Representation of functional Data

Before beginning to use the functionalities of the package, it is necessary to represent the data in functional form, using one of the following classes, which allow the visualization, evaluation and treatment of the data in a simple way, using the advantages of the object-oriented programming.

Discrete representation

A functional datum may be treated using a non-parametric representation, storing the values of the functions in a finite grid of points. The FDataGrid class supports multivariate functions using this approach. In the discretized function representation example it is shown the creation and basic visualisation of a FDataGrid.

`skfda.representation.grid.FDataGrid`(data_matrix)	Represent discretised functional data.

Functional data grids may be evaluated using interpolation, as it is shown in the interpolation example. The following class allows interpolation with different splines.

`skfda.representation.interpolation.SplineInterpolator`([...])	Spline interpolator of `FDataGrid`.

Basis representation

The package supports a parametric representation using a linear combination of elements of a basis function system.

`skfda.representation.basis.FDataBasis`(basis, ...)	Basis representation of functional data.

The following classes are used to define different basis systems.

`skfda.representation.basis.BSpline`([...])	BSpline basis.
`skfda.representation.basis.Fourier`([...])	Fourier basis.
`skfda.representation.basis.Monomial`([...])	Monomial basis.

Generic representation

Functional objects of the package are instances of FData, which contains the common attributes and methods used in all representations. This is an abstract class and cannot be instantiated directly, because it does not specify the representation of the data. Many of the package's functionalities receive an element of this class as an argument.

`skfda.representation.FData`(extrapolation, ...)	Defines the structure of a functional data object.

Extrapolation

All representations of functional data allow evaluation outside of the original interval using extrapolation methods.

skfda.representation.FData

class **skfda.representation.FData**(*extrapolation*, *dataset_label*, *axes_labels*, *keepdims*)
[source]

Defines the structure of a functional data object.

nsamples

Number of samples.

Type: int

ndim_domain

Dimension of the domain.

Type: int

ndim_image

Dimension of the image.

Type: int

extrapolation

Default extrapolation mode.

Type: Extrapolation

dataset_label

name of the dataset.

Type: str

axes_labels

list containing the labels of the different axis. The first element is the x label, the second the y label and so on.

Type: list

`keepdims`

Default value of argument keepdims in `evaluate()`.

Type: bool

`__init__`(*extrapolation*, *dataset_label*, *axes_labels*, *keepdims*) [source]

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(extrapolation, dataset_label, ...)	Initialize self.
`argsort`([ascending, kind])	Return the indices that would sort this array.
`astype`(dtype[, copy])	Cast to a NumPy array with 'dtype'.
`compose`(fd, *[, eval_points])	Composition of functions.
`concatenate`(other)	Join samples from a similar FData object.
`copy`(**kwargs)	Make a copy of the object.
`derivative`([order])	Differentiate a FData object.
`dropna`()	Return ExtensionArray without NA values
`evaluate`(eval_points, *[, derivative, ...])	Evaluate the object or its derivatives at a list
`factorize`([na_sentinel])	Encode the extension array as an enumerate
`fillna`([value, method, limit])	Fill NA/NaN values using the specified meth
`generic_plotting_checks`([fig, ax, nrows, ncols])	Check the arguments passed to both `plot` a
`isna`()	A 1-D array indicating if each value is missin
`mean`()	Compute the mean of all the samples.
`plot`([chart, derivative, fig, ax, nrows, ...])	Plot the FDatGrid object.
`repeat`(repeats[, axis])	Repeat elements of a ExtensionArray.
`searchsorted`(value[, side, sorter])	Find indices where elements should be inser
`set_figure_and_axes`(nrows, ncols)	Set figure and its axes.
`set_labels`([fig, ax, patches])	Set labels if any.

`shift` (shifts, *[, restrict_domain, ...])	Perform a shift of the curves.
`take` (indices[, allow_fill, fill_value])	Take elements from an array.
`to_basis` (basis[, eval_points])	Return the basis representation of the object
`to_grid` ([eval_points])	Return the discrete representation of the obj
`to_numpy` ()	Returns a numpy array with the objects
`unique` ()	Compute the ExtensionArray of unique value

Attributes

`domain_range`	Return the domain range of the object
`dtype`	An instance of 'ExtensionDtype'.
`extrapolation`	Return default type of extrapolation.
`extrapolator_evaluator`	Return the evaluator constructed by the extrapolator.
`nbytes`	The number of bytes needed to store this object in memory.
`ndim`	Return number of dimensions of the functional data.
`ndim_codomain`	Return number of dimensions of the codomain.
`ndim_domain`	Return number of dimensions of the domain.
`ndim_image`	Return number of dimensions of the image.
`nsamples`	Return the number of samples.
`shape`	Return a tuple of the array dimensions.

skfda.representation.interpolation.SplineInterpolator

class **skfda.representation.interpolation.SplineInterpolator**(*interpolation_order=1, smoothness_parameter=0.0, monotone=False*) [source]

Spline interpolator of `FDataGrid`.

Spline interpolator of discretized functional objects. Implements different interpolation methods based in splines, using the sample points of the grid as nodes to interpolate.

See the interpolation example to a detailled explanation.

interpolator_order

Order of the interpolation, 1 for linear interpolation, 2 for cuadratic, 3 for cubic and so on. In case of curves and surfaces there is available interpolation up to degree 5. For higher dimensional objects only linear or nearest interpolation is available. Default lineal interpolation.

Type: int, optional

smoothness_parameter

Penalisation to perform smoothness interpolation. Option only available for curves and surfaces. If 0 the residuals of the interpolation will be 0. Defaults 0.

Type: float, optional

monotone

Performs monotone interpolation in curves using a PCHIP interpolator. Only valid for curves (domain dimension equal to 1) and interpolation order equal to 1 or 3. Defaults false.

Type: boolean, optional

__init__(*interpolation_order=1, smoothness_parameter=0.0, monotone=False*) [source]

Constructor of the SplineInterpolator.

Parameters:

- **interpolator_order** (*int*, *optional*) – Order of the interpolation, 1 for linear interpolation, 2 for cuadratic, 3 for cubic and so on. In case of curves and surfaces there is available interpolation up to degree 5. For higher dimensional objects only linear or nearest interpolation is available. Default lineal interpolation.
- **smoothness_parameter** (*float*, *optional*) – Penalisation to perform smoothness interpolation. Option only available for curves and surfaces. If 0 the residuals of the interpolation will be 0. Defaults 0.
- **monotone** (*boolean*, *optional*) – Performs monotone interpolation in curves using a PCHIP interpolator. Only valid for curves (domain dimension equal to 1) and interpolation order equal to 1 or 3. Defaults false.

Methods

`__init__`([interpolation_order, ...])	Constructor of the SplineInterpolator.
`evaluator`(fdatagrid)	Construct a SplineInterpolatorEvaluator used in the evalu

Attributes

`interpolation_order`	Returns the interpolation order
`monotone`	Returns flag to perform monotone interpolation
`smoothness_parameter`	Returns the smoothness parameter

Extrapolation

This module contains the extrapolators used to evaluate points outside the domain range of `FDataBasis` or `FDataGrid`. See Extrapolation Example for detailed explanation.

Extrapolation Methods

The following classes are used to define common methods of extrapolation.

`skfda.representation.extrapolation.BoundaryExtrapolation`	Extends the domain range using the
`skfda.representation.extrapolation.ExceptionExtrapolation`	Raise and exception.
`skfda.representation.extrapolation.FillExtrapolation`(...)	Values outside the domain range w
`skfda.representation.extrapolation.PeriodicExtrapolation`	Extends the domain range periodic

Custom Extrapolation

Custom extrapolators could be done subclassing `EvaluatorConstructor`.

`skfda.representation.evaluator.EvaluatorConstructor`	Constructor of an evaluator.
`skfda.representation.evaluator.Evaluator`	Structure of an evaluator.

skfda.representation.extrapolation.ExceptionExtrapolatio

class **skfda.representation.extrapolation.ExceptionExtrapolation** [source]

Raise and exception.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import ExceptionExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = ExceptionExtrapolation()
>>> try:
...     fd([-.5, 0, 1.5]).round(3)
... except ValueError as e:
...     print(e)
Attempt to evaluate 2 points outside the domain range.
```

This extrapolator is equivalent to the string *"exception"*.

```
>>> fd.extrapolation = 'exception'
>>> try:
...     fd([-.5, 0, 1.5]).round(3)
... except ValueError as e:
...     print(e)
Attempt to evaluate 2 points outside the domain range.
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.PeriodicExtrapolation

class **skfda.representation.extrapolation.PeriodicExtrapolation** [source]

Extends the domain range periodically.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import PeriodicExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = PeriodicExtrapolation()
>>> fd([-.5, 0, 1.5]).round(3)
array([[-0.724,  0.976, -0.724],
       [-1.086,  0.759, -1.086]])
```

This extrapolator is equivalent to the string *"periodic"*

```
>>> fd.extrapolation = 'periodic'
>>> fd([-.5, 0, 1.5]).round(3)
array([[-0.724,  0.976, -0.724],
       [-1.086,  0.759, -1.086]])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.BoundaryExtrapolatio

class **skfda.representation.extrapolation.BoundaryExtrapolation** [source]

Extends the domain range using the boundary values.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import BoundaryExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = BoundaryExtrapolation()
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.976,  0.976,  0.797],
       [ 0.759,  0.759,  1.125]])
```

This extrapolator is equivalent to the string *"bounds"*.

```
>>> fd.extrapolation = 'bounds'
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.976,  0.976,  0.797],
       [ 0.759,  0.759,  1.125]])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.FillExtrapolation

class **skfda.representation.extrapolation.FillExtrapolation**(*fill_value*) [source]

Values outside the domain range will be filled with a fixed value.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import FillExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = FillExtrapolation(0)
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.   ,  0.976,  0.   ],
       [ 0.   ,  0.759,  0.   ]])
```

The previous extrapolator is equivalent to the string *"zeros"*. In the same way FillExtrapolation(np.nan) is equivalent to *"nan"*.

```
>>> fd.extrapolation = "nan"
>>> fd([-.5, 0, 1.5]).round(3)
array([[   nan,  0.976,    nan],
       [   nan,  0.759,    nan]])
```

__init__(*fill_value*) [source]

Returns the evaluator used by `FData`.

Returns: Evaluator of the periodic extrapolation.

Return type: (`Evaluator`)

Methods

`__init__`(fill_value)	Returns the evaluator used by `FData`.

`evaluator`(fdata)	Construct an evaluator.

Attributes

`fill_value`	Returns the fill value of the extrapolation

Registration

We see often that variation in functional observations involves phase and amplitude variation, which may hinder further analysis. That problem is treated during the registration process. This module contains procedures for the registration of the data.

Shift Registration

Many of the issues involved in registration can be solved by considering the simplest case, a simple shift in the time scale. This often happens because the time at which the recording process begins is arbitrary, and is unrelated to the beginning of the interesting segment of the data. In the Shift Registration Example it is shown the basic usage of this methods applied to periodic data.

`skfda.preprocessing.registration.shift_registration`(fd, *)	Perform shift registration of
`skfda.preprocessing.registration.shift_registration_deltas`(fd, *)	Return the lists of shifts usec

Landmark Registration

Landmark registration aligns features applying a transformation of the time that takes all the times of a given feature into a common value.

The simplest case in which each sample presents a unique landmark can be solved by performing a translation in the time scale. See the Landmark Shift Example.

`skfda.preprocessing.registration.landmark_shift`(fd, ...)	Perform a shift of the curves to
`skfda.preprocessing.registration.landmark_shift_deltas`(fd, ...)	Returns the corresponding shifts

The general case of landmark registration may present multiple landmarks for each sample and a non-linear transformation in the time scale should be applied. See the Landmark Registration Example

`skfda.preprocessing.registration.landmark_registration`(fd, ...)	Perform landmark regist
`skfda.preprocessing.registration.landmark_registration_warping`(fd, ...)	Calculate the transform

Elastic Registration

The elastic registration is a novel approach to this problem that uses the properties of the Fisher-Rao metric to perform the alignment of the curves. In the examples of pairwise alignment and elastic registration is shown a brief introduction to this topic along the usage of the corresponding functions.

`skfda.preprocessing.registration.elastic_registration`(...)	Align a FDatagrid using the
`skfda.preprocessing.registration.elastic_registration_warping`(...)	Calculate the warping to alig

The module contains some routines related with the elastic registration, making a transformation of the sampling, computing different means or distances based on the elastic framework.

`skfda.preprocessing.registration.elastic_mean`(...)	Compute the karcher mean under the elastic
`skfda.preprocessing.registration.warping_mean`(...)	Compute the karcher mean of a set of warpi
`skfda.preprocessing.registration.to_srsf`(...)	Calculate the square-root slope function (SR
`skfda.preprocessing.registration.from_srsf`(...)	Given a SRSF calculate the corresponding fu

Amplitude and Phase Decomposition

The amplitude and phase variation may be quantified by comparing a sample before and after registration. The package contains an implementation of the decomposition procedure developed by *Kneip and Ramsay (2008)*.

`skfda.preprocessing.registration.mse_decomposition`(...)	Compute mean square error measures f

Utility functions

There are some other method related with the registration problem in this module.

`skfda.preprocessing.registration.invert_warping`(...)	Compute the inverse of a diffeom
`skfda.preprocessing.registration.normalize_warping`(warping)	Rescale a warping to normalize th

References

- Ramsay, J., Silverman, B. W. (2005). Functional Data Analysis. Springer.
- Kneip, Alois & Ramsay, James. (2008). Quantifying amplitude and phase variation. Journal of the American Statistical Association.
- Ramsay, J., Hooker, G. & Graves S. (2009). Functional Data Analysis with R and Matlab. Springer.
- Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. Springer.

- J. S. Marron, James O. Ramsay, Laura M. Sangalli and Anuj Srivastava (2015). Functional Data Analysis of Amplitude and Phase Variation. Statistical Science 2015, Vol. 30, No. 4

skfda.preprocessing.registration.shift_registration

skfda.preprocessing.registration.shift_registration(*fd*, *, *maxiter=5*, *tol=0.01*, *restrict_domain=False*, *extrapolation=None*, *step_size=1*, *initial=None*, *eval_points=None*, **kwargs*) [source]

Perform shift registration of the curves.

Realizes a registration of the curves, using shift aligment, as is defined in [RS05-7-2]. Calculates δ_i for each sample such that $x_i(t + \delta_i)$ minimizes the least squares criterion:

$$\text{REGSSE} = \sum_{i=1}^{N} \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)]^2 ds$$

Estimates the shift parameter δ_i iteratively by using a modified Newton-Raphson algorithm, updating the mean in each iteration, as is described in detail in [RS05-7-9-1].

Parameters:

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (*int*, *optional*) – Maximun number of iterations. Defaults to 5.
- **tol** (*float*, *optional*) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(\nu)} - \delta_i^{(\nu-1)}| < tol$. Default sets to 1e-2.
- **restrict_domain** (*bool*, *optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (str or `Extrapolation`, optional) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See :module: *extrapolation* to obtain more information.
- **step_size** (*int or float*, *optional*) – Parameter to adjust the rate of convergence in the Newton-Raphson algorithm, see [RS05-7-9-1]. Defaults to 1.
- **initial** (*array_like*, *optional*) – Initial estimation of shifts. Default uses a list of zeros for the initial shifts.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object to integrate. If None is passed it calls numpy.linspace in FDataBasis and uses the *sample_points* in FDataGrids.
- ****kwargs** – Keyword arguments to be passed to `shift()`.

Returns: `FData` A `FData` object with the curves registered.

Raises: `ValueError` – If the initial array has different length than the number of samples.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.basis import Fourier
>>> from skfda.preprocessing.registration import shift_registration
>>> fd = make_sinusoidal_process(n_samples=2, error_std=0, random_state=1)
```

Registration of data in discretized form:

```
>>> shift_registration(fd)
FDataGrid(...)
```

Registration of data in basis form:

```
>>> fd = fd.to_basis(Fourier())
>>> shift_registration(fd)
FDataBasis(...)
```

References

[RS05-7-2] Ramsay, J., Silverman, B. W. (2005). Shift registration. In *Functional Data Analysis* (pp. 129-132). Springer.

[RS05-7-9-1] *(1, 2)* Ramsay, J., Silverman, B. W. (2005). Shift registration by the Newton-Raphson algorithm. In *Functional Data Analysis* (pp. 142-144). Springer.

skfda.preprocessing.registration.shift_registration_deltas

skfda.preprocessing.registration.shift_registration_deltas(*fd*, ***, *maxiter=5*, *tol=0.01*, *restrict_domain=False*, *extrapolation=None*, *step_size=1*, *initial=None*, *eval_points=None*) [source]

Return the lists of shifts used in the shift registration procedure.

Realizes a registration of the curves, using shift aligment, as is defined in [RS05-7-2-1]. Calculates δ_i for each sample such that $x_i(t + \delta_i)$ minimizes the least squares criterion:

$$\text{REGSSE} = \sum_{i=1}^{N} \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)]^2 ds$$

Estimates the shift parameter δ_i iteratively by using a modified Newton-Raphson algorithm, updating the mean in each iteration, as is described in detail in [RS05-7-9-1-1].

Method only implemented for Funtional objects with domain and image dimension equal to 1.

Parameters:

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (*int*, *optional*) – Maximun number of iterations. Defaults to 5.
- **tol** (*float*, *optional*) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(\nu)} - \delta_i^{(\nu-1)}| < tol$. Default sets to 1e-2.
- **restrict_domain** (*bool*, *optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (str or `Extrapolation` , optional) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See :module: *extrapolation* to obtain more information.
- **step_size** (*int or float*, *optional*) – Parameter to adjust the rate of convergence in the Newton-Raphson algorithm, see [RS05-7-9-1-1]. Defaults to 1.
- **initial** (*array_like*, *optional*) – Initial estimation of shifts. Default uses a list of zeros for the initial shifts.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object to integrate. If None is passed it calls numpy.linspace in FDataBasis and uses the *sample_points* in FDataGrids.

Returns: list with the shifts.

Return type: `numpy.ndarray`

Raises: `ValueError` – If the initial array has different length than the number of samples.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.basis import Fourier
>>> from skfda.preprocessing.registration import shift_registration_deltas
>>> fd = make_sinusoidal_process(n_samples=2, error_std=0, random_state=1)
```

Registration of data in discretized form:

```
>>> shift_registration_deltas(fd).round(3)
array([-0.022,  0.03 ])
```

Registration of data in basis form:

```
>>> fd = fd.to_basis(Fourier())
>>> shift_registration_deltas(fd).round(3)
array([-0.022,  0.03 ])
```

References

[RS05-7-2-1] Ramsay, J., Silverman, B. W. (2005). Shift registration. In *Functional Data Analysis* (pp. 129-132). Springer.

[RS05-7-9-1-1] *(1, 2)* Ramsay, J., Silverman, B. W. (2005). Shift registration by the Newton-Raphson algorithm. In *Functional Data Analysis* (pp. 142-144). Springer.

skfda.preprocessing.registration.landmark_shift

skfda.preprocessing.registration.landmark_shift(*fd*, *landmarks*, *location=None*, ***, *restrict_domain=False*, *extrapolation=None*, *eval_points=None*, ***kwargs*) [source]

Perform a shift of the curves to align the landmarks.

Let t^* the time where the landmarks of the curves will be aligned, t_i the location of the landmarks for each curve and $\delta_i = t_i - t^*$.

The registered samples will have their feature aligned.

$$x_i^*(t^*) = x_i(t^* + \delta_i) = x_i(t_i)$$

Parameters:

- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List with the landmarks of the samples.
- **location** (*numeric or callable*, *optional*) – Defines where the landmarks will be alligned. If a numeric value is passed the landmarks will be alligned to it. In case of a callable is passed the location will be the result of the the call, the function should be accept as an unique parameter a numpy array with the list of landmarks. By default it will be used as location $\frac{1}{2}(max(\text{landmarks}) + min(\text{landmarks}))$ wich minimizes the max shift.
- **restrict_domain** (*bool*, *optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (*str or Extrapolation*, *optional*) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See extrapolation to more information.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated in `shift()` .
- ****kwargs** – Keyword arguments to be passed to `shift()` .

Returns: Functional data object with the registered samples.

Return type: `FData`

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_shift
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
```

The function will return the sample registered

```
>>> landmark_shift(fd, landmarks)
FDataGrid(...)
```

skfda.preprocessing.registration.landmark_shift_deltas

skfda.preprocessing.registration.landmark_shift_deltas(*fd*, *landmarks*, *location=None*)
[source]

Returns the corresponding shifts to align the landmarks of the curves.

Let t^* the time where the landmarks of the curves will be aligned, and t_i the location of the landmarks for each curve. The function will calculate the corresponding δ_i shuch that $t_i = t^* + \delta_i$.

This procedure will work independent of the dimension of the domain and the image.

Parameters:

- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List with the landmarks of the samples.
- **location** (*numeric or callable, optional*) – Defines where the landmarks will be alligned. If a numer or list is passed the landmarks will be alligned to it. In case of a callable is passed the location will be the result of the the call, the function should be accept as an unique parameter a numpy array with the list of landmarks. By default it will be used as location $\frac{1}{2}(max(\text{landmarks}) + min(\text{landmarks}))$ wich minimizes the max shift.

Returns: Array containing the corresponding shifts.

Return type: `numpy.ndarray`

Raises: `ValueError` – If the list of landmarks does not match with the number of samples.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_shift_deltas
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
```

The function will return the corresponding shifts

```
>>> shifts = landmark_shift_deltas(fd, landmarks)
>>> shifts.round(3)
array([ 0.25 , -0.25 , -0.231])
```

The registered samples can be obtained with a shift

```
>>> fd.shift(shifts)
FDataGrid(...)
```

skfda.preprocessing.registration.landmark_registration

skfda.preprocessing.registration.landmark_registration(*fd, landmarks, *, location=None, eval_points=None*) [source]

Perform landmark registration of the curves.

Let t_{ij} the time where the sample i has the feature j and t_j^* the new time for the feature. The registered samples will have their features aligned, i.e., $x_i^*(t_j^*) = x_i(t_{ij})$.

See [RS05-7-3] for a detailed explanation.

Parameters:
- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List containing landmarks for each samples.
- **location** (*array_like*, *optional*) – Defines where the landmarks will be alligned. By default it will be used as location the mean of the landmarks.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain a discrete representation of the object. In case of objects with multidimensional domain a list axis with points of evaluation for each dimension.

Returns: FData with the functional data object registered.

Return type: `FData`

References:

[RS05-7-3] Ramsay, J., Silverman, B. W. (2005). Feature or landmark registration. In *Functional Data Analysis* (pp. 132-136). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_registration
>>> from skfda.representation.basis import BSpline
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, n_modes=2, random_state=9)
>>> landmarks = make_multimodal_landmarks(n_samples=3, n_modes=2,
...                                       random_state=9)
>>> landmarks = landmarks.squeeze()
```

The function will return the registered curves

```
>>> landmark_registration(fd, landmarks)
FDataGrid(...)
```

This method will work for FDataBasis as for FDataGrids

```
>>> fd = fd.to_basis(BSpline(nbasis=12, domain_range=(-1,1)))
>>> landmark_registration(fd, landmarks)
FDataBasis(...)
```

skfda.preprocessing.registration.landmark_registration_war

skfda.preprocessing.registration.landmark_registration_warping(*fd*, *landmarks*, ***, *location=None*, *eval_points=None*) [source]

Calculate the transformation used in landmark registration.

Let t_{ij} the time where the sample i has the feature j and t_j^* the new time for the feature. The warping function will transform the new time in the old time, i.e., $h_i(t_j^*) = t_{ij}$. The registered samples can be obtained as $x_i^*(t) = x_i(h_i(t))$.

See [RS05-7-3-1] for a detailed explanation.

Parameters:
- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List containing landmarks for each samples.
- **location** (*array_like*, *optional*) – Defines where the landmarks will be alligned. By default it will be used as location the mean of the landmarks.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain a discrete representation of the object.

Returns: FDataGrid with the warpings function needed to register the functional data object.

Return type: `FDataGrid`

Raises: `ValueError` – If the object to be registered has domain dimension greater than 1 or the list of landmarks or locations does not match with the number of samples.

References:

[RS05-7-3-1] Ramsay, J., Silverman, B. W. (2005). Feature or landmark registration. In *Functional Data Analysis* (pp. 132-136). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_registration_warping
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, n_modes=2, random_state=9)
>>> landmarks = make_multimodal_landmarks(n_samples=3, n_modes=2,
...                                       random_state=9)
>>> landmarks = landmarks.squeeze()
```

The function will return the corresponding warping function

```
>>> warping = landmark_registration_warping(fd, landmarks)
>>> warping
FDataGrid(...)
```

The registered function can be obtained using function composition

```
>>> fd.compose(warping)
FDataGrid(...)
```

skfda.preprocessing.registration.mse_decomposition

skfda.preprocessing.registration.mse_decomposition(*original_fdata, registered_fdata, warping_function=None, *, eval_points=None*) [source]

Compute mean square error measures for amplitude and phase variation.

Once the registration has taken place, this function computes two mean squared error measures, one for amplitude variation, and the other for phase variation. It also computes a squared multiple correlation index of the amount of variation in the unregistered functions is due to phase.

Let $x_i(t)$, $y_i(t)$ be the unregistered and registered functions respectively. The total mean square error measure (see [RGS09-8-5]) is defined as

$$\mathrm{MSE}_{total} = \frac{1}{N}\sum_{i=1}^{N}\int [x_i(t) - \overline{x}(t)]^2 dt$$

We define the constant C_R as

$$C_R = 1 + \frac{\frac{1}{N}\sum_i^N \int [Dh_i(t) - \overline{Dh}(t)][y_i^2(t) - \overline{y^2}(t)]dt}{\frac{1}{N}\sum_i^N \int y_i^2(t)dt}$$

Whose structure is related to the covariation between the deformation functions $Dh_i(t)$ and the squared registered functions $y_i^2(t)$. When these two sets of functions are independents $C_R = 1$, as in the case of shift registration.

The measures of amplitude and phase mean square error are

$$\mathrm{MSE}_{amp} = C_R \frac{1}{N}\sum_{i=1}^{N}\int \left[y_i(t) - \overline{y}(t)\right]^2 dt$$

$$\mathrm{MSE}_{phase} = \int \left[C_R \overline{y}^2(t) - \overline{x}^2(t)\right] dt$$

It can be shown that

$$\mathrm{MSE}_{total} = \mathrm{MSE}_{amp} + \mathrm{MSE}_{phase}$$

The squared multiple correlation index of the proportion of the total variation due to phase is defined as:

$$R^2 = \frac{\mathrm{MSE}_{phase}}{\mathrm{MSE}_{total}}$$

See [KR08-3] for a detailed explanation.

Parameters:

- **original_fdata** (`FData`) – Unregistered functions.
- **regfd** (`FData`) – Registered functions.
- **warping_function** (`FData`) – Warping functions.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated to obtain a discrete representation.

Returns: Tuple with amplitude mean square error MSE_{amp}, phase mean square error MSE_{phase}, squared correlation index R^2 and constant C_R.

Return type: `collections.namedtuple`

Raises: `ValueError` – If the curves do not have the same number of samples.

References

[KR08-3] Kneip, Alois & Ramsay, James. (2008). Quantifying amplitude and phase variation. In *Combining Registration and Fitting for Functional Models* (pp. 14-15). Journal of the American Statistical Association.

[RGS09-8-5] Ramsay J.O., Giles Hooker & Spencer Graves (2009). In *Functional Data Analysis with R and Matlab* (pp. 125-126). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import (landmark_registration_warping,
...                                mse_decomposition)
```

We will create and register data.

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
>>> warping = landmark_registration_warping(fd, landmarks)
>>> fd_registered = fd.compose(warping)
>>> mse_amp, mse_pha, rsq, cr = mse_decomposition(fd, fd_registered, warping)
```

Mean square error produced by the amplitude variation.

```
>>> f'{mse_amp:.6f}'
'0.000987'
```

In this example we can observe that the main part of the mean square error is due to the phase variation.

```
>>> f'{mse_pha:.6f}'
'0.115769'
```

Nearly 99% of the variation is due to phase.

```
>>> f'{rsq:.6f}'
'0.991549'
```

skfda.preprocessing.registration.to_srsf

skfda.preprocessing.registration.to_srsf(*fdatagrid*, *eval_points=None*) [source]

Calculate the square-root slope function (SRSF) transform.

Let $f_i : [a, b] \to \mathbb{R}$ be an absolutely continuous function, the SRSF transform is defined as

$$SRSF(f_i(t)) = sgn(f_i(t))\sqrt{|Df_i(t)|} = q_i(t)$$

This representation it is used to compute the extended non-parametric Fisher-Rao distance between functions, wich under the SRSF representation becomes the usual $\mathbb{L}^2$ distance between functions. See [SK16-4-6-1] .

Parameters:
- **fdatagrid** (`FDataGrid`) – Functions to be transformed.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.

Returns: SRSF functions.

Return type: `FDataGrid`

Raises: `ValueError` – If functions are multidimensional.

References

[SK16-4-6-1] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Square-Root Slope Function Representation* (pp. 91-93). Springer.

skfda.preprocessing.registration.from_srsf

skfda.preprocessing.registration.from_srsf(*fdatagrid*, *initial=None*, ***, *eval_points=None*)
[source]

Given a SRSF calculate the corresponding function in the original space.

Let $f_i : [a, b] \to \mathbb{R}$ be an absolutely continuous function, the SRSF transform is defined as

$$SRSF(f_i(t)) = sgn(f_i(t))\sqrt{|Df_i(t)|} = q_i(t)$$

This transformation is a mapping up to constant. Given the srsf and the initial value the original function can be obtained as

$$f_i(t) = f(a) + \int_a^t q(t)|q(t)|dt$$

This representation it is used to compute the extended non-parametric Fisher-Rao distance between functions, wich under the SRSF representation becomes the usual $\mathbb{L}^2$ distance between functions. See [SK16-4-6-2] .

Parameters:
- **fdatagrid** (`FDataGrid`) – SRSF to be transformed.
- **initial** (*array_like*) – List of values of initial values of the original functions.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.

Returns: Functions in the original space.

Return type: `FDataGrid`

Raises: `ValueError` – If functions are multidimensional.

References

[SK16-4-6-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Square-Root Slope Function Representation* (pp. 91-93). Springer.

skfda.preprocessing.registration.elastic_registration

skfda.preprocessing.registration.elastic_registration(*fdatagrid, template=None, *, lam=0.0, eval_points=None, fdatagrid_srsf=None, template_srsf=None, grid_dim=7, **kwargs*) [source]

Align a FDatagrid using the SRSF framework.

Let f be a function of the functional data object wich will be aligned to the template g. Calculates the warping wich minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = argmin_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended Fisher-Rao distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\dot{\gamma}} - SRSF(g)\|_{\mathbb{L}^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\dot{\gamma}} - 1\|_{\mathbb{L}^2}^2$$

Wich restrict the amount of elasticity employed in the alignment.

The registered function $f^*(t)$ can be calculated using the composition $f^*(t) = f(\gamma^*(t))$.

If the template is not specified it is used the Karcher mean of the set of functions under the elastic metric to perform the alignment, wich is the local minimum of the sum of squares of elastic distances. See `elastic_mean()`.

In [SK16-4-2] are described extensively the algorithms employed and the SRSF framework.

Parameters:

- **fdatagrid** (`FDataGrid`) – Functional data object to be aligned.
- **template** (`FDataGrid` , optional) – Template to align the curves. Can contain 1 sample to align all the curves to it or the same number of samples than the fdatagrid. By default it is used the elastic mean.
- **lam** (*float*, *optional*) – Controls the amount of elasticity. Defaults to 0.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid` , optional) – SRSF of the fdatagrid, may be passed to avoid repeated calculation.
- **template_srsf** (`FDataGrid` , optional) – SRSF of the template, may be passed to avoid repeated calculation.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- ****kwargs** – Named arguments to be passed to `elastic_mean()` .

Returns:

FDatagrid with the samples aligned to the template.

Return type:

(`FDataGrid`)

Raises:

`ValueError` – If functions are multidimensional or the number of samples are different.

References

[SK16-4-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.

skfda.preprocessing.registration.elastic_registration_warpi

skfda.preprocessing.registration.elastic_registration_warping(*fdatagrid, template=None, *, lam=0.0, eval_points=None, fdatagrid_srsf=None, template_srsf=None, grid_dim=7, **kwargs*) [source]

Calculate the warping to align a FDatagrid using the SRSF framework.

Let f be a function of the functional data object wich will be aligned to the template g. Calculates the warping wich minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = argmin_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended amplitude distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\dot{\gamma}} - SRSF(g)\|_{\mathbb{L}^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\dot{\gamma}} - 1\|_{\mathbb{L}^2}^2$$

Wich restrict the amount of elasticity employed in the alignment.

The registered function $f^*(t)$ can be calculated using the composition $f^*(t) = f(\gamma^*(t))$.

If the template is not specified it is used the Karcher mean of the set of functions under the Fisher-Rao metric to perform the alignment, wich is the local minimum of the sum of squares of elastic distances. See `elastic_mean()`.

In [SK16-4-3] are described extensively the algorithms employed and the SRSF framework.

Parameters:

- **fdatagrid** (`FDataGrid`) – Functional data object to be aligned.
- **template** (`FDataGrid` , optional) – Template to align the curves. Can contain 1 sample to align all the curves to it or the same number of samples than the fdatagrid. By default it is used the elastic mean.
- **lam** (*float*, *optional*) – Controls the amount of elasticity. Defaults to 0.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid` , optional) – SRSF of the fdatagrid, may be passed to avoid repeated calculation.
- **template_srsf** (`FDataGrid` , optional) – SRSF of the template, may be passed to avoid repeated calculation.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- ****kwargs** – Named arguments to be passed to `elastic_mean()` .

Returns: Warping to align the given fdatagrid to the template.

Return type: (`FDataGrid`)

Raises: `ValueError` – If functions are multidimensional or the number of samples are different.

References

[SK16-4-3] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.

skfda.preprocessing.registration.warping_mean

skfda.preprocessing.registration.warping_mean(*warping, *, iter=20, tol=1e-05, step_size=1.0, eval_points=None, return_shooting=False*) [source]

Compute the karcher mean of a set of warpings.

Let $\gamma_i i = 1 ... n$ be a set of warping functions $\gamma_i : [a, b] \to [a, b]$ in Γ, i.e., monotone increasing and with the restriction $\gamma_i(a) = a\, \gamma_i(b) = b$.

The karcher mean $\bar{\gamma}$ is defined as the warping that minimises locally the sum of Fisher-Rao squared distances. [SK16-8-3-2].

$$\bar{\gamma} = argmin_{\gamma \in \Gamma} \sum_{i=1}^{n} d_{FR}^2(\gamma, \gamma_i)$$

The computation is performed using the structure of Hilbert Sphere obtained after a transformation of the warpings, see [S11-3-3].

Parameters:

- **warping** (`FDataGrid`) – Set of warpings.
- **iter** (*int*) – Maximun number of interations. Defaults to 20.
- **tol** (*float*) – Convergence criterion, if the norm of the mean of the shooting vectors, $|\bar{v}| < tol$, the algorithm will stop. Defaults to 1e-5.
- **step_size** (*float*) – Step size ϵ used to update the mean. Default to 1.
- **eval_points** (*array_like*) – Discretisation points of the warpings.
- **shooting** (*boolean*) – If true it is returned a tuple with the mean and the shooting vectors, otherwise only the mean is returned.

Returns: (`FDataGrid`) Fdatagrid with the mean of the warpings. If shooting is True the shooting vectors will be returned in a tuple with the mean.

References

[SK16-8-3-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Template: Center of the Mean Orbit* (pp. 274-277). Springer.

[S11-3-3] Srivastava, Anuj et. al. Registration of Functional Data Using Fisher-Rao Metric (2011). In *Center of an Orbit* (pp. 9-10). arXiv:1103.3817v2.

skfda.preprocessing.registration.elastic_mean

skfda.preprocessing.registration.elastic_mean(*fdatagrid, *, lam=0.0, center=True, iter=20, tol=0.001, initial=None, eval_points=None, fdatagrid_srsf=None, grid_dim=7, **kwargs*)
[source]

Compute the karcher mean under the elastic metric.

Calculates the karcher mean of a set of functional samples in the amplitude space $\mathcal{A} = \mathcal{F}/\Gamma$.

Let q_i the corresponding SRSF of the observation f_i. The space $\mathcal{A}$ is defined using the equivalence classes $[q_i] = \{q_i \circ \gamma \| \gamma \in \Gamma\}$, where Γ denotes the space of warping functions. The karcher mean in this space is defined as

$$[\mu_q] = argmin_{[q] \in \mathcal{A}} \sum_{i=1}^{n} d_\lambda^2([q], [q_i])$$

Once $[\mu_q]$ is obtained it is selected the element of the equivalence class which makes the mean of the warpings employed be the identity.

See [SK16-8-3-1] and [S11-3].

Parameters:

- **fdatagrid** (`FDataGrid`) – Set of functions to compute the mean.
- **lam** (*float*) – Penalisation term. Defaults to 0.
- **center** (*boolean*) – If true it is computed the mean of the warpings and used to select a central mean. Defaults True.
- **iter** (*int*) – Maximun number of iterations. Defaults to 20.
- **tol** (*float*) – Convergence criterion, the algorithm will stop if :math: ´|mu^{(nu)} - mu^{(nu - 1)} |_2 / | mu^{(nu-1)} |_2 < tol´.
- **initial** (*float*) – Value of the mean at the starting point. By default takes the average of the initial points of the samples.
- **eval_points** (*array_like*) – Points of discretization of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid`) – SRSF if the fdatagrid, if it is passed it is not computed in the algorithm.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- **kwargs** (**) – Named options to be pased to `warping_mean()` .

Returns: FDatagrid with the mean of the functions.

Return type: (`FDataGrid`)

Raises: `ValueError` – If the object is multidimensional or the shape of the srsf do not match with the fdatagrid.

References

[SK16-8-3-1] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Karcher Mean of Amplitudes* (pp. 273-274). Springer.

[S11-3] Srivastava, Anuj et. al. Registration of Functional Data Using Fisher-Rao Metric (2011). In *Karcher Mean and Function Alignment* (pp. 7-10). arXiv:1103.3817v2.

skfda.preprocessing.registration.invert_warping

skfda.preprocessing.registration.invert_warping(*fdatagrid*, ***, *eval_points=None*)
[source]

Compute the inverse of a diffeomorphism.

Let $\gamma : [a, b] \to [a, b]$ be a function strictly increasing, calculates the corresponding inverse $\gamma^{-1} : [a, b] \to [a, b]$ such that $\gamma^{-1} \circ \gamma = \gamma \circ \gamma^{-1} = \gamma_{id}$.

Uses a PCHIP interpolator to compute approximately the inverse.

Parameters:
- **fdatagrid** (`FDataGrid`) – Functions to be inverted.
- **eval_points** – (array_like, optional): Set of points where the functions are interpolated to obtain the inverse, by default uses the sample points of the fdatagrid.

Returns: Inverse of the original functions.

Return type: `FDataGrid`

Raises: `ValueError` – If the functions are not strictly increasing or are multidimensional.

Examples

```
>>> import numpy as np
>>> from skfda import FDataGrid
>>> from skfda.preprocessing.registration import invert_warping
```

We will construct the warping $\gamma : [0, 1]$ wich maps t to t^3.
$rightarrow [0, 1]$

```
>>> t = np.linspace(0, 1)
>>> gamma = FDataGrid(t**3, t)
>>> gamma
FDataGrid(...)
```

We will compute the inverse.

```
>>> inverse = invert_warping(gamma)
>>> inverse
FDataGrid(...)
```

The result of the composition should be approximately the identity function .

```
>>> identity = gamma.compose(inverse)
>>> identity([0, 0.25, 0.5, 0.75, 1]).round(3)
array([[ 0.  ,  0.25,  0.5 ,  0.75,  1.  ]])
```

skfda.preprocessing.registration.normalize_warping

skfda.preprocessing.registration.normalize_warping(*warping*, *domain_range=None*)
[source]

Rescale a warping to normalize their domain.

Given a set of warpings $\gamma_i : [a, b] \to [a, b]$ it is used an affine traslation to change the domain of the transformation to other domain, $\tilde{\gamma}_i : [\tilde{a}, \tilde{b}] \to [\tilde{a}, \tilde{b}]$.

Parameters:
- **warping** (`FDatagrid`) – Set of warpings to rescale.
- **domain_range** (*tuple*, *optional*) – New domain range of the warping. By default it is used the same domain range.

Returns: FDataGrid with the warpings normalized.

Return type: (`FDataGrid`)

skfda.ml.classification.NearestNeighbors

class **skfda.ml.classification.NearestNeighbors**(*n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Unsupervised learner for implementing neighbor searches.

Parameters:

- **n_neighbors** (*int*, *optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **radius** (*float*, *optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}*, *optional*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int*, *optional (default = 30)*) – 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.
- **metric** (*string or callable*, *(default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict*, *optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None*, *optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean*, *optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
```

We will fit a Nearest Neighbors estimator

```
>>> from skfda.ml.classification import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
NearestNeighbors(algorithm='auto', leaf_size=30,...)
```

Now we can query the k-nearest neighbors.

```
>>> distances, index = neigh.kneighbors(fd[:2])
>>> index # Index of k-neighbors of samples 0 and 1
array([[ 0,  7,  6, 11,  2],...)
```

```
>>> distances.round(2) # Distances to k-neighbors
array([[ 0.  ,  0.28,  0.29,  0.29,  0.3 ],
       [ 0.  ,  0.27,  0.28,  0.29,  0.3 ]])
```

We can query the neighbors in a given radius too.

```
>>> distances, index = neigh.radius_neighbors(fd[:2])
>>> index[0]
array([ 0,  2,  6,  7, 11]...)
```

```
>>> distances[0].round(2) # Distances to neighbors of the sample 0
array([ 0.  ,  0.3 ,  0.29,  0.28,  0.29])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.KNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **`__init__`**(*n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the nearest neighbors searcher.

Methods

`__init__` ([n_neighbors, radius, algorithm, ...])	Initialize the nearest neighbors searcher.
`fit` (X[, y])	Fit the model using X as training data.
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbor
`radius_neighbors` ([X, radius, return_distance])	Finds the neighbors within a given radius of a
`radius_neighbors_graph` ([X, radius, mode])	Computes the (weighted) graph of Neighbors
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.KNeighborsClassifier

class **skfda.ml.classification.KNeighborsClassifier**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Classifier implementing the k-nearest neighbors vote.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable, optional (default = 'uniform')*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – 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.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a K-Nearest Neighbors classifier

```
>>> from skfda.ml.classification import KNeighborsClassifier
>>> neigh = KNeighborsClassifier()
>>> neigh.fit(fd, y)
KNeighborsClassifier(algorithm='auto', leaf_size=30,...)
```

We can predict the class of new samples

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

And the estimated probabilities.

```
>>> neigh.predict_proba(fd[0]) # Probabilities of sample 0
array([[ 1.,  0.]])
```

See also

`RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.KNeighborsClassifier*.

Warning

Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Initialize the classifier.

Methods

`__init__` ([n_neighbors, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y a
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbc
`predict` (X)	Predict the class labels for the provided data.
`predict_proba` (X)	Return probability estimates for the test data
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.RadiusNeighborsClassifier

class **skfda.ml.classification.RadiusNeighborsClassifier**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_label=None, n_jobs=1, sklearn_metric=False*) [source]

Classifier implementing a vote among neighbors within a given radius

Parameters:

- **radius** (*float*, *optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}*, *optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int*, *optional (default = 30)*) – Leaf size passed to BallTree. 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.
- **metric** (*string or callable*, *(default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **outlier_label** (*int*, *optional (default = None)*) – Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected.
- **metric_params** (*dict*, *optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None*, *optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean*, *optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes.

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a Radius Nearest Neighbors classifier.

```
>>> from skfda.ml.classification import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=.3)
>>> neigh.fit(fd, y)
RadiusNeighborsClassifier(algorithm='auto', leaf_size=30,...)
```

We can predict the class of new samples.

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

See also

`KNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_label=None, n_jobs=1, sklearn_metric=False*) [source]

Initialize the classifier.

Methods

`__init__`([radius, weights, algorithm, ...])	Initialize the classifier.
`fit`(X, y)	Fit the model using X as training data and y as
`get_params`([deep])	Get parameters for this estimator.

`predict`(X)	Predict the class labels for the provided data.
`radius_neighbors`([X, radius, return_distance])	Finds the neighbors within a given radius of a
`radius_neighbors_graph`([X, radius, mode])	Computes the (weighted) graph of Neighbors
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test d
`set_params`(**params)	Set the parameters of this estimator.

skfda.ml.regression.KNeighborsScalarRegressor

class **skfda.ml.regression.KNeighborsScalarRegressor**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Regression based on k-nearest neighbors with scalar response.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable, optional (default = 'uniform')*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – 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.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted.

```
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.datasets import make_multimodal_landmarks
>>> y = make_multimodal_landmarks(n_samples=30, std=.5, random_state=0)
>>> y = y.flatten()
>>> fd = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
```

We will fit a K-Nearest Neighbors regressor to regress a scalar response.

```
>>> from skfda.ml.regression import KNeighborsScalarRegressor
>>> neigh = KNeighborsScalarRegressor()
>>> neigh.fit(fd, y)
KNeighborsScalarRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples

```
>>> neigh.predict(fd[:4]).round(2) # Predict first 4 locations
array([ 0.79,  0.27,  0.71,  0.79])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn regressor *sklearn.neighbors.KNeighborsRegressor*.

Warning

Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__` ([n_neighbors, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y a
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbc
`predict` (X)	Predict the target for the provided data :para
`score` (X, y[, sample_weight])	Returns the coefficient of determination R^2
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.regression.RadiusNeighborsScalarRegressor

class **skfda.ml.regression.RadiusNeighborsScalarRegressor**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Scalar regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **radius** (*float*, *optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}*, *optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int*, *optional (default = 30)*) – Leaf size passed to BallTree. 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.
- **metric** (*string or callable*, *(default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict*, *optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None*, *optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean*, *optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted.

```
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.datasets import make_multimodal_landmarks
>>> y = make_multimodal_landmarks(n_samples=30, std=.5, random_state=0)
>>> y = y.flatten()
>>> fd = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
```

We will fit a K-Nearest Neighbors regressor to regress a scalar response.

```
>>> from skfda.ml.regression import RadiusNeighborsScalarRegressor
>>> neigh = RadiusNeighborsScalarRegressor(radius=.2)
>>> neigh.fit(fd, y)
RadiusNeighborsScalarRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples.

```
>>> neigh.predict(fd[:4]).round(2) # Predict first 4 locations
array([ 0.84,  0.27,  0.66,  0.79])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__`([radius, weights, algorithm, ...])	Initialize the classifier.
`fit`(X, y)	Fit the model using X as training data and y as t
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the target for the provided data :param
`radius_neighbors`([X, radius, return_distance])	Finds the neighbors within a given radius of a fc
`radius_neighbors_graph`([X, radius, mode])	Computes the (weighted) graph of Neighbors fc

`score`(X, y[, sample_weight])	Returns the coefficient of determination R^2 of
`set_params`(**params)	Set the parameters of this estimator.

skfda.ml.regression.RadiusNeighborsFunctionalRegresso

class **skfda.ml.regression.RadiusNeighborsFunctionalRegressor**(*radius=1.0, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_response=None, n_jobs=1, sklearn_metric=False*) [source]

Functional regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **radius** (*float*, *optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **regressor** (*callable*, *optional ((default =)* – `mean`)) Function to perform the local regression. By default used the mean. Can accept a user-defined function wich accepts a `FDataGrid` with the neighbors of a test sample, and if weights != 'uniform' an array of weights as second parameter.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}*, *optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int*, *optional (default = 30)*) – Leaf size passed to BallTree. 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.
- **metric** (*string or callable*, *(default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict*, *optional (default = None)*) – Additional keyword arguments for the metric function.
- **outlier_response** (`FDataGrid`, optional (default = None)) – Default response for test samples without neighbors.
- **n_jobs** (*int or None*, *optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean*, *optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted, and we will try to predict 5 X +1.

```
>>> from skfda.datasets import make_multimodal_samples
>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
>>> y_train = 5 * X_train + 1
>>> X_test = make_multimodal_samples(n_samples=5, std=.5, random_state=0)
```

We will fit a Radius Nearest Neighbors functional regressor.

```
>>> from skfda.ml.regression import RadiusNeighborsFunctionalRegressor
>>> neigh = RadiusNeighborsFunctionalRegressor(radius=.03)
>>> neigh.fit(X_train, y_train)
RadiusNeighborsFunctionalRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the response of new samples.

```
>>> neigh.predict(X_test)
FDataGrid(...)
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **`__init__`**(*radius=1.0, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_response=None, n_jobs=1, sklearn_metric=False*) [source]

Initialize the classifier.

Methods

`__init__`([radius, weights, regressor, ...])	Initialize the classifier.
`fit`(X, y)	Fit the model using X as training data.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict functional responses.
`radius_neighbors`([X, radius, return_distance])	Finds the neighbors within a given radius of a fdatag
`radius_neighbors_graph`([X, radius, mode])	Computes the (weighted) graph of Neighbors for poi

`score`(X, y)	TODO
`set_params`(**params)	Set the parameters of this estimator.

skfda.ml.regression.KNeighborsFunctionalRegressor

class **skfda.ml.regression.KNeighborsFunctionalRegressor**(*n_neighbors=5, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Functional regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **regressor** (*callable, optional ((default =*) – `mean`)) Function to perform the local regression. By default used the mean. Can accept a user-defined function wich accepts a `FDataGrid` with the neighbors of a test sample, and if weights != 'uniform' an array of weights as second parameter.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree. 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.
- **metric** (*string or callable, (default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted, and we will try to predict 5 X +1.

```
>>> from skfda.datasets import make_multimodal_samples
>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
>>> y_train = 5 * X_train + 1
>>> X_test = make_multimodal_samples(n_samples=5, std=.5, random_state=0)
```

We will fit a K-Nearest Neighbors functional regressor.

```
>>> from skfda.ml.regression import KNeighborsFunctionalRegressor
>>> neigh = KNeighborsFunctionalRegressor()
>>> neigh.fit(X_train, y_train)
KNeighborsFunctionalRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the response of new samples.

```
>>> neigh.predict(X_test)
FDataGrid(...)
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **`__init__`**(*n_neighbors=5, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*)
> [source]
>
> Initialize the classifier.

Methods

`__init__`([n_neighbors, weights, regressor, ...])	Initialize the classifier.
`fit`(X, y)	Fit the model using X as training data.

`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbo
`predict` (X)	Predict functional responses.
`score` (X, y)	TODO
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.NearestCentroids

class **skfda.ml.classification.NearestCentroids**(*metric=<function lp_distance>, mean=<function mean>*) [source]

Nearest centroid classifier for functional data.

Each class is represented by its centroid, with test samples classified to the class with the nearest centroid.

Parameters:

- **metric** (*callable, (default)* – `lp_distance`) The metric to use when calculating distance between test samples and centroids. See the documentation of the metrics module for a list of available metrics. Defaults used L2 distance.
- **mean** (callable, (default `mean`)) – The centroids for the samples corresponding to each class is the point from which the sum of the distances (according to the metric) of all samples that belong to that particular class are minimized. By default it is used the usual mean, which minimizes the sum of L2 distance. This parameter allows change the centroid constructor. The function must accept a `FData` with the samples of one class and return a `FData` object with only one sample representing the centroid.

centroids_

`FDataGrid` – FDatagrid containing the centroid of each class

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a Nearest centroids classifier

```
>>> from skfda.ml.classification import NearestCentroids
>>> neigh = NearestCentroids()
>>> neigh.fit(fd, y)
NearestCentroids(...)
```

We can predict the class of new samples

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`

__init__(*metric=<function lp_distance>*, *mean=<function mean>*) [source]

Initialize the classifier.

Methods

`__init__`([metric, mean])	Initialize the classifier.
`fit`(X, y)	Fit the model using X as training data and y as target values.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict the class labels for the provided data.
`score`(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

skfda.datasets.make_sinusoidal_process

skfda.datasets.make_sinusoidal_process(*n_samples: int = 15, n_features: int = 100, *, start: float = 0.0, stop: float = 1.0, period: float = 1.0, phase_mean: float = 0.0, phase_std: float = 0.6, amplitude_mean: float = 1.0, amplitude_std: float = 0.05, error_std: float = 0.2, random_state=None*)
[source]

Generate sinusoidal proccess.

Each sample $x_i(t)$ is generated as:

$$x_i(t) = \alpha_i \sin(\omega t + \phi_i) + \epsilon_i(t)$$

where $\omega = \frac{2\pi}{\text{period}}$. Amplitudes α_i and phases ϕ_i are normally distributed. $\epsilon_i(t)$ is a gaussian white noise process.

Parameters:

- **n_samples** – Total number of samples.
- **n_features** – Points per sample.
- **start** – Starting point of the samples.
- **stop** – Ending point of the samples.
- **period** – Period of the sine function.
- **phase_mean** – Mean of the phase.
- **phase_std** – Standard deviation of the phase.
- **amplitude_mean** – Mean of the amplitude.
- **amplitude_std** – Standard deviation of the amplitude.
- **error_std** – Standard deviation of the gaussian Noise.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.

skfda.datasets.make_multimodal_samples

skfda.datasets.make_multimodal_samples(*n_samples: int = 15, *, n_modes: int = 1, points_per_dim: int = 100, ndim_domain: int = 1, ndim_image: int = 1, start: float = -1, stop: float = 1.0, std: float = 0.05, mode_std: float = 0.02, noise: float = 0.0, modes_location=None, random_state=None*)
[source]

Generate multimodal samples.

Each sample $x_i(t)$ is proportional to a gaussian mixture, generated as the sum of multiple pdf of multivariate normal distributions with different means.

$$x_i(t) \propto \sum_{n=1}^{\text{n\_modes}} \exp\left(-\frac{1}{2\sigma}(t-\mu_n)^T \mathbb{1}(t-\mu_n)\right)$$

Where $\mu_n = \text{mode\_location}_n + \epsilon$ and ϵ is normally distributed, with mean $\mathbb{0}$ and standard deviation given by the parameter *std*.

Parameters:

- **n_samples** – Total number of samples.
- **n_modes** – Number of modes of each sample.
- **points_per_dim** – Points per sample. If the object is multidimensional indicates the number of points for each dimension in the domain. The sample will have :math: *text{points_per_dim}^text{ndim_domain}* points of discretization.
- **ndim_domain** – Number of dimensions of the domain.
- **ndim_image** – Number of dimensions of the image
- **start** – Starting point of the samples. In multidimensional objects the starting point of each axis.
- **stop** – Ending point of the samples. In multidimensional objects the ending point of each axis.
- **std** – Standard deviation of the variation of the modes location.
- **mode_std** – Standard deviation σ of each mode.
- **noise** – Standard deviation of Gaussian noise added to the data.
- **modes_location** – List of coordinates of each mode.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.

skfda.datasets.make_multimodal_landmarks

skfda.datasets.make_multimodal_landmarks(*n_samples: int = 15, *, n_modes: int = 1, ndim_domain: int = 1, ndim_image: int = 1, start: float = -1, stop: float = 1, std: float = 0.05, random_state=None*) [source]

Generate landmarks points.

Used by `make_multimodal_samples()` to generate the location of the landmarks.

Generates a matrix containing the landmarks or locations of the modes of the samples generates by `make_multimodal_samples()`.

If the same random state is used when generating the landmarks and multimodal samples, these will correspond to the position of the modes of the multimodal samples.

Parameters:

- **n_samples** – Total number of samples.
- **n_modes** – Number of modes of each sample.
- **ndim_domain** – Number of dimensions of the domain.
- **ndim_image** – Number of dimensions of the image
- **start** – Starting point of the samples. In multidimensional objects the starting point of the axis.
- **stop** – Ending point of the samples. In multidimensional objects the ending point of the axis.
- **std** – Standard deviation of the variation of the modes location.
- **random_state** – Random state.

Returns:

`np.ndarray` with the location of the modes, where the component (i,j,k) corresponds to the mode k of the image dimension j of the sample i.

skfda.datasets.make_random_warping

skfda.datasets.make_random_warping(*n_samples: int = 15, n_features: int = 100, *, start: float = 0.0, stop: float = 1.0, sigma: float = 1.0, shape_parameter: float = 50, n_random: int = 4, random_state=None*) [source]

Generate random warping functions.

Let $v(t)$ be a randomly generated function defined in $[0, 1]$

$$v(t) = \sum_{j=0}^{N} a_j \sin(\frac{2\pi j}{K} t) + b_j \cos(\frac{2\pi j}{K} t)$$

where $a_j, b_j \sim N(0, \sigma)$.

The random warping it is constructed making an exponential map to Γ.

$$\gamma(t) = \int_0^t \left(\frac{\sin(\|v\|)}{\|v\|} v(s) + \cos(\|v\|) \right)^2 ds$$

An affine traslation it is used to define the warping in $[a, b]$.

The smoothing and shape of the warpings can be controlling changing N, σ and $K = 1 + \text{shape\_parameter}$.

Parameters:

- **n_samples** – Total number of samples. Defaults 15.
- **n_features** – The total number of trajectories. Defaults 100.
- **start** – Starting point of the samples. Defaults 1.
- **stop** – Ending point of the samples. Defaults 0.
- **sigma** – Parameter to control the variance of the samples. Defaults 1.
- **shape_parameter** – Parameter to control the shape of the warpings. Should be a positive value. When the shape parameter goes to infinity the warpings generated are γ_{id}. Defaults to 50.
- **n_random** – Number of random sines and cosines to be sum to construct the warpings.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.