimplexTree class itself invisibly.

Method betti_numbers(): This function returns the Betti numbers of the simplicial complex.

Usage:

```
SimplexTree$betti_numbers()
```

Returns: An integer vector storing the Betti numbers.

Method `collapse_edges()`**:** Assuming the simplex tree is a 1-skeleton graph, this method collapse edges (simplices of higher dimension are ignored) and resets the simplex tree from the remaining edges. A good candidate is to build a simplex tree on top of a `RipsComplex` of dimension 1 before collapsing edges as done in this Python example. For implementation details, please refer to Boissonnat and Pritam (2020).

Usage:

```
SimplexTree$collapse_edges(nb_iterations = 1)
```

Arguments:

`nb_iterations` An integer value specifying the number of edge collapse iterations to perform. Defaults to `1L`.

Details: It requires `Eigen >= 3.1.0` and an exception is thrown if not available.

References:

Boissonnat J, Pritam S (2020). "Edge collapse and persistence of flag complexes." In *SoCG 2020-36th International Symposium on Computational Geometry*.

Returns: The updated `SimplexTree` class itself invisibly.

Method `compute_persistence()`**:** This function computes the persistence of the simplicial complex, so it can be accessed through `$persistent_betti_numbers()`, `$persistence_pairs()`, etc. This function is equivalent to `$persistence()` when you do not want the list that `$persistence()` returns.

Usage:

```
SimplexTree$compute_persistence(
  homology_coeff_field = 11,
  min_persistence = 0,
  persistence_dim_max = FALSE
)
```

Arguments:

`homology_coeff_field` An integer value specifying the homology coefficient field. Must be a prime number. Defaults to `11L`. Maximum is `46337L`.

`min_persistence` A numeric value specifying the minimum persistence value to take into account (strictly greater than `min_persistence`). Defaults to `0.0`. Set `min_persistence = -1.0` to see all values.

`persistence_dim_max` A boolean specifying whether the persistent homology for the maximal dimension in the complex is computed (`persistence_dim_max = TRUE`). If `FALSE`, it is ignored. Defaults to `FALSE`.

Returns: The updated `SimplexTree` class itself invisibly.

Method `dimension()`**:** This function returns the dimension of the simplicial complex.

Usage:

```
SimplexTree$dimension()
```

Details: This function is not constant time because it can recompute dimension if required (can be triggered by `$remove_maximal_simplex()` or `$prune_above_filtration()` methods for instance).

Returns: An integer value storing the simplicial complex dimension.

Method `expansion()`**:** Expands the simplex tree containing only its one skeleton until dimension `max_dim`.

Usage:

```
SimplexTree$expansion(max_dim)
```

Arguments:

`max_dim` An integer value specifying the maximal dimension to expented the simplex tree to.

Details: The expanded simplicial complex until dimension `d` attached to a graph `G` is the maximal simplicial complex of dimension at most `d` admitting the graph `G` as 1-skeleton. The filtration value assigned to a simplex is the maximal filtration value of one of its edges.
The simplex tree must contain no simplex of dimension bigger than 1 when calling the method.

Returns: The updated `SimplexTree` class itself invisibly.

Method `extend_filtration()`**:** Extend filtration for computing extended persistence. This function only uses the filtration values at the 0-dimensional simplices, and computes the extended persistence diagram induced by the lower-star filtration computed with these values.

Usage:

```
SimplexTree$extend_filtration()
```

Details: Note that after calling this function, the filtration values are actually modified within the simplex tree. The method `$extended_persistence()` retrieves the original values.
Note that this code creates an extra vertex internally, so you should make sure that the simplex tree does not contain a vertex with the largest possible value (i.e., `4294967295`).
This notebook explains how to compute an extension of persistence called extended persistence.

Returns: The updated `SimplexTree` class itself invisibly.

Method `extended_persistence()`**:** This function retrieves good values for extended persistence, and separate the diagrams into the Ordinary, Relative, Extended+ and Extended- subdiagrams.

Usage:

```
SimplexTree$extended_persistence(
  homology_coeff_field = 11,
  min_persistence = 0
)
```

Arguments:

`homology_coeff_field` An integer value specifying the homology coefficient field. Must be a prime number. Defaults to `11L`. Maximum is `46337L`.

`min_persistence` A numeric value specifying the minimum persistence value to take into account (strictly greater than `min_persistence`). Defaults to `0.0`. Set `min_persistence = -1.0` to see all values.

Details: The coordinates of the persistence diagram points might be a little different than the original filtration values due to the internal transformation (scaling to `[-2,-1]`) that is performed on these values during the computation of extended persistence.
This notebook explains how to compute an extension of persistence called extended persistence.

Returns: A list of four persistence diagrams in the format described in `$persistence()`. The first one is `Ordinary`, the second one is `Relative`, the third one is `Extended+` and the fourth one is `Extended-`. See this article and/or Section 2.2 in this article for a description of these subtypes.

Method `filtration()`**:** This function returns the filtration value for a given N-simplex in this simplicial complex, or +infinity if it is not in the complex.

Usage:

```
SimplexTree$filtration(simplex)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

Returns: A numeric value storing the filtration value for the input N-simplex.

Method `find()`**:** This function returns if the N-simplex was found in the simplicial complex or not.

Usage:

```
SimplexTree$find(simplex)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

Returns: A boolean storing whether the input N-simplex was found in the simplicial complex.

Method `flag_persistence_generators()`**:** Assuming this is a flag complex, this function returns the persistence pairs, where each simplex is replaced with the vertices of the edges that gave it its filtration value.

Usage:

```
SimplexTree$flag_persistence_generators()
```

Returns: A list with the following components:

- An `n x 3` integer matrix containing the regular persistence pairs of dimension 0, with one vertex for birth and two for death;
- A list of `m x 4` integer matrices containing the other regular persistence pairs, grouped by dimension, with 2 vertices per extremity;
- An `l x ?` integer matrix containing the connected components, with one vertex each;
- A list of `k x 2` integer matrices containing the other essential features, grouped by dimension, with 2 vertices for birth.

Method `get_boundaries()`**:** For a given N-simplex, this function returns a list of simplices of dimension N-1 corresponding to the boundaries of the N-simplex.

Usage:

```
SimplexTree$get_boundaries(simplex)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

Returns: A `tibble` listing the (simplicies of the) boundary of the input N-simplex in column `simplex` along with their corresponding filtration value in column `filtration`.

Method `get_cofaces()`**:** This function returns the cofaces of a given N-simplex with a given codimension.

Usage:

```
SimplexTree$get_cofaces(simplex, codimension)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

`codimension` An integer value specifying the codimension. If `codimension = 0`, all cofaces are returned (equivalent of `$get_star()` function).

Returns: A `tibble` listing the (simplicies of the) cofaces of the input N-simplex in column `simplex` along with their corresponding filtration value in column `filtration`.

Method `get_filtration()`**:** This function retrieves the list of simplices and their given filtration values sorted by increasing filtration values.

Usage:

```
SimplexTree$get_filtration()
```

Returns: A `tibble` listing the simplicies in column `simplex` along with their corresponding filtration value in column `filtration`, in increasing order of filtration value.

Method `get_simplices()`**:** This function retrieves the list of simplices and their given filtration values.

Usage:

```
SimplexTree$get_simplices()
```

Returns: A `tibble` listing the simplicies in column `simplex` along with their corresponding filtration value in column `filtration`, in increasing order of filtration value.

Method `get_skeleton()`**:** This function returns a generator with the (simplices of the) skeleton of a maximum given dimension.

Usage:

```
SimplexTree$get_skeleton(dimension)
```

Arguments:

`dimension` A integer value specifying the skeleton dimension value.

Returns: A `tibble` listing the (simplicies of the) skeleton of a maximum dimension in column `simplex` along with their corresponding filtration value in column `filtration`.

Method `get_star()`**:** This function returns the star of a given N-simplex.

Usage:

```
SimplexTree$get_star(simplex)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

Returns: A `tibble` listing the (simplicies of the) star of a simplex in column `simplex` along with their corresponding filtration value in column `filtration`.

Method `insert()`**:** This function inserts the given N-simplex and its subfaces with the given filtration value. If some of those simplices are already present with a higher filtration value, their filtration value is lowered.

Usage:

```
SimplexTree$insert(simplex, filtration = 0, chainable = TRUE)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

`filtration` A numeric value specifying the filtration value of the simplex. Defaults to `0.0`.

`chainable` A boolean specifying whether the method should return the class itself, hence allowing its use in pipe chaining. Defaults to `TRUE`, which enables chaining.

Returns: The updated `SimplexTree` class itself invisibly if `chainable` is set to `TRUE` (default behavior), or a boolean set to `TRUE` if the simplex was not yet in the complex or `FALSE` otherwise (whatever its original filtration value).

Method `lower_star_persistence_generators()`**:** Assuming this is a lower-star filtration, this function returns the persistence pairs, where each simplex is replaced with the vertex that gave it its filtration value.

Usage:

```
SimplexTree$lower_star_persistence_generators()
```

Returns: A list with the following components:

- A list of `n x 2` integer matrices containing the regular persistence pairs, grouped by dimension, with one vertex per extremity;
- A list of `m x ?` integer matrices containing the essential features, grouped by dimension, with one vertex each.

Method `make_filtration_non_decreasing()`**:** This function ensures that each simplex has a higher filtration value than its faces by increasing the filtration values.

Usage:

```
SimplexTree$make_filtration_non_decreasing(chainable = TRUE)
```

Arguments:

`chainable` A boolean specifying whether the method should return the class itself, hence allowing its use in pipe chaining. Defaults to `TRUE`, which enables chaining.

Returns: The updated `SimplexTree` class itself invisibly if `chainable` is set to `TRUE` (default behavior), or a boolean set to `TRUE` if any filtration value was modified or to `FALSE` if the filtration was already non-decreasing.

Method `num_simplices()`**:** This function returns the number of simplices of the simplicial complex.

Usage:

```
SimplexTree$num_simplices()
```

Returns: An integer value storing the number of simplices in the simplicial complex.

Method `num_vertices()`**:** This function returns the number of vertices of the simplicial complex.

Usage:

```
SimplexTree$num_vertices()
```

Returns: An integer value storing the number of vertices in the simplicial complex.

Method `persistence()`**:** This function computes and returns the persistence of the simplicial complex.

Usage:

```
SimplexTree$persistence(
  homology_coeff_field = 11,
  min_persistence = 0,
  persistence_dim_max = FALSE
)
```

Arguments:

`homology_coeff_field` An integer value specifying the homology coefficient field. Must be a prime number. Defaults to `11L`. Maximum is `46337L`.

`min_persistence` A numeric value specifying the minimum persistence value to take into account (strictly greater than `min_persistence`). Defaults to `0.0`. Set `min_persistence = -1.0` to see all values.

`persistence_dim_max` A boolean specifying whether the persistent homology for the maximal dimension in the complex is computed (`persistence_dim_max = TRUE`). If `FALSE`, it is ignored. Defaults to `FALSE`.

Returns: A `tibble` listing all persistence feature summarised by 3 variables: `dimension`, `birth` and `death`.

Method `persistence_intervals_in_dimension()`**:** This function returns the persistence intervals of the simplicial complex in a specific dimension.

Usage:

```
SimplexTree$persistence_intervals_in_dimension(dimension)
```

Arguments:

`dimension` An integer value specifying the desired dimension.

Returns: A `tibble` storing the persistence intervals for the required dimension in two columns `birth` and `death`.

Method `persistence_pairs()`**:** This function returns a list of persistence birth and death simplices pairs.

Usage:

```
SimplexTree$persistence_pairs()
```

Returns: A list of pairs of integer vectors storing a list of persistence simplices intervals.

Method `persistent_betti_numbers()`**:** This function returns the persistent Betti numbers of the simplicial complex.

Usage:

```
SimplexTree$persistent_betti_numbers(from_value, to_value)
```

Arguments:

`from_value` A numeric value specifying the persistence birth limit to be added in the numbers (`persistent birth <= from_value`).

`to_value` A numeric value specifying the persistence death limit to be added in the numbers (`persistent death > to_value`).

Returns: An integer vector storing the persistent Betti numbers.

Method `prune_above_filtration()`**:** Prune above filtration value given as parameter.

Usage:

```
SimplexTree$prune_above_filtration(filtration, chainable = TRUE)
```

Arguments:

`filtration` A numeric value specifying the maximum threshold value.

`chainable` A boolean specifying whether the method should return the class itself, hence allowing its use in pipe chaining. Defaults to `TRUE`, which enables chaining.

Details: Note that the dimension of the simplicial complex may be lower after calling `prune_above_filtration()` than it was before. However, `upper_bound_dimension()` will return the old value, which remains a valid upper bound. If you care, you can call `dimension()` method to recompute the exact dimension.

Returns: The updated `SimplexTree` class itself invisibly if `chainable` is set to `TRUE` (default behavior), or a boolean set to `TRUE` if the filtration has been modified or to `FALSE` otherwise.

Method `remove_maximal_simplex()`**:** This function removes a given maximal N-simplex from the simplicial complex.

Usage:

```
SimplexTree$remove_maximal_simplex(simplex)
```

Arguments:

`simplex` An integer vector representing the N-simplex in the form of a list of vertices.

Details: The dimension of the simplicial complex may be lower after calling `$remove_maximal_simplex()` than it was before. However, `$upper_bound_dimension()` method will return the old value, which remains a valid upper bound. If you care, you can call `$dimension()` to recompute the exact dimension.

Returns: The updated `SimplexTree` class itself invisibly.

Method `reset_filtration()`**:** This function resets the filtration value of all the simplices of dimension at least `min_dim`. Resets all the simplex tree when `min_dim = 0L`. `reset_filtration` may break the filtration property with `min_dim > 0`, and it is the user's responsibility to make it a valid filtration (using a large enough `filtration` value, or calling `$make_filtration_non_decreasing()` afterwards for instance).

Usage:

```
SimplexTree$reset_filtration(filtration, min_dim = 0)
```

Arguments:

`filtration` A numeric value specyfing the filtration threshold.

`min_dim` An integer value specifying the minimal dimension. Defaults to `0L`.

Returns: The updated `SimplexTree` class itself invisibly.

Method `set_dimension()`**:** This function sets the dimension of the simplicial complex.

Usage:

```
SimplexTree$set_dimension(dimension)
```

Arguments:

`dimension` An integer value specifying the dimension.

Details: This function must be used with caution because it disables dimension recomputation when required (this recomputation can be triggered by `$remove_maximal_simplex()` or `$prune_above_filtration()`).

Returns: The updated `SimplexTree` class itself invisibly.

Method `upper_bound_dimension()`**:** This function returns a valid dimension upper bound of the simplicial complex.

Usage:

```
SimplexTree$upper_bound_dimension()
```

Returns: An integer value storing an upper bound on the dimension of the simplicial complex.

Method `write_persistence_diagram()`**:** This function writes the persistence intervals of the simplicial complex in a user given file name.

Usage:

```
SimplexTree$write_persistence_diagram(persistence_file)
```

Arguments:

`persistence_file` A string specifying the name of the file.

Returns: The updated `SimplexTree` class itself invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SimplexTree$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Clément Maria

See Also

Other data structures for cell complexes: `CubicalComplex`, `PeriodicCubicalComplex`

Examples

```
st <- SimplexTree$new()
st


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$set_is_flag(TRUE)


X <- seq_circle(10)
```

```
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$filtration(1)
st$assign_filtration(1, 0.8)
st$filtration(1)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$compute_persistence()$betti_numbers()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$collapse_edges()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$dimension()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$expansion(2)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$extend_filtration()
st$extended_persistence()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$filtration(0)
st$filtration(1:2)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$find(0)


X <- seq_circle(10)
rc <- RipsComplex$new(data = X, max_edge_length = 1)
```

```
st <- rc$create_simplex_tree(1)
st$compute_persistence()$flag_persistence_generators()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
splx <- st$get_simplices()$simplex[[1]]
st$get_boundaries(splx)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$get_cofaces(1:2, 0)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$get_filtration()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$get_simplices()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$get_skeleton(0)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$get_star(1:2)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$insert(1:2)
st$insert(1:3, chainable = FALSE)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$compute_persistence()$lower_star_persistence_generators()
```

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$make_filtration_non_decreasing()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$num_simplices()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$num_vertices()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$persistence()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$compute_persistence()$persistence_intervals_in_dimension(1)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$compute_persistence()$persistence_pairs()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$compute_persistence()$persistent_betti_numbers(0, 0.1)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$prune_above_filtration(0.12)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$remove_maximal_simplex(1:2)
```

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$reset_filtration(0.1)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$set_dimension(1)


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
st$upper_bound_dimension()


X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
f <- fs::file_temp(ext = ".dgm")
st$compute_persistence()$write_persistence_diagram(f)
fs::file_delete(f)
```

SlicedWassersteinDistance

Metrics: Sliced Wasserstein Distance

Description

Computes the sliced Wasserstein distance matrix from a list of persistence diagrams. The Sliced Wasserstein distance is computed by projecting the persistence diagrams onto lines, comparing the projections with the 1-norm, and finally integrating over all possible lines. See http://proceedings.mlr.press/v70/carriere17a.ht for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::MetricStep` -> `SlicedWassersteinDistance`

Methods

Public methods:

- `SlicedWassersteinDistance$new()`
- `SlicedWassersteinDistance$clone()`

Method `new()`**:** The `SlicedWassersteinDistance` constructor.

Usage:

```
SlicedWassersteinDistance$new(num_directions = 10, n_jobs = 1)
```

Arguments:

`num_directions` An integer value specifying the number of lines evenly sampled from $[-\pi/2, \pi/2]$ in order to approximate and speed up the kernel computation. Defaults to `10L`.

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1L`.

Returns: An object of class `SlicedWassersteinDistance`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SlicedWassersteinDistance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
dis <- SlicedWassersteinDistance$new()
dis$apply(dgm, dgm)
dis$fit_transform(list(dgm))
```

SpectralBiclustering *Performs clustering according to the spectral biclustering algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.SpectralBiclustering.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `SpectralBiclustering`

Methods

Public methods:

- SpectralBiclustering$new()
- SpectralBiclustering$clone()

Method `new()`**:** The SpectralBiclustering class constructor.

Usage:

```
SpectralBiclustering$new(
  n_clusters = 3L,
  method = c("bistochastic", "scale", "log"),
  n_components = 6L,
  n_best = 3L,
  svd_method = c("randomized", "arpack"),
  n_svd_vecs = NULL,
  mini_batch = FALSE,
  init = c("k-means++", "random"),
  n_init = 10L,
  random_state = NULL
)
```

Arguments:

`n_clusters` An integer value or a length-2 vector specifying the number of row and column clusters in the checkerboard structure. Defaults to `3L`.

`method` A string specifying the method of normalizing and converting singular vectors into biclusters. May be one of `"scale"`, `"bistochastic"` or `"log"`. The authors recommend using `"log"`. If the data is sparse, however, log-normalization will not work, which is why the default is `"bistochastic"`. Warning: if `method == "log"`, the data must not be sparse.

`n_components` An integer value specifying the number of singular vectors to check. Defaults to `6L`.

`n_best` An integer value specifying the number of best singular vectors to which to project the data for clustering. Defaults to `3L`.

`svd_method` A string specifying the algorithm for finding singular vectors. May be `"randomized"` or `"arpack"`. If `"randomized"`, uses `randomized_svd()`, which may be faster for large matrices. If `"arpack"`, uses `scipy.sparse.linalg.svds()`, which is more accurate, but possibly slower in some cases. Defaults to `"randomized"`.

`n_svd_vecs` An integer value specifying the number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method == "arpack"` and `n_oversamples` when `svd_method == "randomized"`. Defaults to `NULL`.

`mini_batch` A boolean value specifying whether to use mini-batch k-means, which is faster but may get different results. Defaults to `FALSE`.

`init` A string specifying the method for initialization of k-means algorithm. Choices are `"k-means++"` or `"random"`. Defaults to `"k-means++"`.

`n_init` An integer value specifying the number of random initializations that are tried with the k-means algorithm. If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen. Defaults to `10L`.

`random_state` An integer value specifying a pseudo random number generator used for the initialization of the `lobpcg` eigenvectors decomposition when `eigen_solver == "amg"`, and for the k-means initialization. Defaults to `NULL` which uses clock time.

Returns: An object of class SpectralBiclustering.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpectralBiclustering$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- SpectralBiclustering$new()
```

`SpectralClustering` *Performs clustering according to the spectral clustering algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.SpectralClustering.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseClustering` -> `SpectralClustering`

Methods

Public methods:

- `SpectralClustering$new()`
- `SpectralClustering$clone()`

Method `new()`**:** The SpectralClustering class constructor.

Usage:

```
SpectralClustering$new(
  n_clusters = 8L,
  eigen_solver = c("arpack", "lobpcg", "amg"),
  n_components = NULL,
  random_state = NULL,
  n_init = 10L,
  gamma = 1,
  affinity = c("rbf", "nearest_neighbors", "precomputed",
    "precomputed_nearest_neighbors"),
  n_neighbors = 10L,
```

```
  eigen_tol = "auto",
  assign_labels = c("kmeans", "discretize", "cluster_qr"),
  degree = 3L,
  coef0 = 1,
  kernel_params = NULL,
  n_jobs = 1L,
  verbose = FALSE
)
```

Arguments:

`n_clusters` An integer value specifying the dimension of the projection subspace. Defaults to `8L`.

`eigen_solver` A string specifying the eigenvalue decomposition strategy to use. Choices are `c("arpack", "lobpcg", "amg")`. AMG requires **pyamg** to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities. Defaults to `"arpack"`.

`n_components` An integer value specifying the number of eigenvectors to use for the spectral embedding. Defaults to `NULL`, in which case, `n_clusters` is used.

`random_state` An integer value specifying a pseudo random number generator used for the initialization of the `lobpcg` eigenvectors decomposition when `eigen_solver == "amg"`, and for the k-means initialization. Defaults to `NULL` which uses clock time.

`n_init` An integer value specifying the number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of `n_init` consecutive runs in terms of inertia. Only used if `assign_labels == "kmeans"`. Defaults to `10L`.

`gamma` A numeric value specifying the kernel coefficient for `rbf`, `poly`, `sigmoid`, `laplacian` and `chi2` kernels. Ignored for `affinity == "nearest_neighbors"`. Defaults to `1.0`.

`affinity` Either a string or an object coercible to a function via `rlang::as_function()` specifying how to construct the affinity matrix:

- `"nearest_neighbors"`: construct the affinity matrix by computing a graph of nearest neighbors;
- `"rbf"`: construct the affinity matrix using a radial basis function (RBF) kernel;
- `"precomputed"`: interpret `X` as a precomputed affinity matrix, where larger values indicate greater similarity between instances;
- `"precomputed_nearest_neighbors"`: interpret `X` as a sparse graph of precomputed distances, and construct a binary affinity matrix from the `n_neighbors` nearest neighbors of each instance;
- one of the kernels supported by pairwise_kernels.

Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.
Defaults to `"rbf"`.

`n_neighbors` An integer value specifying the number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity == "rbf"`. Defaults to `10L`.

`eigen_tol` A numeric value specifying the stopping criterion for the eigen-decomposition of the Laplacian matrix. If `eigen_tol == "auto"`, then the passed tolerance will depend on the `eigen_solver`:

- If `eigen_solver == "arpack"`, then `eigen_tol = 0.0`;

- If `eigen_solver == "lobpcg"` or `eigen_solver == "amg"`, then `eigen_tol == NULL` which configures the underlying `lobpcg` solver to automatically resolve the value according to their heuristics.

Note that when using `eigen_solver == "lobpcg"` or `eigen_solver == "amg"` values of `tol < 1e-5` may lead to convergence issues and should be avoided.
Defaults to `"auto"`.

`assign_labels` A string specifying the strategy for assigning labels in the embedding space. There are two ways to assign labels after the Laplacian embedding. k-means is a popular choice (`"kmeans"`), but it can be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization (`"discretize"`). The `cluster_qr` method directly extract clusters from eigenvectors in spectral clustering. In contrast to k-means and discretization, `cluster_qr` has no tuning parameters and runs no iterations, yet may outperform k-means and discretization in terms of both quality and speed. Defaults to `"kmeans"`.

`degree` An integer value specifying the degree of the polynomial kernel. Ignored by other kernels. Defaults to `3L`.

`coef0` A numeric value specifying the value of the zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels. Defaults to `1.0`.

`kernel_params` A named list specifying extra arguments to the kernels passed as functions. Ignored by other kernels. Defaults to `NULL`.

`n_jobs` An integer value specifying the number of parallel jobs to run for neighbors search. Defaults to `1L`. A value of `-1L` means using all processors.

`verbose` A boolean value specifying the verbosity mode. Defaults to `FALSE`.

Returns: An object of class SpectralClustering.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpectralClustering$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- SpectralClustering$new()
```

`SpectralCoclustering` *Performs clustering according to the spectral coclustering algorithm*

Description

This is a wrapper around the Python class sklearn.cluster.SpectralCoclustering.

Super classes

rgudhi::PythonClass -> rgudhi::SKLearnClass -> rgudhi::BaseClustering -> SpectralCoclustering

Methods

Public methods:

- SpectralCoclustering$new()
- SpectralCoclustering$clone()

Method `new()`**:** The SpectralCoclustering class constructor.

Usage:

```
SpectralCoclustering$new(
  n_clusters = 3L,
  svd_method = c("randomized", "arpack"),
  n_svd_vecs = NULL,
  mini_batch = FALSE,
  init = c("k-means++", "random"),
  n_init = 10L,
  random_state = NULL
)
```

Arguments:

`n_clusters` An integer value specifying the number of biclusters to find. Defaults to `3L`.

`svd_method` A string specifying the algorithm for finding singular vectors. May be `"randomized"` or `"arpack"`. If `"randomized"`, uses `sklearn.utils.extmath.randomized_svd()`, which may be faster for large matrices. If `"arpack"`, uses `scipy.sparse.linalg.svds()`, which is more accurate, but possibly slower in some cases. Defaults to `"randomized"`.

`n_svd_vecs` An integer value specifying the number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method == "arpack"` and `n_oversamples` when `svd_method == "randomized"`. Defaults to `NULL`.

`mini_batch` A boolean value specifying whether to use mini-batch k-means, which is faster but may get different results. Defaults to `FALSE`.

`init` A string specifying the method for initialization of k-means algorithm. Choices are `"k-means++"` or `"random"`. Defaults to `"k-means++"`.

`n_init` An integer value specifying the number of random initializations that are tried with the k-means algorithm. If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen. Defaults to `10L`.

`random_state` An integer value specifying a pseudo random number generator used for the initialization of the `lobpcg` eigenvectors decomposition when `eigen_solver == "amg"`, and for the k-means initialization. Defaults to `NULL` which uses clock time.

Returns: An object of class SpectralCoclustering.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SpectralCoclustering$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
cl <- SpectralCoclustering$new()
```

`sphere` *Sampling on the Sphere*

Description

The function `sphere()` enables uniform sampling of random *i.i.d.* points on a $(d-1)$-sphere in R^d. The user should provide the number of points `n_samples` to be generated on the sphere and the ambient dimension `ambient_dim`. The radius of the sphere is optional and is equal to `1` by default. Only random points generation is currently available.

Usage

```
sphere(n_samples, ambient_dim, radius = 1)
```

Arguments

`n_samples`	An integer value specifying the sample size.
`ambient_dim`	An integer value specifying the dimension of the ambient space.
`radius`	A numeric value specifying the radius of the sphere. Defaults to `1.0`.

Value

A numeric array of shape $n_{\text{samples}} \times \text{ambient\_dim}$ storing `n_samples` points uniformly sampled on the sphere of dimension `ambient_dim - 1`.

Examples

```
sphere(10, 2)
```

`StandardScaler` *Standardizes features by removing the mean and scaling to unit variance*

Description

The standard score of a sample x is calculated as:

$$z = \frac{(x - u)}{s}$$

where u is the mean of the training samples or 0 if `with_mean = FALSE`, and s is the standard deviation of the training samples or 1 if `with_std = FALSE`.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

This scaler can also be applied to sparse CSR or CSC matrices by passing `with_mean = FALSE` to avoid breaking the sparsity structure of the data.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::BaseScaler` -> `StandardScaler`

Methods

Public methods:

- `StandardScaler$new()`
- `StandardScaler$clone()`

Method `new()`**:** The StandardScaler class constructor.

Usage:

```
StandardScaler$new(copy = TRUE, with_mean = TRUE, with_std = TRUE)
```

Arguments:

`copy` A boolean value specifying whether to perform in-place scaling and avoid a copy (if the input is already a numpy array). Defaults to `TRUE`.

`with_mean` A boolean value specifying whether to center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory. Defaults to `TRUE`.

`with_std` A boolean value specifying whether to scale the data to unit variance (or equivalently, unit standard deviation). Defaults to `TRUE`.

Returns: An object of class StandardScaler.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
StandardScaler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
ss <- StandardScaler$new()
```

`StrongWitnessComplex` *R6 Class for Strong Witness Complex*

Description

A Witness complex $\mathrm{Wit}(W, L)$ is a simplicial complex defined on two sets of points in R^D. The data structure is described in Boissonnat and Maria (2014).

Details

The class constructs a (strong) witness complex for a given table of nearest landmarks with respect to witnesses.

Super classes

`rgudhi::PythonClass` -> `rgudhi::WitnessComplex` -> `StrongWitnessComplex`

Methods

Public methods:

- `StrongWitnessComplex$new()`
- `StrongWitnessComplex$clone()`

Method `new()`**:** `StrongWitnessComplex` constructor.

Usage:

```
StrongWitnessComplex$new(nearest_landmark_table)
```

Arguments:

`nearest_landmark_table` A list of `tibble`s specifying for each *witness* `w`, the ordered list of nearest landmarks with id in column `nearest_landmark` and distance to `w` in column `distance`.

Returns: A `StrongWitnessComplex` object storing the strong Witness complex.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
StrongWitnessComplex$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
withr::with_seed(1234, {
  l <- list(
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    ),
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    )
  )
})
wc <- StrongWitnessComplex$new(nearest_landmark_table = l)
wc
```

`TangentialComplex` *R6 Class for Tangential Complex*

Description

A Tangential Delaunay complex is a simplicial complex designed to reconstruct a k-dimensional manifold embedded in d-dimensional Euclidean space. The input is a point sample coming from an unknown manifold. The running time depends only linearly on the extrinsic dimension d and exponentially on the intrinsic dimension k.

Details

The TangentialComplex class represents a tangential complex. After the computation of the complex, an optional post-processing called perturbation can be run to attempt to remove inconsistencies.

Super class

`rgudhi::PythonClass` -> `TangentialComplex`

Methods

Public methods:

- TangentialComplex$new()
- TangentialComplex$compute_tangential_complex()
- TangentialComplex$create_simplex_tree()
- TangentialComplex$get_point()
- TangentialComplex$num_inconsistent_simplices()
- TangentialComplex$num_inconsistent_stars()
- TangentialComplex$num_simplices()
- TangentialComplex$num_vertices()
- TangentialComplex$set_max_squared_edge_length()
- TangentialComplex$clone()

Method `new()`**:** `TangentialComplex` constructor.

Usage:

```
TangentialComplex$new(points, intrinsic_dim = NULL)
```

Arguments:

`points` Either a character string specifying the path to an OFF file which the points can be read from or a numeric matrix or list of numeric vectors specifying the points directly.

`intrinsic_dim` An integer value specifying the intrinsic dimension of the manifold. This is nedded when points are provided as a numeric matrix or a list of numeric vectors. Defaults to `NULL`.

Returns: A `TangentialComplex` object storing the tangential complex.

Method `compute_tangential_complex()`**:** This function computes the tangential complex.

Usage:

```
TangentialComplex$compute_tangential_complex()
```

Details: In debug mode, it may raise a `ValueError` if the computed star dimension is too low. Try to set a bigger maximal edge length value via the `$set_max_squared_edge_length()` method if this happens.

Returns: The updated `TangentialComplex` class itself invisibly.

Method `create_simplex_tree()`**:** Exports the complex into a simplex tree.

Usage:

```
TangentialComplex$create_simplex_tree()
```

Returns: A `SimplexTree` object storing the computed simplex tree.

Method `get_point()`**:** This function returns the point corresponding to a given vertex from the `SimplexTree`.

Usage:

```
TangentialComplex$get_point(vertex)
```

Arguments:

`vertex` An integer value specifying the desired vertex.

Returns: A numeric vector storing the point corresponding to the input vertex.

Method `num_inconsistent_simplices()`**:**

Usage:

```
TangentialComplex$num_inconsistent_simplices()
```

Returns: An integer value storing the number of inconsistent simplicies.

Method `num_inconsistent_stars()`**:**

Usage:

```
TangentialComplex$num_inconsistent_stars()
```

Returns: An integer value storing the number of stars containing at least one inconsistent simplex.

Method `num_simplices()`**:**

Usage:

```
TangentialComplex$num_simplices()
```

Returns: An integer value storing the total number of simplices in stars (including duplicates that appear in several stars).

Method `num_vertices()`**:**

Usage:

```
TangentialComplex$num_vertices()
```

Returns: An integer value storing the number of vertices.

Method `set_max_squared_edge_length()`**:** Sets the maximal possible squared edge length for the edges in the triangulations.

Usage:

```
TangentialComplex$set_max_squared_edge_length(max_squared_edge_length)
```

Arguments:

`max_squared_edge_length` A numeric value specifying the maximal possible squared edge length.

Details: If the maximal edge length value is too low, the `$compute_tangential_complex()` method will throw an exception in debug mode.

Returns: The updated `TangentialComplex` class itself invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TangentialComplex$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Clément Jamin

See Also

Other filtrations and reconstructions: AlphaComplex, RipsComplex, WitnessComplex

Examples

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$compute_tangential_complex()
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
st <- tc$compute_tangential_complex()$create_simplex_tree()
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
st <- tc$compute_tangential_complex()$create_simplex_tree()
tc$get_point(1)
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$compute_tangential_complex()
tc$num_inconsistent_simplices()
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$compute_tangential_complex()
tc$num_inconsistent_stars()
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$compute_tangential_complex()
tc$num_simplices()
```

```
X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$compute_tangential_complex()
```

```
tc$num_vertices()


X <- seq_circle(10)
tc <- TangentialComplex$new(points = X, intrinsic_dim = 1)
tc$set_max_squared_edge_length(1)
```

`Tomato` *Clustering: Tomato*

Description

This clustering algorithm needs a neighborhood graph on the points, and an estimation of the density at each point. A few possible graph constructions and density estimators are provided for convenience, but it is perfectly natural to provide your own.

Super class

`rgudhi::PythonClass` -> `Tomato`

Methods

Public methods:

- `Tomato$new()`
- `Tomato$fit()`
- `Tomato$fit_predict()`
- `Tomato$set_n_clusters()`
- `Tomato$get_n_clusters()`
- `Tomato$set_merge_threshold()`
- `Tomato$get_merge_threshold()`
- `Tomato$get_labels()`
- `Tomato$plot_diagram()`
- `Tomato$clone()`

Method `new()`**:** The `Tomato` constructor.

Usage:

```
Tomato$new(
  graph_type = c("knn", "radius", "manual"),
  density_type = c("logDTM", "DTM", "logKDE", "KDE", "manual"),
  n_clusters = NULL,
  merge_threshold = NULL,
  ...
)
```

Arguments:

`graph_type` A string specifying the method to compute the neighboring graph. Choices are `"knn"`, `"radius"` or `"manual"`. Defaults to `"knn"`.

`density_type` A string specifying the choice of density estimator. Choicea are `"logDTM"`, `"DTM"`, `"logKDE"` or `"manual"`. When you have many points, `"KDE"` and `"logKDE"` tend to be slower. Defaults to `"logDTM"`

`n_clusters` An integer value specifying the number of clusters. Defaults to `NULL`, i.e. no merging occurs and we get the maximal number of clusters.

`merge_threshold` A numeric value specifying the minimum prominence of a cluster so it doesn't get merged. Defaults to `NULL`, i.e. no merging occurs and we get the maximal number of clusters.

`...` Extra parameters passed to `KNearestNeighbors` and `DTMDensity`.

Returns: An object of class `Tomato`.

Method `fit()`**:** Runs the Tomato algorithm on the provided data.

Usage:

```
Tomato$fit(X, y = NULL, weights = NULL)
```

Arguments:

`X` Either a numeric matrix specifying the coordinates (in column) of each point (in row) or a **full** distance matrix if `metric == "precomputed"` or a list of neighbors for each point if `graph_type == "manual"`. The number of points is currently limited to about 2 billion.

`y` Not used, present here for API consistency with **scikit-learn** by convention.

`weights` A numeric vector specifying a density estimate at each point. Used only if `density_type == "manual"`.

Returns: The updated `Tomato` class itself invisibly.

Method `fit_predict()`**:** Runs the Tomato algorithm on the provided data **and** returns the class memberships.

Usage:

```
Tomato$fit_predict(X, y = NULL, weights = NULL)
```

Arguments:

`X` Either a numeric matrix specifying the coordinates (in column) of each point (in row) or a **full** distance matrix if `metric == "precomputed"` or a list of neighbors for each point if `graph_type == "manual"`. The number of points is currently limited to about 2 billion.

`y` Not used, present here for API consistency with **scikit-learn** by convention.

`weights` A numeric vector specifying a density estimate at each point. Used only if `density_type == "manual"`.

Returns: An integer vector storing the class memberships.

Method `set_n_clusters()`**:** Sets the number of clusters which automatically adjusts class memberships.

Usage:

```
Tomato$set_n_clusters(n_clusters)
```

Arguments:

`n_clusters` An integer value specifying the number of clusters.

Returns: The updated `Tomato` class itself invisibly.

Method `get_n_clusters()`**:** Gets the number of clusters.

Usage:

```
Tomato$get_n_clusters()
```

Returns: The number of clusters.

Method `set_merge_threshold()`**:** Sets the threshold for merging clusters which automatically adjusts class memberships.

Usage:

```
Tomato$set_merge_threshold(merge_threshold)
```

Arguments:

`merge_threshold` A numeric value specifying the threshold for merging clusters.

Returns: The updated `Tomato` class itself invisibly.

Method `get_merge_threshold()`**:** Gets the threshold for merging clusters.

Usage:

```
Tomato$get_merge_threshold()
```

Returns: The threshold for merging clusters.

Method `get_labels()`**:** Gets the class memberships.

Usage:

```
Tomato$get_labels()
```

Returns: An integer vector storing the class memberships.

Method `plot_diagram()`**:** Computes the persistence diagram of the merge tree of the initial clusters. This is a convenient graphical tool to help decide how many clusters we want.

Usage:

```
Tomato$plot_diagram()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Tomato$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Marc Glisse

Examples

```
X <- seq_circle(100)
cl <- Tomato$new()
cl$fit_predict(X)
cl$set_n_clusters(2)
cl$get_labels()
```

TopologicalVector *Vector Representation: Topological Vector*

Description

Computes topological vectors from a list of persistence diagrams. The topological vector associated to a persistence diagram is the sorted vector of a slight modification of the pairwise distances between the persistence diagram points. See https://diglib.eg.org/handle/10.1111/cgf12692 for more details.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::VectorRepresentationStep` -> `TopologicalVector`

Methods

Public methods:

- `TopologicalVector$new()`
- `TopologicalVector$clone()`

Method `new()`**:** The `TopologicalVector` constructor.

Usage:

```
TopologicalVector$new(threshold = 10)
```

Arguments:

`threshold` An integer value specifying the number of distances to keep. Defaults to `10L`. This is the dimension of the topological vector. If `-1`, this threshold is computed from the list of persistence diagrams by considering the one with the largest number of points and using the dimension of its corresponding topological vector as threshold.

Returns: An object of class `TopologicalVector`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TopologicalVector$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
tv <- TopologicalVector$new()
tv$apply(dgm)
tv$fit_transform(list(dgm))
```

---

torus *Sampling on the Torus*

Description

The user should provide the number of points `n_samples` to be generated on the torus and the dimension `dim` of the torus on which points would be generated in $R^{2\mathrm{dim}}$. The `sample` argument is optional and is set to `"random"` by default. The generated points are returned as an array of shape $n_{\mathrm{samples}} \times R^{2\mathrm{dim}}$.

Usage

```
torus(n_samples, dim, sample = c("random", "grid"))
```

Arguments

`n_samples`	An integer value specifying the sample size.
`dim`	An integer value specifying the dimension $R^{2\mathrm{dim}}$ of the torus.
`sample`	A string specifying the sampling type. Choices are `"random"` or `"grid"`. Defaults to `"random"`.

Value

A numeric array of shape $n_{\mathrm{samples}} \times R^{2\mathrm{dim}}$ storing the sampled points.

Examples

```
torus(10, 1)
```

WassersteinDistance *Metrics: Wasserstein Distance*

Description

Computes the Wasserstein distance matrix from a list of persistence diagrams.

Super classes

`rgudhi::PythonClass` -> `rgudhi::SKLearnClass` -> `rgudhi::MetricStep` -> `WassersteinDistance`

Methods

Public methods:

- `WassersteinDistance$new()`
- `WassersteinDistance$clone()`

Method `new()`**:** The `WassersteinDistance` constructor.

Usage:

```
WassersteinDistance$new(
  order = 1,
  internal_p = Inf,
  mode = c("hera", "pot"),
  delta = 0.01,
  n_jobs = 1
)
```

Arguments:

`order` An integer value specifying the exponent of the Wasserstein distance. Defaults to `1.0`.

`internal_p` An integer value specifying the ground metric on the (upper-half) plane (i.e. the norm ℓ_p in R^2). Defaults to `Inf`.

`mode` A string specifying the method for computing the Wasserstein distance. Choices are either `"pot"` or `"hera"`. Defaults to `"hera"`.

`delta` A numeric value specifying the relative error $1 + \delta$. Defaults to `0.01`. Used only if `mode == "hera"`.

`n_jobs` An integer value specifying the number of jobs to use for the computation. Defaults to `1L`.

Returns: An object of class `WassersteinDistance`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
WassersteinDistance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Mathieu Carrière

Examples

```
X <- seq_circle(10)
ac <- AlphaComplex$new(points = X)
st <- ac$create_simplex_tree()
dgm <- st$compute_persistence()$persistence_intervals_in_dimension(0)
ds <- DiagramSelector$new(use = TRUE)
dgm <- ds$apply(dgm)
dis <- WassersteinDistance$new()
dis$apply(dgm, dgm)
dis$fit_transform(list(dgm))
```

WitnessComplex *R6 Class for Witness Complex*

Description

A Witness complex $\mathrm{Wit}(W, L)$ is a simplicial complex defined on two sets of points in R^D. The data structure is described in Boissonnat and Maria (2014).

Details

The class constructs a (weak) witness complex for a given table of nearest landmarks with respect to witnesses.

References:

Boissonnat J, Maria C (2014). "The simplex tree: An efficient data structure for general simplicial complexes." *Algorithmica*, **70**(3), 406–427.

Super class

`rgudhi::PythonClass` -> `WitnessComplex`

Methods

Public methods:

- `WitnessComplex$new()`
- `WitnessComplex$create_simplex_tree()`
- `WitnessComplex$clone()`

Method `new()`**:** The WitnessComplex constructor.

Usage:

```
WitnessComplex$new(nearest_landmark_table)
```

Arguments:

`nearest_landmark_table` A list of tibble::tibbles specifying for each *witness* `w`, the ordered list of nearest landmarks with id in column `nearest_landmark` and distance to `w` in column `distance`.

Returns: A `WitnessComplex` object storing the Witness complex.

Method `create_simplex_tree()`**:**

Usage:

```
WitnessComplex$create_simplex_tree(max_alpha_square = Inf)
```

Arguments:

`max_alpha_square` The maximum relaxation parameter. Defaults to `Inf`.

Returns: A `SimplexTree` object storing the computed simplex tree created from the Delaunay triangulation.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
WitnessComplex$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Siargey Kachanovich

See Also

Other filtrations and reconstructions: `AlphaComplex`, `RipsComplex`, `TangentialComplex`

Examples

```
withr::with_seed(1234, {
  l <- list(
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    ),
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    )
  )
})
wc <- WitnessComplex$new(nearest_landmark_table = l)
wc
```

```
withr::with_seed(1234, {
  l <- list(
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    ),
    tibble::tibble(
      nearest_landmark = sample.int(10),
      distance = sort(rexp(10))
    )
  )
})
wc <- WitnessComplex$new(nearest_landmark_table = l)
st <- wc$create_simplex_tree()
st$num_vertices()
```

Index