### Contents:

1 **API Reference** ................................................. 3
   1.1 Representation of functional Data .................................... 3
   1.2 Preprocessing ................................................................... 63
   1.3 Exploratory analysis .................................................. 115
   1.4 Datasets ....................................................................... 141
   1.5 Miscellaneous .................................................................. 150
   1.6 Machine Learning ....................................................... 157

2 **Examples** ................................................................ 201
   2.1 Discretized function representation .................................... 201
   2.2 Function composition .................................................. 203
   2.3 Surface Boxplot ........................................................ 207
   2.4 Shift Registration ....................................................... 211
   2.5 Neighbors Functional Regression ...................................... 216
   2.6 Elastic registration ...................................................... 221
   2.7 Landmark registration ................................................... 226
   2.8 Exploring data .......................................................... 231
   2.9 Representation of functional data ..................................... 235
   2.10 Boxplot ....................................................................... 250
   2.11 Neighbors Scalar Regression .......................................... 255
   2.12 Landmark shift ........................................................ 262
   2.13 K-nearest neighbors classification ................................... 267
   2.14 Magnitude-Shape Plot ................................................ 272
   2.15 Kernel Smoothing ...................................................... 278
   2.16 Interpolation ............................................................ 287
   2.17 Magnitude-Shape Plot synthetic example ........................... 297
   2.18 Radius neighbors classification ....................................... 302
   2.19 Clustering ............................................................... 308
   2.20 Pairwise alignment ...................................................... 317
   2.21 Extrapolation ........................................................... 326

3 **Installation** .......................................................... 337

4 **Contributions** ....................................................... 339

5 **License** .............................................................. 341
Bibliography 343
Index 347
This package offers classes, methods and functions to give support to Functional Data Analysis in Python. Includes a wide range of utils to work with functional data, and its representation, exploratory analysis, or preprocessing, among other tasks such as inference, classification, regression or clustering of functional data.

In the project page hosted by Github you can find more information related to the development of the package.
1.1 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.

1.1.1 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.

```python
skfda.representation.grid.FDataGrid(data_matrix)
```

Represent discretised functional data.

**FDataGrid**

A class for representing functional data as a set of curves discretised in a grid of points.

**data_matrix**

a matrix where each entry of the first axis contains the values of a functional datum evaluated at the points of discretisation.

**Type** numpy.ndarray
sample_points
2 dimension matrix where each row contains the points of discretisation for each axis of data_matrix.

Type numpy.ndarray
domain_range
2 dimension matrix where each row contains the bounds of the interval in which the functional data is considered to exist for each one of the axes.

Type numpy.ndarray
dataset_label
name of the dataset.

Type str
axes_labels
list containing the labels of the different axis.

Type list
extrapolation
defines the default type of extrapolation. By default None, which does not apply any type of extrapolation. See Extrapolation for detailed information of the types of extrapolation.

Type str or Extrapolation
interpolator
Defines the type of interpolation applied in evaluate.

Type GridInterpolator
keepdims

Type bool

Examples

Representation of a functional data object with 2 samples representing a function \( f : \mathbb{R} \rightarrow \mathbb{R} \).

```python
>>> data_matrix = [[1, 2], [2, 3]]
>>> sample_points = [2, 4]
>>> FDataGrid(data_matrix, sample_points)
FDataGrid(array([[1],
                 [2]],
                 [[2],
                 [3]]),
sample_points=[array([2, 4])],
domain_range=array([[2, 4]],
                   ...

The number of columns of data_matrix have to be the length of sample_points.

>>> FDataGrid(np.array([1,2,4,5,8]), range(6))
Traceback (most recent call last):
  ....
ValueError: Incorrect dimension in data_matrix and sample_points...
FDataGrid support higher dimensional data both in the domain and image. Representation of a functional data object with 2 samples representing a function $f : \mathbb{R} \rightarrow \mathbb{R}^2$.

```python
>>> data_matrix = [[[1, 0.3], [2, 0.4]], [[2, 0.5], [3, 0.6]]
>>> sample_points = [2, 4]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> fd.dim_domain, fd.dim_codomain
(1, 2)
```

Representation of a functional data object with 2 samples representing a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$.

```python
>>> data_matrix = [[[1, 0.3], [2, 0.4]], [[2, 0.5], [3, 0.6]]
>>> sample_points = [[2, 4], [3, 6]]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> fd.dim_domain, fd.dim_codomain
(2, 1)
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>(data_matrix[, sample_points, …]) Construct a FDataGrid object.</td>
</tr>
<tr>
<td><code>argsort</code></td>
<td>(ascending, kind, *args, **kwargs) Return the indices that would sort this array.</td>
</tr>
<tr>
<td><code>astype</code></td>
<td>(dtype[, copy]) Cast to a NumPy array with 'dtype'.</td>
</tr>
<tr>
<td><code>compose</code></td>
<td>(fd, *[, eval_points]) Composition of functions.</td>
</tr>
<tr>
<td><code>concatenate</code></td>
<td>(*others[, as_coordinates]) Join samples from a similar FDataGrid object.</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>([*, deep, data_matrix, sample_points, …]) Returns a copy of the FDataGrid.</td>
</tr>
<tr>
<td><code>cov</code></td>
<td>Compute the covariance.</td>
</tr>
<tr>
<td><code>derivative</code></td>
<td>([order]) Differentiate a FDataGrid object.</td>
</tr>
<tr>
<td><code>dropna</code></td>
<td>() Return ExtensionArray without NA values.</td>
</tr>
<tr>
<td><code>evaluate</code></td>
<td>(eval_points, *[[], derivative, …]) Evaluate the object or its derivatives at a list of values or a grid.</td>
</tr>
<tr>
<td><code>factorize</code></td>
<td>(na_sentinel) Encode the extension array as an enumerated type.</td>
</tr>
<tr>
<td><code>fillna</code></td>
<td>([value, method, limit]) Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>gmean</code></td>
<td>() Compute the geometric mean of all samples in the FDataGrid object.</td>
</tr>
<tr>
<td><code>isna</code></td>
<td>() A 1-D array indicating if each value is missing.</td>
</tr>
<tr>
<td><code>mean</code></td>
<td>(weights) Compute the mean of all the samples.</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>(*args, **kwargs) Plot the FDatGrid object.</td>
</tr>
<tr>
<td><code>ravel</code></td>
<td>([order]) Return a flattened view on this array.</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>(repeats[, axis]) Repeat elements of a ExtensionArray.</td>
</tr>
<tr>
<td><code>round</code></td>
<td>(decimals) Evenly round to the given number of decimals.</td>
</tr>
<tr>
<td><code>scatter</code></td>
<td>(*args, **kwargs) Scatter plot of the FDatGrid object.</td>
</tr>
<tr>
<td><code>searchsorted</code></td>
<td>(value[, side, sorter]) Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>shift</code></td>
<td>(shifts, *[[], restrict_domain, …]) Perform a shift of the curves.</td>
</tr>
<tr>
<td><code>take</code></td>
<td>(indices[, allow_fill, fill_value, axis]) Take elements from an array.</td>
</tr>
<tr>
<td><code>to_basis</code></td>
<td>(basis, **kwargs) Return the basis representation of the object.</td>
</tr>
<tr>
<td><code>to_grid</code></td>
<td>([sample_points]) Return the discrete representation of the object.</td>
</tr>
<tr>
<td><code>to_numpy</code></td>
<td>() Returns a numpay array with the objects</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>() Compute the ExtensionArray of unique values.</td>
</tr>
</tbody>
</table>

1.1. Representation of functional Data
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var()</code></td>
<td>Compute the variance of a set of samples in a FDataGrid object.</td>
</tr>
<tr>
<td><code>view([dtype])</code></td>
<td>Return a view on the array.</td>
</tr>
</tbody>
</table>

**__init__**

```python
__init__(data_matrix, sample_points=None, domain_range=None, dataset_label=None, axes_labels=None, extrapolation=None, interpolator=None, keepdims=False)
```

Construct a FDataGrid object.

**Parameters**

- **data_matrix (array_like)** – a matrix where each row contains the values of a functional datum evaluated at the points of discretisation.
- **sample_points (array_like, optional)** – an array containing the points of discretisation where values have been recorded or a list of lists with each of the list containing the points of discretisation for each axis.
- **domain_range (tuple or list of tuples, optional)** – contains the edges of the interval in which the functional data is considered to exist (if the argument has 2 dimensions each row is interpreted as the limits of one of the dimension of the domain).
- **dataset_label (str, optional)** – name of the dataset.
- **axes_labels (list, optional)** – list containing the labels of the different axes. The length of the list must be equal to the sum of the number of dimensions of the domain plus the number of dimensions of the image.

**Attributes**

- **axes_labels**
  - Return the list of axes labels
- **coordinates**
  - Returns an object to access to the image coordinates.
- **dim_codomain**
  - Return number of dimensions of the image.
- **dim_domain**
  - Return number of dimensions of the domain.
- **domain_range**
  - Return the edges of the interval in which the functional data is considered to exist by the sample points.
- **dtype**
  - The dtype for this extension array, FDataGrid-DType
- **extrapolation**
  - Return default type of extrapolation.
- **extrapolator_evaluator**
  - Return the evaluator constructed by the extrapolator.
- **interpolator**
  - Defines the type of interpolation applied in evaluate.
- **n_samples**
  - Return number of rows of the data_matrix.
- **nbytes**
  - The number of bytes needed to store this object in memory.
- **ncol**
  - Return number of columns of the data_matrix.
- **ndim**
  - Return number of dimensions of the functional data.

Continued on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sample_range</code></td>
<td>Return the edges of the interval in which the functional data is considered to exist by the sample points.</td>
</tr>
<tr>
<td><code>shape</code></td>
<td>Return a tuple of the array dimensions.</td>
</tr>
</tbody>
</table>

`argsort(ascending: bool = True, kind: str = 'quicksort', *args, **kwargs) → numpy.ndarray`

Return the indices that would sort this array.

**Parameters**

- `ascending (bool, default True)` – Whether the indices should result in an ascending or descending sort.
- `kind ({'quicksort', 'mergesort', 'heapsort'}, optional)` – Sorting algorithm.
- `**kwargs (*args,)` – passed through to `numpy.argsort()`.

**Returns** Array of indices that sort `self`. If NaN values are contained, NaN values are placed at the end.

**Return type** `ndarray`

See also:
- `numpy.argsort()` Sorting implementation used internally.

`astype(dtype, copy=True)`

Cast to a NumPy array with ‘dtype’.

**Parameters**

- `dtype (str or dtype)` – Typecode or data-type to which the array is cast.
- `copy (bool, default True)` – Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

**Returns** `array` – NumPy ndarray with ‘dtype’ for its dtype.

**Return type** `ndarray`

`axes_labels`

Return the list of axes labels

`compose(fd, *, eval_points=None)`

Composition of functions.

Performs the composition of functions.

**Parameters**

- `fd (FData)` – FData object to make the composition. Should have the same number of samples and image dimension equal to 1.
- `eval_points (array_like)` – Points to perform the evaluation.

`concatenate(*others, as_coordinates=False)`

Join samples from a similar FDataGrid object.

Joins samples from another FDataGrid object if it has the same dimensions and sampling points.

**Parameters**

- `others (FDataGrid)` – Objects to be concatenated.
• **as_coordinates** *(boolean, optional)* – If False concatenates as new samples, else, concatenates the other functions as new components of the image. Defaults to false.

**Returns** FDataGrid object with the samples from the original objects.

**Return type** *FDataGrid*

**Examples**

```python
>>> fd = FDataGrid([1, 2, 4, 5, 8], range(5))
>>> fd_2 = FDataGrid([3, 4, 7, 9, 2], range(5))
>>> fd.concatenate(fd_2)
FDataGrid(
    array([[1],
           [2],
           [4],
           [5],
           [8]],
          <BLANKLINE>
           [[3],
           [4],
           [7],
           [9],
           [2]]),
    sample_points=[array([0, 1, 2, 3, 4])],
    domain_range=[array([0, 4])],
    ...
)
```

**coordinates**

Returns an object to access to the image coordinates.

If the functional object contains multivariate samples \( f : \mathbb{R}^n \rightarrow \mathbb{R}^d \), this class allows iterate and get coordinates of the vector \( f = (f_0, ..., f_{d-1}) \).

**Examples**

We will construct a dataset of curves in \( \mathbb{R}^3 \)

```python
>>> from skfda.datasets import make_multimodal_samples
>>> fd = make_multimodal_samples(dim_codomain=3, random_state=0)
>>> fd.dim_codomain
3
```

The functions of this dataset are vectorial functions \( f(t) = (f_0(t), f_1(t), f_2(t)) \). We can obtain a specific component of the vector, for example, the first one.

```python
>>> fd_0 = fd.coordinates[0]
>>> fd_0
FDataGrid(...)
```

The object returned has image dimension equal to 1
Or we can get multiple components, it can be accessed as a 1-d numpy array of coordinates, for example, \((f_0(t), f_1(t))\).

```python
>>> fd_01 = fd.coordinates[0:2]
>>> fd_01.dim_codomain
2
```

We can use this method to iterate through all the coordinates.

```python
>>> for fd_i in fd.coordinates:
...     ... fd_i.dim_codomain
1
1
1
```

This object can be used to split a FDataGrid in a list with their components.

```python
>>> fd_list = list(fd.coordinates)
>>> len(fd_list)
3
```

**copy\(\ast\), deep=False, data_matrix=None, sample_points=None, domain_range=None, dataset_label=None, axes_labels=None, extrapolation=None, interpolator=None, keepdims=None\)**  
Returns a copy of the FDataGrid.

If an argument is provided the corresponding attribute in the new copy is updated.

**cov()**  
Compute the covariance.

Calculates the covariance matrix representing the covariance of the functional samples at the observation points.

- **Returns** Matrix of covariances.
- **Return type** numpy.darray

**derivative(order=1)**  
Differentiate a FDataGrid object.

It is calculated using lagged differences. If we call \(D\) the data_matrix, \(D^1\) the derivative of order 1 and \(T\) the vector containing the points of discretisation; \(D^1\) is calculated as it follows:

\[
D^1_{ij} = \begin{cases} 
\frac{D_i - D_2}{T_j - T_2} & \text{if } j = 1 \\
\frac{D_{i(m-1)} - D_{im}}{T_{m-1} - T_m} & \text{if } j = m \\
\frac{D_{i(j-1)} - D_{i(j+1)}}{T_{j-1} - T_{j+1}} & \text{if } 1 < j < m
\end{cases}
\]

Where \(m\) is the number of columns of the matrix \(D\).

Order \(> 1\) derivatives are calculated by using derivative recursively.

- **Parameters** order (int, optional) – Order of the derivative. Defaults to one.
**Examples**

First order derivative

```python
>>> fdata = FDataGrid([1,2,4,5,8], range(5))
>>> fdata.derivative()
FDataGrid(
    array([[ 1. ],
            [ 1.5],
            [ 1.5],
            [ 2. ],
            [ 3. ]]),
    sample_points=array([0, 1, 2, 3, 4]),
    domain_range=array([[0, 4]]),
    ...)
```

Second order derivative

```python
>>> fdata = FDataGrid([1,2,4,5,8], range(5))
>>> fdata.derivative(2)
FDataGrid(
    array([[ 0.5 ],
            [ 0.25],
            [ 0.25],
            [ 0.75],
            [ 1. ]]),
    sample_points=array([0, 1, 2, 3, 4]),
    domain_range=array([[0, 4]]),
    ...)
```

dim_codomain

Return number of dimensions of the image.

**Returns** Number of dimensions of the image.

**Return type** int

dim_domain

Return number of dimensions of the domain.

**Returns** Number of dimensions of the domain.

**Return type** int

domain_range

Return the edges of the interval in which the functional data is considered to exist by the sample points.

Do not have to be equal to the sample_range.

dropna()

Return ExtensionArray without NA values.

**Returns** valid

**Return type** ExtensionArray

dtype

The dtype for this extension array, FDataGridDType
evaluate(eval_points, *, derivative=0, extrapolation=None, grid=False, aligned_evaluation=True, keepdims=None)

Evaluate the object or its derivatives at a list of values or a grid.

Parameters

- **eval_points (array_like)** – List of points where the functions are evaluated. If a matrix of shape nsample x eval_points is given each sample is evaluated at the values in the corresponding row in eval_points.
- **derivative (int, optional)** – Order of the derivative. Defaults to 0.
- **extrapolation (str or Extrapolation, optional)** – Controls the extrapolation mode for elements outside the domain range. By default it is used the mode defined during the instance of the object.
- **grid (bool, optional)** – Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays. If true the eval_points should be a list of size dim_domain with the corresponding times for each axis. The return matrix has shape n_samples x len(t1) x len(t2) x ... x len(t_dim_domain) x dim_codomain. If the domain dimension is 1 the parameter has no effect. Defaults to False.
- **keepdims (bool, optional)** – If the image dimension is equal to 1 and keepdims is True the return matrix has shape n_samples x eval_points x 1 else n_samples x eval_points. By default is used the value given during the instance of the object.

Returns

Matrix whose rows are the values of the each function at the values specified in eval_points.

Return type (np.darray)

extrapolation

Return default type of extrapolation.

extrapolator_evaluator

Return the evaluator constructed by the extrapolator.

factorize(na_sentinel: int = -1) → Tuple[numpy.ndarray, pandas.core.dtypes.generic.ABCExtensionArray]

Encode the extension array as an enumerated type.

Parameters **na_sentinel (int, default -1)** – Value to use in the codes array to indicate missing values.

Returns

- **codes (ndarray)** – An integer NumPy array that’s an indexer into the original ExtensionArray.
- **uniques (ExtensionArray)** – An ExtensionArray containing the unique values of self.

Note: uniques will not contain an entry for the NA value of the ExtensionArray if there are any missing values present in self.

See also:

factorize() Top-level factorize method that dispatches here.
Notes

pandas.factorize() offers a sort keyword as well.

fillna(value=None, method=None, limit=None)
Fill NA/NaN values using the specified method.

Parameters

- **value (scalar, array-like)** – If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like ‘value’ can be given. It’s expected that the array-like have the same length as ‘self’.

- **method (\{'backfill', 'bfill', 'pad', 'ffill', None\}, default None)** – Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap.

- **limit (int, default None)** – If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns With NA/NaN filled.

Return type ExtensionArray

gmean()
Compute the geometric mean of all samples in the FDataGrid object.

Returns A FDataGrid object with just one sample representing the geometric mean of all the samples in the original FDataGrid object.

Return type FDataGrid

interpolator
Defines the type of interpolation applied in evaluate.

isna()
A 1-D array indicating if each value is missing.

Returns Array full of False values.

Return type na_values (np.ndarray)

mean(weights=None)
Compute the mean of all the samples.

Parameters weights (array-like, optional) – List of weights.

Returns A FDataGrid object with just one sample representing the mean of all the samples in the original object.

Return type FDataGrid

n_samples
Return number of rows of the data_matrix. Also the number of samples.

Returns

Number of samples of the FDataGrid object. Also the number of rows of the data_matrix.

Return type int
nbytes
The number of bytes needed to store this object in memory.

ncol
Return number of columns of the data_matrix.
Also the number of points of discretisation.

Returns Number of columns of the data_matrix.
Return type int

ndim
Return number of dimensions of the functional data. It is always 1, as each observation is considered a “scalar” object.

Returns Number of dimensions of the functional data.
Return type int

plot(*args, **kwargs)
Plot the FDatGrid object.

Parameters

- chart (figure object, axe or list of axes, optional) – figure over with the graphs are plotted or axis over where the graphs are plotted. If None and ax is also None, the figure is initialized.
- derivative (int or tuple, optional) – Order of derivative to be plotted. In case of surfaces a tuple with the order of derivation in each direction can be passed. See evaluate() to obtain more information. Defaults 0.
- fig (figure object, optional) – figure over with the graphs are plotted in case ax is not specified. If None and ax is also None, the figure is initialized.
- ax (list of axis objects, optional) – axis over where the graphs are plotted. If None, see param fig.
- n_rows (int, optional) – designates the number of rows of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- n_cols (int, optional) – designates the number of columns of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- n_points (int or tuple, optional) – Number of points to evaluate in the plot. In case of surfaces a tuple of length 2 can be passed with the number of points to plot in each axis, otherwise the same number of points will be used in the two axes. By default in unidimensional plots will be used 501 points; in surfaces will be used 30 points per axis, which makes a grid with 900 points.
- domain_range (tuple or list of tuples, optional) – Range where the function will be plotted. In objects with unidimensional domain the domain range should be a tuple with the bounds of the interval; in the case of surfaces a list with 2 tuples with the ranges for each dimension. Default uses the domain range of the functional object.
- group (list of int) – contains integers from [0 to number of labels) indicating to which group each sample belongs to. Then, the samples with the same label are plotted in the same color. If None, the default value, each sample is plotted in the color assigned by matplotlib.pyplot.rcParams['axes.prop_cycle'].

1.1. Representation of functional Data 13
- **group_colors** *(list of colors)* – colors in which groups are represented, there must be one for each group. If None, each group is shown with distinct colors in the “Greys” colormap.

- **group_names** *(list of str)* – name of each of the groups which appear in a legend, there must be one for each one. Defaults to None and the legend is not shown.

- ****kwargs** – if dim_domain is 1, keyword arguments to be passed to the matplotlib.pyplot.plot function; if dim_domain is 2, keyword arguments to be passed to the matplotlib.pyplot.plot_surface function.

**Returns** figure object in which the graphs are plotted.

**Return type** fig (figure object)

```python
ravel(order='C') → pandas.core.dtypes.generic.ABCExtensionArray
```

Return a flattened view on this array.

**Parameters**

- order *(None, 'C', 'F', 'A', 'K'), default 'C'* –

**Returns**

**Return type** ExtensionArray

**Notes**

- Because ExtensionArrays are 1D-only, this is a no-op.

- The “order” argument is ignored, is for compatibility with NumPy.

```python
repeat(repeats, axis=None)
```

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

**Parameters**

- repeats *(int or array of ints)* – The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.

- axis *(None)* – Must be None. Has no effect but is accepted for compatibility with numpy.

**Returns** repeated_array – Newly created ExtensionArray with repeated elements.

**Return type** ExtensionArray

**See also:**

--Series.repeat() Equivalent function for Series.

-Index.repeat() Equivalent function for Index.

-numpy.repeat() Similar method for numpy.ndarray.

-ExtensionArray.take() Take arbitrary positions.
Examples

```python
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]
```

```python
>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
```

```python
>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

```python
round(decimals=0)
```
Evenly round to the given number of decimals.

Parameters: `decimals` (int, optional) – Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point. Defaults to 0.

Returns: obj: FDataGrid: Returns a FDataGrid object where all elements in its data_matrix are rounded. The real and imaginary parts of complex numbers are rounded separately.

```python
sample_range
```
Return the edges of the interval in which the functional data is considered to exist by the sample points.

Do not have to be equal to the domain_range.

```python
scatter(*args, **kwargs)
```
Scatter plot of the FDatGrid object.

Parameters:

- `fig` (figure object, optional) – figure over with the graphs are plotted in case ax is not specified. If None and ax is also None, the figure is initialized.

- `axes` (list of axis objects, optional) – axis over where the graphs are plotted. If None, see param fig.

- `n_rows` (int, optional) – designates the number of rows of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.

- `n_cols` (int, optional) – designates the number of columns of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.

- `kwargs` – keyword arguments to be passed to the matplotlib.pyplot.scatter function;

Returns: figure object in which the graphs are plotted.

Return type: fig (figure)

```python
searchsorted(value, side='left', sorter=None)
```
Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array `self` (a) such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

Assuming that `self` is sorted:
Parameters

- **value** (*array_like*) – Values to insert into `self`.
- **side** ({'left', 'right'}, optional) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).
- **sorter** (*1-D array_like, optional*) – Optional array of integer indices that sort array `a` into ascending order. They are typically the result of argsort.

Returns

Array of insertion points with the same shape as `value`.

Return type

array of ints

See also:

numpy.searchsorted() Similar method from NumPy.

**shape**

Return a tuple of the array dimensions.

**shift**(*shifts, *restrict_domain=False, extrapolation=None, eval_points=None*)

Perform a shift of the curves.

Parameters

- **shifts** (*array_like or numeric*) – List with the shifts corresponding for each sample or numeric with the shift to apply to all samples.
- **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 to obtain the discrete representation of the object to operate. If an empty list the current sample_points are used to unificate the domain of the shifted data.

Returns

`FDataGrid` with the shifted data.

**take**(*indices, allow_fill=False, fill_value=None, axis=0*)

Take elements from an array.

Parameters

- **indices** (*sequence of integers*) – Indices to be taken.
- **allow_fill** (*bool, default False*) – How to handle negative values in `indices`.
  - False: negative values in `indices` indicate positional indices from the right (the default). This is similar to numpy.take().
  - True: negative values in `indices` indicate missing values. These values are set to `fill_value`. Any other negative values raise a ValueError.
• **fill_value** *(any, optional)* – Fill value to use for NA-indices when `allow_fill` is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used. For many ExtensionArrays, there will be two representations of `fill_value`: a user-facing “boxed” scalar, and a low-level physical NA value. `fill_value` should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

**Returns** FData

**Raises**

- `IndexError` – When the indices are out of bounds for the array.
- `ValueError` – When `indices` contains negative values other than `-1` and `allow_fill` is True.

**Notes**

`ExtensionArray.take` is called by `Series.__getitem__`, `.loc`, `.iloc`, when `indices` is a sequence of values. Additionally, it’s called by `Series.reindex()`, or any other method that causes realignment, with a `fill_value`.

**See also:**

`numpy.take` `pandas.api.extensions.take`

to_basis(*basis, **kwargs*)

Return the basis representation of the object.

**Parameters**

- `basis` *(Basis)* – basis object in which the functional data are going to be represented.
- `**kwargs` – keyword arguments to be passed to `FDataBasis.from_data()`.

**Returns** Basis representation of the functional data object.

**Return type** `FDataBasis`

**Examples**

```python
def f_data
>>> import numpy as np
>>> import skfda
>>> t = np.linspace(0, 1, 5)
>>> x = np.sin(2 * np.pi * t) + np.cos(2 * np.pi * t)
>>> x
array([ 1., 1., -1., -1., 1.])
```

```python
def f_data
>>>basis = skfda.representation.basis.Fourier(n_basis=3)
>>>fd_b = fd.to_basis(basis)
>>>fd_b.coefficients.round(2)
array([[ 0. , 0.71, 0.71]])
```
Parameters

- **sample_points** *(array_like, optional)* – 2 dimension matrix where,
  - **row contains the points of discretisation for each axis of (each)* –
  - **data_matrix.** –

Returns Discrete representation of the functional data object.

Return type *FDataGrid*

**to_numpy()**

Returns a numpy array with the objects

**unique()**

Compute the ExtensionArray of unique values.

Returns *uniques*

Return type *ExtensionArray*

**var()**

Compute the variance of a set of samples in a FDataGrid object.

Returns A FDataGrid object with just one sample representing the variance of all the
  samples in the original FDataGrid object.

Return type *FDataGrid*

**view(** **dtype** *=None*)*

Return a view on the array.

Parameters **dtype** *(str, np.dtype, or ExtensionDtype, optional) – Default None.*

Returns A view of the ExtensionArray.

Return type *ExtensionArray*

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

```python
skfda.representation.interpolation.
SplineInterpolator([...])
```

**SplineInterpolator**

Class *skfda.representation.interpolation.SplineInterpolator*(interpolation_order=1,

smoothness_parameter=0.0,

monotone=False)

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 detailed explanation.

**interpolator_order**

Order of the interpolation, 1 for linear interpolation, 2 for quadratic, 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

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([interpolation_order,...])</td>
<td>Constructor of the SplineInterpolator.</td>
</tr>
<tr>
<td>evaluator(fdatagrid)</td>
<td>Construct a SplineInterpolatorEvaluator used in the evaluation.</td>
</tr>
</tbody>
</table>

```
__init__(interpolation_order=1, smoothness_parameter=0.0, monotone=False)
```

Constructor of the SplineInterpolator.

**Parameters**

- **interpolation_order (int, optional)** – Order of the interpolation, 1 for linear interpolation, 2 for quadratic, 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 linear 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.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpolation_order</td>
<td>Returns the interpolation order</td>
</tr>
<tr>
<td>monotone</td>
<td>Returns flag to perform monotone interpolation</td>
</tr>
<tr>
<td>smoothness_parameter</td>
<td>Returns the smoothness parameter</td>
</tr>
</tbody>
</table>

```
evaluator(fdatagrid)
```

Construct a SplineInterpolatorEvaluator used in the evaluation.

**Parameters**

- **fdatagrid (FDataGrid)** – Functional object where the evaluator will be used.

**Returns**

Evaluator of the fdatagrid.

**Return type** (SplineInterpolatorEvaluator)

**interpolation_order**
Returns the interpolation order
monotone
    Returns flag to perform monotone interpolation

smoothness_parameter
    Returns the smoothness parameter

Examples using skfda.representation.interpolation.SplineInterpolator

- Function composition
- Elastic registration
- Representation of functional data
- Interpolation

1.1.2 Basis representation

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

```python
skfda.representation.basis.FDataBasis(basis, ...)
```

Basis representation of functional data.

**FDataBasis**

```python
class skfda.representation.basis.FDataBasis(basis, coefficients, *, dataset_label=None, axes_labels=None, extrapolation=None, keep_dims=False)
```

Basis representation of functional data.

Class representation for functional data in the form of a set of basis functions multiplied by a set of coefficients.

\[ f(x) = \sum_{k=1}^{n} c_k \phi_k \]

Where \( n \) is the number of basis functions, \( c = (c_1, c_2, ..., c_K) \) the vector of coefficients and \( \phi = (\phi_1, \phi_2, ..., \phi_K) \) the basis function system.

**basis**

- **Type**: Basis

**coefficients**

- **List or matrix of coefficients. Has to have the same length or number of columns as the number of basis function in the basis. If a matrix, each row contains the coefficients that multiplied by the basis functions produce each functional datum.**

- **Type**: numpy.ndarray

Examples
```python
>>> basis = Monomial(n_basis=4)
>>> coefficients = [1, 1, 3, .5]
>>> FDataBasis(basis, coefficients)
FDataBasis(basis=Monomial(domain_range=[array([0, 1])], n_basis=4), coefficients=[[ 1.  1.  3.  0.5]], ...)
```

Methods

- **__init__**(basis, coefficients, *[, ...]) Construct a FDataBasis object.
- **argsort**(ascending, kind, *args, **kwargs) 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**(others, as_coordinates) Join samples from a similar FDataBasis object.
- **copy**(*, basis, coefficients, ...) FDataBasis copy
- **cov**(eval_points) Compute the covariance of the functional data object.
- **derivative**(order) Differentiate a FDataBasis object.
- **dropna**( ) Return ExtensionArray without NA values.
- **evaluate**(eval_points, *, derivative, ...) Evaluate the object or its derivatives at a list of values or a grid.
- **factorize**(na_sentinel) Encode the extension array as an enumerated type.
- **fillna**(value, method, limit) Fill NA/Nan values using the specified method.
- **from_data**(data_matrix, sample_points, basis) Transform raw data to a smooth functional form.
- **gmean**(eval_points) Compute the geometric mean of the functional data object.
- **inner_product**(other[, lfd_self, lfd_other, ...]) Return an inner product matrix given a FDataBasis object.
- **isna**( ) A 1-D array indicating if each value is missing.
- **mean**(weights) Compute the mean of all the samples in a FDataBasis object.
- **plot**(*, args, **kwargs) Plot the FDatGrid object.
- **ravel**(order) Return a flattened view on this array.
- **repeat**(repeats[, axis]) Repeat elements of a ExtensionArray.
- **searchsorted**(value[, side, sorter]) Find indices where elements should be inserted to maintain order.
- **shift**(shifts, *, restrict_domain, ...) Perform a shift of the curves.
- **take**(indices[, allow_fill, fill_value, axis]) Take elements from an array.
- **times**(other) "Provides a numerical approximation of the multiplication between
- **to_basis**(basis[, eval_points]) Return the basis representation of the object.
- **to_grid**(eval_points) Return the discrete representation of the object.
- **tolist**( ) Splits FDataBasis samples into a list
- **to_numpy**( ) Returns a numpy array with the objects
- **unique**( ) Compute the ExtensionArray of unique values.
- **var**(eval_points) Compute the variance of the functional data object.

Continued on next page
Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>view([dtype])</td>
<td>Return a view on the array.</td>
</tr>
</tbody>
</table>

```
__init__(basis, coefficients, *, dataset_label=None, axes_labels=None, extrapolation=None, keepdims=False)
```

Construct a FDataBasis object.

**Parameters**

- `basis (Basis)` – Basis function system.
- `coefficients (array_like)` – List or matrix of coefficients. Has to have the same length or number of columns as the number of basis function in the basis.

**Attributes**

- `axes_labels` – Return the list of axes labels.
- `coordinates` – Return a component of the FDataBasis.
- `dim_codomain` – Return number of dimensions of the image.
- `dim_domain` – Return number of dimensions of the domain.
- `domain_range` – Definition range.
- `dtype` – The dtype for this extension array, FDataGridDType
- `extrapolation` – Return default type of extrapolation.
- `extrapolator_evaluator` – Return the evaluator constructed by the extrapolator.
- `n_basis` – Return number of basis.
- `n_samples` – Return number of samples.
- `nbytes` – The number of bytes needed to store this object in memory.
- `ndim` – Return number of dimensions of the functional data.
- `shape` – Return a tuple of the array dimensions.

```
argsort(ascending: bool = True, kind: str = 'quicksort', *args, **kwargs) -> numpy.ndarray
```

Return the indices that would sort this array.

**Parameters**

- `ascending (bool, default True)` – Whether the indices should result in an ascending or descending sort.
- `**kwargs (*args,)` – passed through to `numpy.argsort()`.

**Returns** Array of indices that sort self. If NaN values are contained, NaN values are placed at the end.

**See also:**

`numpy.argsort()` Sorting implementation used internally.
astype(dtype, copy=True)
   Cast to a NumPy array with ‘dtype’.

Parameters

- **dtype** *(str or dtype)* – Typecode or data-type to which the array is cast.
- **copy** *(bool, default True)* – Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

Returns array – NumPy ndarray with ‘dtype’ for its dtype.

Return type ndarray

axes_labels
   Return the list of axes labels

compose(fd, *, eval_points=None, **kwargs)
   Composition of functions.

Performs the composition of functions. The basis is discretized to compute the composition.

Parameters

- **fd** *(FData)* – FData object to make the composition. Should have the same number of samples and image dimension equal to 1.
- **eval_points** *(array_like)* – Points to perform the evaluation. kwargs: Named arguments to be passed to from_data().

concatenate(*others, as_coordinates=False)
   Join samples from a similar FDataBasis object.

Joins samples from another FDataBasis object if they have the same basis.

Parameters

- **others** *(FDataBasis)* – Objects to be concatenated.
- **as_coordinates** *(boolean, optional)* – If False concatenates as new samples, else, concatenates the other functions as new components of the image. Defaults to False.

Returns FDataBasis object with the samples from the original objects.

Return type FDataBasis

Todo: By the moment, only unidimensional objects are supported in basis representation.

coordinates
   Return a component of the FDataBasis.

If the functional object contains samples \( f : \mathbb{R}^n \to \mathbb{R}^d \), this object allows a component of the vector \( f = (f_1, ..., f_d) \).

Todo: By the moment, only unidimensional objects are supported in basis form.

copy(*, basis=None, coefficients=None, dataset_label=None, axes_labels=None, extrapolation=None, keepdims=None)
   FDataBasis copy

1.1. Representation of functional Data
cov(eval_points=None)
Compute the covariance of the functional data object.
A numerical approach its used. The object its transformed into its discrete representation and
then the covariance matrix is computed.

Parameters eval_points (array_like, optional) – Set of points where the functions
are evaluated to obtain the discrete representation of the object. If none are passed
it calls numpy.linspace with bounds equal to the ones defined in self.domain_range
and the number of points the maximum between 501 and 10 times the number of
basis.

Returns Matrix of covariances.

Return type numpy.darray
derivative(order=1)
Differentiate a FDataBasis object.

Parameters order (int, optional) – Order of the derivative. Defaults to one.
dim_codomain
Return number of dimensions of the image.
dim_domain
Return number of dimensions of the domain.
domain_range
Definition range.
dropna()
Return ExtensionArray without NA values.

Returns valid

Return type ExtensionArray
dtype
The dtype for this extension array, FDataGridDType
evaluate(eval_points, *, derivative=0, extrapolation=None, grid=False, aligned_evaluation=True, keepdims=None)
Evaluate the object or its derivatives at a list of values or a grid.

Parameters

- eval_points (array_like) – List of points where the functions are evaluated. If
  a matrix of shape n_samples x eval_points is given each sample is evaluated at the
  values in the corresponding row in eval_points.
- derivative (int, optional) – Order of the derivative. Defaults to 0.
- extrapolation (str or Extrapolation, optional) – Controls the extrapolation mode for elements outside the domain range. By default it is used the mode defined during the instance of the object.
- grid (bool, optional) – Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays. If true the eval_points should be a list of size dim_domain with the corresponding times for each axis. The return matrix has shape n_samples x len(t1) x len(t2) x ... x len(t_dim_domain) x dim_codomain. If the domain dimension is 1 the parameter has no effect. Defaults to False.
• **keepdims** *(bool, optional)* – If the image dimension is equal to 1 and keepdims is True the return matrix has shape `n_samples x eval_points x 1` else `n_samples x eval_points`. By default is used the value given during the instance of the object.

**Returns** Matrix whose rows are the values of the each function at the values specified in `eval_points`.

**Return type** *(np.darray)*

**extrapolation**

Return default type of extrapolation.

**extrapolator_evaluator**

Return the evaluator constructed by the extrapolator.

**factorize**(na_sentinel: int = -1) → Tuple[numpy.ndarray, pandas.core.dtypes.generic.ABCExtensionArray]

Encode the extension array as an enumerated type.

**Parameters**

- **na_sentinel** *(int, default -1)* – Value to use in the `codes` array to indicate missing values.

**Returns**

- **codes** *(ndarray)* – An integer NumPy array that’s an indexer into the original ExtensionArray.
- **uniques** *(ExtensionArray)* – An ExtensionArray containing the unique values of `self`.

**Note:** uniques will not contain an entry for the NA value of the ExtensionArray if there are any missing values present in `self`.

**See also:**

`factorize()` Top-level factorize method that dispatches here.

**Notes**

*pandas.factorize()* offers a `sort` keyword as well.

**fillna**(value=None, method=None, limit=None)

Fill NA/NaN values using the specified method.

**Parameters**

- **value** *(scalar, array-like)* – If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like ‘value’ can be given. It’s expected that the array-like have the same length as ‘self’.
- **limit** *(int, default None)* – If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled.

1.1. Representation of functional Data
If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

Returns With NA/NaN filled.

Return type ExtensionArray

classmethod from_data(data_matrix, sample_points, basis, method='cholesky', keepdims=False)

Transform raw data to a smooth functional form.

Takes functional data in a discrete form and makes an approximates it to the closest function that can be generated by the basis. This function does not attempt to smooth the original data. If smoothing is desired, it is better to use BasisSmother.

The fit is made so as to reduce the sum of squared errors \[RS05-5-2-5\]:

\[ SSE(c) = (y - \Phi c)'(y - \Phi c) \]

where \( y \) is the vector or matrix of observations, \( \Phi \) the matrix whose columns are the basis functions evaluated at the sampling points and \( c \) the coefficient vector or matrix to be estimated.

By deriving the first formula we obtain the closed formed of the estimated coefficients matrix:

\[ \hat{c} = (\Phi'\Phi)^{-1}\Phi'y \]

The solution of this matrix equation is done using the cholesky method for the resolution of a LS problem. If this method throughs a rounding error warning you may want to use the QR factorisation that is more numerically stable despite being more expensive to compute. \[RS05-5-2-7\]

Parameters

- **data_matrix** (*array_like*) – List or matrix containing the observations. If a matrix each row represents a single functional datum and the columns the different observations.
- **sample_points** (*array_like*) – Values of the domain where the previous data were taken.
- **basis** – (Basis): Basis used.
- **method** (*str*) – Algorithm used for calculating the coefficients using the least squares method. The values admitted are ‘cholesky’ and ‘qr’ for Cholesky and QR factorisation methods respectively.

Returns

- **Representation of the data in a functional form** as product of coefficients by basis functions.

Return type *FDataBasis*

Examples

```python
>>> import numpy as np
>>> t = np.linspace(0, 1, 5)
>>> x = np.sin(2 * np.pi * t) + np.cos(2 * np.pi * t)
>>> x
array([[ 1.,  1., -1., -1.,  1.]])
```
```python
>>> basis = Fourier((0, 1), n_basis=3)
>>> fd = FDataBasis.from_data(x, t, basis)
>>> fd.coefficients.round(2)
array([[ 0. , 0.71, 0.71]])
```

### References

**gmean** *(eval_points=None)*

Compute the geometric mean of the functional data object.

A numerical approach is used. The object is transformed into its discrete representation and then the geometric mean is computed and then the object is taken back to the basis representation.

**Parameters**

- **eval_points** *(array_like, optional)* — Set of points where the functions are evaluated to obtain the discrete representation of the object. If none are passed it calls `numpy.linspace` with bounds equal to the ones defined in `self.domain_range` and the number of points the maximum between 501 and 10 times the number of basis.

**Returns**

Geometric mean of the original object.

**Return type** *FDataBasis*

**inner_product** *(other, lfd_self=None, lfd_other=None, weights=None)*

Return an inner product matrix given a FDataBasis object.

The inner product of two functions is defined as

\[
< x, y > = \int_a^b x(t)y(t)dt
\]

When we talk about FDataBasis objects, they have many samples, so we talk about inner product matrix instead. So, for two FDataBasis objects we define the inner product matrix as

\[
a_{ij} = < x_i, y_j > = \int_a^b x_i(s)y_j(s)ds
\]

where \( f_i(s), g_j(s) \) are the \( i^{th} \) \( j^{th} \) sample of each object. The return matrix has a shape of \( IxJ \) where \( I \) and \( J \) are the number of samples of each object respectively.

**Parameters**

- **other** *(FDataBasis, Basis)* — FDataBasis object containing the second object to make the inner product
- **lfd_self** *(Lfd)* — LinearDifferentialOperator object for the first function evaluation
- **lfd_other** *(Lfd)* — LinearDifferentialOperator object for the second function evaluation
- **weights** *(FDataBasis)* — a FDataBasis object with only one sample that defines the weight to calculate the inner product

**Returns**

Inner Product matrix.

**Return type** *numpy.array*

**isna()**

A 1-D array indicating if each value is missing.

**Returns**

Array full of False values.
Return type na_values (np.ndarray)

mean(weights=None)
Compute the mean of all the samples in a FDataBasis object.

Returns A FDataBais object with just one sample representing the mean of all the
samples in the original FDataBasis object.

Return type FDataBasis

Examples

```python
>>> basis = Monomial(n_basis=4)
>>> coefficients = [[0.5, 1, 2, .5], [1.5, 1, 4, .5]]
>>> FDataBasis(basis, coefficients).mean()
FDataBasis(
    basis=Monomial(domain_range=[array([0, 1])], n_basis=4),
    coefficients=[[ 1.  1.  3.  0.5]],
    ...
)
```

n_basis
Return number of basis.

n_samples
Return number of samples.

nbytes
The number of bytes needed to store this object in memory.

ndim
Return number of dimensions of the functional data. It is always 1, as each observation is con-
sidered a “scalar” object.

Returns Number of dimensions of the functional data.

Return type int

plot(*args, **kwargs)
Plot the FDatGrid object.

Parameters

- chart (figure object, axe or list of axes, optional) – figure over with
  the graphs are plotted or axis over where the graphs are plotted. If None and
  ax is also None, the figure is initialized.

- derivative (int or tuple, optional) – Order of derivative to be plotted. In
  case of surfaces a tuple with the order of derivation in each direction can be passed.
  See evaluate() to obtain more information. Defaults 0.

- fig (figure object, optional) – figure over with the graphs are plotted in case
  ax is not specified. If None and ax is also None, the figure is initialized.

- ax (list of axis objects, optional) – axis over where the graphs are plotted.
  If None, see param fig.

- n_rows (int, optional) – designates the number of rows of the figure to plot the
different dimensions of the image. Only specified if fig and ax are None.

- n_cols (int, optional) – designates the number of columns of the figure to plot
  the different dimensions of the image. Only specified if fig and ax are None.
• **n_points** *(int or tuple, optional)* – Number of points to evaluate in the plot. In case of surfaces a tuple of length 2 can be passed with the number of points to plot in each axis, otherwise the same number of points will be used in the two axes. By default in unidimensional plots will be used 501 points; in surfaces will be used 30 points per axis, which makes a grid with 900 points.

• **domain_range** *(tuple or list of tuples, optional)* – Range where the function will be plotted. In objects with unidimensional domain the domain range should be a tuple with the bounds of the interval; in the case of surfaces a list with 2 tuples with the ranges for each dimension. Default uses the domain range of the functional object.

• **group** *(list of int)* – contains integers from [0 to number of labels) indicating to which group each sample belongs to. Then, the samples with the same label are plotted in the same color. If None, the default value, each sample is plotted in the color assigned by matplotlib.pyplot.rcParams["axes.prop_cycle"].

• **group_colors** *(list of colors)* – colors in which groups are represented, there must be one for each group. If None, each group is shown with distinct colors in the “Greys” colormap.

• **group_names** *(list of str)* – name of each of the groups which appear in a legend, there must be one for each one. Defaults to None and the legend is not shown.

• ****kwargs** – if dim_domain is 1, keyword arguments to be passed to the matplotlib.pyplot.plot function; if dim_domain is 2, keyword arguments to be passed to the matplotlib.pyplot.plot_surface function.

Returns figure object in which the graphs are plotted.

Return type fig (figure object)

**ravel**(order='C') → pandas.core.dtypes.generic.ABCExtensionArray

Return a flattened view on this array.


Returns

Return type ExtensionArray

Notes

• Because ExtensionArrays are 1D-only, this is a no-op.

• The “order” argument is ignored, is for compatibility with NumPy.

**repeat**(repeats, axis=None)

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

Parameters

• **repeats** *(int or array of ints)* – The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.
- **axis** (*None*) – Must be *None*. Has no effect but is accepted for compatibility with numpy.

**Returns** repeated_array – Newly created ExtensionArray with repeated elements.

**Return type** ExtensionArray

**See also:**

- **Series.repeat()** Equivalent function for Series.
- **Index.repeat()** Equivalent function for Index.
- **numpy.repeat()** Similar method for numpy.ndarray.
- **ExtensionArray.take()** Take arbitrary positions.

**Examples**

```python
>>> cat = pd.Categorical(['a', 'b', 'c'])
>>> cat
[a, b, c]
Categories (3, object): [a, b, c]

>>> cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]

>>> cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

**searchsorted**(value, side='left', sorter=None)

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array `self` (a) such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

Assuming that `self` is sorted:

<table>
<thead>
<tr>
<th>side</th>
<th>returned index <code>i</code> satisfies</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td><code>self[i-1] &lt; value &lt;= self[i]</code></td>
</tr>
<tr>
<td>right</td>
<td><code>self[i-1] &lt;= value &lt; self[i]</code></td>
</tr>
</tbody>
</table>

**Parameters**

- **value** (*array_like*) – Values to insert into `self`.

- **side** (*left*, *right*, optional) – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).

- **sorter** (*1-D array_like, optional*) – Optional array of integer indices that sort array `a` into ascending order. They are typically the result of argsort.

**Returns** Array of insertion points with the same shape as `value`.

**Return type** array of ints
See also:

\texttt{numpy.searchsorted()} Similar method from NumPy.

\texttt{shape}

Return a tuple of the array dimensions.

\texttt{shift(shifts, *, restrict\_domain=False, extrapolation=None, eval\_points=None, **kwargs)}

Perform a shift of the curves.

\textbf{Parameters}

- \texttt{shifts (array\_like or numeric)} – List with the shift corresponding for each sample or numeric with the shift to apply to all samples.
- \texttt{restrict\_domain (bool, optional)} – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- \texttt{extrapolation (str or Extrapolation, optional)} – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in \texttt{fd}. See extrapolation to more information.
- \texttt{eval\_points (array\_like, optional)} – Set of points where the functions are evaluated to obtain the discrete representation of the object to operate. If an empty list is passed it calls \texttt{numpy.linspace} with bounds equal to the ones defined in \texttt{fd.domain\_range} and the number of points the maximum between 201 and 10 times the number of basis plus 1.
- \texttt{**kwargs} – Keyword arguments to be passed to \texttt{from\_data()}.

\textbf{Returns} \texttt{FDataBasis} with the shifted data.

\texttt{take(indices, allow\_fill=False, fill\_value=None, axis=0)}

Take elements from an array.

\textbf{Parameters}

- \texttt{indices (sequence of integers)} – Indices to be taken.
- \texttt{allow\_fill (bool, default False)} – How to handle negative values in \texttt{indices}.
  - False: negative values in \texttt{indices} indicate positional indices from the right (the default). This is similar to \texttt{numpy.take()}.
  - True: negative values in \texttt{indices} indicate missing values. These values are set to \texttt{fill\_value}. Any other negative values raise a \texttt{ValueError}.
- \texttt{fill\_value (any, optional)} – Fill value to use for NA-indices when \texttt{allow\_fill} is True. This may be \texttt{None}, in which case the default NA value for the type, \texttt{self.dtype.na\_value}, is used. For many ExtensionArrays, there will be two representations of \texttt{fill\_value}: a user-facing “boxed” scalar, and a low-level physical NA value. \texttt{fill\_value} should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

\textbf{Returns} \texttt{FData}

\textbf{Raises}

- \texttt{IndexError} – When the indices are out of bounds for the array.
- \texttt{ValueError} – When \texttt{indices} contains negative values other than \texttt{-1} and \texttt{allow\_fill} is True.
Notes

ExtensionArray.take is called by Series.__getitem__, .loc, iloc, when indices is a sequence of values. Additionally, it’s called by Series.reindex(), or any other method that causes realignment, with a fill_value.

See also:
numpy.take pandas.api.extensions.take

times(other)

“Provides a numerical approximation of the multiplication between an FDataObject to other object

Parameters

other (int, list, FDataBasis) –

Object to multiply with the FDataBasis object.

• int: Multiplies all samples with the value
• list: multiply each values with the samples respectively. Length should match with FDataBasis samples
• FDataBasis: if there is one sample it multiplies this with all the samples in the object. If not, it multiplies each sample respectively. Samples should match

Returns FDataBasis object containing the multiplication

Return type (FDataBasis)

to_basis(basis, eval_points=None, **kwargs)

Return the basis representation of the object.

Parameters

• basis (Basis) – basis object in which the functional data are going to be represented.
• **kwargs – keyword arguments to be passed to FDataBasis.from_data().

Returns Basis representation of the functional data object.

Return type FDataBasis

to_grid(eval_points=None)

Return the discrete representation of the object.

Parameters
eval_points (array_like, optional) – Set of points where the functions are evaluated. If none are passed it calls numpy.linspace with bounds equal to the ones defined in self.domain_range and the number of points the maximum between 501 and 10 times the number of basis.

Returns Discrete representation of the functional data object.

Return type FDataGrid
Examples

```python
>>> fd = FDataBasis(coefficients=[[1, 1, 1], [1, 0, 1]],
               basis=Monomial((0,5), n_basis=3))
>>> fd.to_grid([0, 1, 2])
FDataGrid(
    array([[ 1.],
           [ 3.],
           [ 7.]],
    sample_points=[array([0, 1, 2])],
    domain_range=array([[0, 5]]),
    ...)
```

to_list()
Splits FDataBasis samples into a list
to_numpy()
Returns a numpy array with the objects
unique()
Compute the ExtensionArray of unique values.

Returns uniques
Return type ExtensionArray

var(eval_points=None)
Compute the variance of the functional data object.
A numerical approach its used. The object its transformed into its discrete representation and
then the variance is computed and then the object is taken back to the basis representation.

Parameters eval_points (array_like, optional) – Set of points where the functions
are evaluated to obtain the discrete representation of the object. If none are passed
it calls numpy.linspace with bounds equal to the ones defined in self.domain_range
and the number of points the maximum between 501 and 10 times the number of
basis.

Returns Variance of the original object.

Return type FDataBasis

view(dtype=None) → Union[pandas.core.dtypes.generic.ABCExtensionArray, numpy.ndarray]
Return a view on the array.

Parameters dtype (str, np.dtype, or ExtensionDtype, optional) – Default None.

Returns A view of the ExtensionArray.

Return type ExtensionArray

The following classes are used to define different basis systems.

| skfda.representation.basis.BSpline(…) | BSpline basis. |
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skfda.representation.basis.Fourier(...)</code></td>
<td>Fourier basis.</td>
</tr>
<tr>
<td><code>skfda.representation.basis.Monomial(...)</code></td>
<td>Monomial basis.</td>
</tr>
<tr>
<td><code>skfda.representation.basis.Constant(...)</code></td>
<td>Constant basis.</td>
</tr>
</tbody>
</table>

**BSpline**

```python
class skfda.representation.basis.BSpline(domain_range=None, n_basis=None, order=4, knots=None)
```

BSpline basis.

BSpline basis elements are defined recursively as:

\[
B_{i,1}(x) = \begin{cases} 
1 & \text{if } t_i \leq x < t_{i+1}, \\
0 & \text{otherwise}
\end{cases}
\]

\[
B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)
\]

Where k indicates the order of the spline.

Implementation details: In order to allow a discontinuous behaviour at the boundaries of the domain it is necessary to placing m knots at the boundaries [RS05]. This is automatically done so that the user only has to specify a single knot at the boundaries.

- **domain_range**
  - A tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.
  - **Type** tuple

- **n_basis**
  - Number of functions in the basis.
  - **Type** int

- **order**
  - Order of the splines. One greater than their degree.
  - **Type** int

- **knots**
  - List of knots of the spline functions.
  - **Type** list

**Examples**

Constructs specifying number of basis and order.

```python
>>> bss = BSpline(n_basis=8, order=4)
```

If no order is specified defaults to 4 because cubic splines are the most used. So the previous example is the same as:

```python
>>> bss = BSpline(n_basis=8)
```

It is also possible to create a BSpline basis specifying the knots.
Once we create a basis we can evaluate each of its functions at a set of points.

```
>>> bss = BSpline(knots=[0, 0.2, 0.4, 0.6, 0.8, 1])
```  
And evaluates first derivative

```
>>> bss.evaluate([0, 0.5, 1], derivative=1)
array([[-2., -1., 0.],
       [ 2., 0., -2.],
       [ 0., 1., 2.]])
```

**References**

**Methods**

```python
__init__(domain_range=None, n_basis=None, order=4, knots=None)
Bspline basis constructor.
```

- **basis_of_product(other)**
  Multiplication of two Bspline Basis

- **copy()**
  Basis copy

- **default_basis_of_product(one, other)**
  Default multiplication for a pair of basis

- **evaluate(eval_points[, derivative])**
  Evaluate Basis objects and its derivatives.

- **gram_matrix()**
  Return the Gram Matrix of a basis

- **inner_product(other)**

- **penalty([derivative_degree, coefficients])**
  Return a penalty matrix given a differential operator.

- **plot([chart, derivative])**
  Plot the basis object or its derivatives.

- **rbasis_of_product(other)**
  Multiplication of a Bspline Basis with other basis

- **rescale(domain_range)**
  Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.

- **same_domain(other)**
  Returns if two basis are defined on the same domain range.

- **to_basis()**

```python
__init__(domain_range=None, n_basis=None, order=4, knots=None)
Bspline basis constructor.
```

**Parameters**

- **domain_range** *(tuple, optional)* – Definition of the interval where the basis defines a space. Defaults to (0,1) if knots are not specified. If knots are specified defaults to the first and last element of the knots.

- **n_basis** *(int, optional)* – Number of splines that form the basis.

- **order** *(int, optional)* – Order of the splines. One greater that their degree. Defaults to 4 which mean cubic splines.

1.1. Representation of functional Data
• **knots** (*array_like*) – List of knots of the splines. If domain_range is specified the first and last elements of the knots have to match with it.

**Attributes**

- **domain_range**
- **inknots**
- **knots**

**basis_of_product**(*other*)

Multiplication of two Bspline Basis

**copy**()

Basis copy

**static default_basis_of_product**(*one, other*)

Default multiplication for a pair of basis

**evaluate**(*eval_points, derivative=0*)

Evaluate Basis objects and its derivatives.

Evaluates the basis function system or its derivatives at a list of given values.

**Parameters**

- **eval_points** (*array_like*) – List of points where the basis is evaluated.
- **derivative** (*int, optional*) – Order of the derivative. Defaults to 0.

**Returns** Matrix whose rows are the values of the each basis function or its derivatives at the values specified in eval_points.

**Return type** (*numpy.darray*)

**gram_matrix**()

Return the Gram Matrix of a basis

The Gram Matrix is defined as

\[ G_{ij} = \langle \phi_i, \phi_j \rangle \]

where \( \phi_i \) is the \( i \)th element of the basis. This is a symmetric matrix and positive-semidefinite.

**Returns** Gram Matrix of the basis.

**Return type** (*numpy.array*)

- **inknots**

Return number of basis.

**penalty**(*derivative_degree=None, coefficients=None*)

Return a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as [RS05-5-6-2-3]:

\[ R_{ij} = \int L\phi_i(s)L\phi_j(s)ds \]

where \( \phi_i(s) \) for \( i = 1, 2, ..., n \) are the basis functions and \( L \) is a differential operator.
Parameters

- **derivative_degree** (int) – Integer indicating the order of the derivative or $f''(x)$.
- **coefficients** (list) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple $(1, 0, \text{numpy.sin})$ means $1 + \sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

References

**plot**(*chart=None, *, derivative=0, **kwargs*)

Plot the basis object or its derivatives.

Parameters

- **chart** (figure object, axe or list of axes, optional) – figure over with the graphs are plotted or axis over where the graphs are plotted.
- **derivative** (int or tuple, optional) – Order of derivative to be plotted. Defaults 0.
- ****kwargs – keyword arguments to be passed to the fdata.plot function.

Returns figure object in which the graphs are plotted.

Return type fig (figure)

**rbasis_of_product**(*other*)

Multiplication of a Bspline Basis with other basis

**rescale**(*domain_range=None*)

Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds. The knots of the BSpline will be rescaled in the new interval.

Parameters **domain_range** (tuple, optional) – Definition of the interval where the basis defines a space. Defaults uses the same as the original basis.

**same_domain**(*other*)

Returns if two basis are defined on the same domain range.

Parameters **other** (Basis) – Basis to check the domain range definition

Examples using skfda.representation.basis.BSpline

- Representation of functional data
- Extrapolation

**Fourier**

**class** skfda.representation.basis.Fourier(*domain_range=None, n_basis=3, period=None*)

Fourier basis.
Defines a functional basis for representing functions on a fourier series expansion of period $T$. The number of basis is always odd. If instantiated with an even number of basis, they will be incremented automatically by one.

$$\phi_0(t) = \frac{1}{\sqrt{2}}$$

$$\phi_{2n-1}(t) = \sin \left( \frac{2\pi n}{T} t \right)$$

$$\phi_{2n}(t) = \cos \left( \frac{2\pi n}{T} t \right)$$

Actually this basis functions are not orthogonal but not orthonormal. To achieve this they are divided by its norm: $\sqrt{\frac{T}{2}}$.

**domain_range**
A tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

Type: tuple

**n_basis**
Number of functions in the basis.

Type: int

**period**
Period ($T$).

Type: int or float

**Examples**

Constructs specifying number of basis, definition interval and period.

```python
>>> fb = Fourier((0, np.pi), n_basis=3, period=1)
>>> fb.evaluate([0, np.pi / 4, np.pi / 2, np.pi]).round(2)
array([[ 1. , 1. , 1. , 1. ],
       [ 0. , -1.38, -0.61, 1.1 ],
       [ 1.41, 0.31, -1.28, 0.89]])
```

And evaluate second derivative

```python
>>> fb.evaluate([0, np.pi / 4, np.pi / 2, np.pi],
               derivative = 2).round(2)
array([[ 0. , 0. , 0. , 0. ],
       [-0. , 54.46, 24.02, -43.37],
       [-55.83, -12.32, 50.4 , -35.16]])
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Construct a Fourier object.</td>
</tr>
<tr>
<td>basis_of_product</td>
<td>Multiplication of two Fourier Basis</td>
</tr>
<tr>
<td>copy()</td>
<td>Basis copy</td>
</tr>
</tbody>
</table>
Table 13 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default_basis_of_product(one, other)</code></td>
<td>Default multiplication for a pair of basis</td>
</tr>
<tr>
<td><code>evaluate(eval_points[, derivative])</code></td>
<td>Evaluate Basis objects and its derivatives.</td>
</tr>
<tr>
<td><code>gram_matrix()</code></td>
<td>Return the Gram Matrix of a basis</td>
</tr>
<tr>
<td><code>inner_product(other)</code></td>
<td>Multiplication of a Fourier Basis with other Basis</td>
</tr>
<tr>
<td><code>penalty([derivative_degree, coefficients])</code></td>
<td>Return a penalty matrix given a differential operator.</td>
</tr>
<tr>
<td><code>plot([chart, derivative])</code></td>
<td>Plot the basis object or its derivatives.</td>
</tr>
<tr>
<td><code>rbasis_of_product(other)</code></td>
<td>Multiplication of a Fourier Basis with other Basis</td>
</tr>
<tr>
<td><code>rescale([domain_range, rescale_period])</code></td>
<td>Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.</td>
</tr>
<tr>
<td><code>same_domain(other)</code></td>
<td>Returns if two basis are defined on the same domain range.</td>
</tr>
<tr>
<td><code>to_basis()</code></td>
<td></td>
</tr>
</tbody>
</table>

`__init__` (domain_range=None, n_basis=3, period=None)

Construct a Fourier object.

It forces the object to have an odd number of basis. If n_basis is even, it is incremented by one.

Parameters

- `domain_range` (tuple) – Tuple defining the domain over which the function is defined.
- `n_basis` (int) – Number of basis functions.
- `period` (int or float) – Period of the trigonometric functions that define the basis.

Attributes

- `domain_range`
- `period`

`basis_of_product(other)`

Multiplication of two Fourier Basis

`copy()`

Basis copy

`static default_basis_of_product(one, other)`

Default multiplication for a pair of basis

`evaluate(eval_points, derivative=0)`

Evaluate Basis objects and its derivatives.

Evaluates the basis function system or its derivatives at a list of given values.

Parameters

- `eval_points` (array_like) – List of points where the basis is evaluated.
- `derivative` (int, optional) – Order of the derivative. Defaults to 0.

Returns Matrix whose rows are the values of the each basis function or its derivatives at the values specified in eval_points.

1.1. Representation of functional Data
Return type (numpy.array)

gram_matrix()
Return the Gram Matrix of a basis

The Gram Matrix is defined as

$$G_{ij} = \langle \phi_i, \phi_j \rangle$$

where $\phi_i$ is the $i$th element of the basis. This is a symmetric matrix and positive-semidefinite.

Returns Gram Matrix of the basis.

Return type numpy.array

penalty(derivative_degree=None, coefficients=None)
Return a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as [RS05-5-6-2-4]:

$$R_{ij} = \int L\phi_i(s)L\phi_j(s)ds$$

where $\phi_i(s)$ for $i = 1, 2, ..., n$ are the basis functions and $L$ is a differential operator.

Parameters

- derivative_degree (int) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is $f''(x)$.
- coefficients (list) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means $1 + \sin(x)D^2$. Only used if derivative degree is None.

Returns Penalty matrix.

Return type numpy.array

References

plot(chart=None, *, derivative=0, **kwargs)
Plot the basis object or its derivatives.

Parameters

- chart (figure object, axe or list of axes, optional) – figure over with the graphs are plotted or axis over where the graphs are plotted.
- derivative (int or tuple, optional) – Order of derivative to be plotted. Defaults 0.
- **kwargs – keyword arguments to be passed to the fdata.plot function.

Returns figure object in which the graphs are plotted.

Return type fig (figure)

rbasis_of_product(other)
Multiplication of a Fourier Basis with other Basis
rescale(domain_range=None, *, rescale_period=False)

Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.

**Parameters**

- **domain_range** *(tuple, optional)* – Definition of the interval where the basis defines a space. Defaults uses the same as the original basis.
- **rescale_period** *(bool, optional)* – If true the period will be rescaled using the ratio between the lengths of the new and old interval. Defaults to False.

same_domain(other)

Returns if two basis are defined on the same domain range.

**Parameters**

- **other** *(Basis)* – Basis to check the domain range definition

Examples using skfda.representation.basis.Fourier

- *Shift Registration*
- *Neighbors Functional Regression*
- *Representation of functional data*
- *Extrapolation*

Monomial

class skfda.representation.basis.Monomial(domain_range=None, n_basis=1)

Monomial basis.

Basis formed by powers of the argument $t$:

$1, t, t^2, t^3, ...$

domain_range

a tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

**Type** tuple

n_basis

number of functions in the basis.

**Type** int

Examples

Defines a monomial base over the interval $[0, 5]$ consisting on the first 3 powers of $t$: $1, t, t^2$.

```python
>>> bs_mon = Monomial((0, 5), n_basis=3)
```

And evaluates all the functions in the basis in a list of discrete values.
>>> bs_mon.evaluate([0, 1, 2])
array([[ 1.,  1.,  1.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  4.]])

And also evaluates its derivatives

>>> bs_mon.evaluate([0, 1, 2], derivative=1)
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.],
       [ 0.,  2.,  4.]])

Methods

__init__((domain_range, n_basis)) Basis constructor.

basis_of_product(other) Multiplication of a Monomial Basis with other Basis

copy() Basis copy

default_basis_of_product(one, other) Default multiplication for a pair of basis

evaluate(eval_points[, derivative]) Evaluate Basis objects and its derivatives.

gram_matrix() Return the Gram Matrix of a basis

inner_product(other) Return a penalty matrix given a differential operator.

penalty([derivative_degree, coefficients]) Plot the basis object or its derivatives.

plot([chart, derivative]) Multiplication of a Monomial Basis with other Basis

rescale([domain_range]) Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.

same_domain(other) Returns if two basis are defined on the same domain range.

to_basis()__init__(domain_range=None, n_basis=1) Basis constructor.

Parameters

- domain_range (tuple or list of tuples, optional) – Definition of the interval where the basis defines a space. Defaults to (0,1).
- n_basis – Number of functions that form the basis. Defaults to 1.

Attributes
**domain_range**

**basis_of_product**(other)
Multiplication of a Monomial Basis with other Basis

**copy**()
Basis copy

**static default_basis_of_product**(one, other)
Default multiplication for a pair of basis

**evaluate**(eval_points, derivative=0)
Evaluate Basis objects and its derivatives.
Evaluates the basis function system or its derivatives at a list of given values.

**Parameters**
- **eval_points** ([array_like]) – List of points where the basis is evaluated.
- **derivative** ([int, optional]) – Order of the derivative. Defaults to 0.

**Returns** Matrix whose rows are the values of the each basis function or its derivatives at the values specified in eval_points.

**Return type** (numpy.array)

**gram_matrix**()
Return the Gram Matrix of a basis
The Gram Matrix is defined as

\[ G_{ij} = \langle \phi_i, \phi_j \rangle \]

where \( \phi_i \) is the ith element of the basis. This is a symmetric matrix and positive-semidefinite.

**Returns** Gram Matrix of the basis.

**Return type** numpy.array

**penalty**(derivative_degree=None, coefficients=None)
Return a penalty matrix given a differential operator.
The differential operator can be either a derivative of a certain degree or a more complex operator.
The penalty matrix is defined as [RS05-5-6-2-1]:

\[ R_{ij} = \int L \phi_i(s) L \phi_j(s) ds \]

where \( \phi_i(s) \) for \( i = 1, 2, ..., n \) are the basis functions and \( L \) is a differential operator.

**Parameters**
- **derivative_degree** ([int]) – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is \( f''(x) \).
- **coefficients** ([list]) – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple (1, 0, numpy.sin) means \( 1 + \sin(x)D^2 \). Only used if derivative degree is None.

**Returns** Penalty matrix.

**Return type** numpy.array
Examples

```python
>>> Monomial(n_basis=4).penalty(2)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  4.,  6.],
       [ 0.,  0.,  6., 12.]])
```

References

```python
plot(chart=None, *, derivative=0, **kwargs)
```
Plot the basis object or its derivatives.

Parameters

- **chart** *(figure object, axe or list of axes, optional)* – figure over with the graphs are plotted or axis over where the graphs are plotted.
- **derivative** *(int or tuple, optional)* – Order of derivative to be plotted. Defaults 0.
- ****kwargs – keyword arguments to be passed to the fdata.plot function.

Returns figure object in which the graphs are plotted.

Return type fig (figure)

```python
rbasis_of_product(other)
```
Multiplication of a Monomial Basis with other Basis

```python
rescale(domain_range=None)
```
Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.

Parameters **domain_range** *(tuple, optional)* – Definition of the interval where the basis defines a space. Defaults uses the same as the original basis.

```python
same_domain(other)
```
Returns if two basis are defined on the same domain range.

Parameters **other** *(Basis)* – Basis to check the domain range definition

Examples using `skfda.representation.basis.Monomial`

- Extrapolation

Constant

```python
class skfda.representation.basis.Constant(domain_range=None)
```
Constant basis.

Basis for constant functions

```python
domain_range
```
a tuple of length 2 containing the initial and end values of the interval over which the basis can be evaluated.

Type tuple
Examples

Defines a constant base over the interval \([0, 5]\) consisting on the constant function 1 on \([0, 5]\).

```python
>>> bs_cons = Constant((0,5))
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>([domain_range])</td>
<td>Constant basis constructor.</td>
</tr>
<tr>
<td><code>basis_of_product</code> (other)</td>
<td>Multiplication of a Constant Basis with other Basis</td>
</tr>
<tr>
<td><code>copy</code> ()</td>
<td>Basis copy</td>
</tr>
<tr>
<td><code>default_basis_of_product</code> (one, other)</td>
<td>Default multiplication for a pair of basis</td>
</tr>
<tr>
<td><code>evaluate</code> (eval_points[, derivative])</td>
<td>Evaluate Basis objects and its derivatives.</td>
</tr>
<tr>
<td><code>gram_matrix</code> ()</td>
<td>Return the Gram Matrix of a basis</td>
</tr>
<tr>
<td><code>inner_product</code> (other)</td>
<td></td>
</tr>
<tr>
<td><code>penalty</code> ([derivative_degree, coefficients])</td>
<td>Return a penalty matrix given a differential operator.</td>
</tr>
<tr>
<td><code>plot</code> ([chart, derivative])</td>
<td>Plot the basis object or its derivatives.</td>
</tr>
<tr>
<td><code>rbasis_of_product</code> (other)</td>
<td>Multiplication of a Constant Basis with other Basis</td>
</tr>
<tr>
<td><code>rescale</code> ([domain_range])</td>
<td>Return a copy of the basis with a new domain range, with the corresponding values rescaled to the new bounds.</td>
</tr>
<tr>
<td><code>same_domain</code> (other)</td>
<td>Returns if two basis are defined on the same domain range.</td>
</tr>
<tr>
<td><code>to_basis</code> ()</td>
<td></td>
</tr>
</tbody>
</table>

```python
__init__ (domain_range=None)
Constant basis constructor.

Parameters
• domain_range (tuple) – Tuple defining the domain over which the
• is defined. (function) –

Attributes

domain_range

basis_of_product (other)
Multiplication of a Constant Basis with other Basis

copy ()
Basis copy

static default_basis_of_product (one, other)
Default multiplication for a pair of basis

evaluate (eval_points, derivative=0)
Evaluate Basis objects and its derivatives.
```
Evaluates the basis function system or its derivatives at a list of given values.

**Parameters**
- `eval_points (array_like)` – List of points where the basis is evaluated.
- `derivative (int, optional)` – Order of the derivative. Defaults to 0.

**Returns** Matrix whose rows are the values of the each basis function or its derivatives at the values specified in `eval_points`.

**Return type** (numpy.darray)

```python
gram_matrix()
```
Return the Gram Matrix of a basis

The Gram Matrix is defined as

\[ G_{ij} = \langle \phi_i, \phi_j \rangle \]

where \( \phi_i \) is the \( i \)th element of the basis. This is a symmetric matrix and positive-semidefinite.

**Returns** Gram Matrix of the basis.

**Return type** numpy.array

```python
penalty(derivative_degree=None, coefficients=None)
```
Return a penalty matrix given a differential operator.

The differential operator can be either a derivative of a certain degree or a more complex operator.

The penalty matrix is defined as [RS05-5-6-2-2]:

\[ R_{ij} = \int L\phi_i(s)L\phi_j(s)ds \]

where \( \phi_i(s) \) for \( i = 1, 2, ..., n \) are the basis functions and \( L \) is a differential operator.

**Parameters**
- `derivative_degree (int)` – Integer indicating the order of the derivative or . For instance 2 means that the differential operator is \( f''(x) \).
- `coefficients (list)` – List of coefficients representing a differential operator. An iterable indicating coefficients of derivatives (which can be functions). For instance the tuple \((1, 0, \text{numpy.sin})\) means \( 1 + \sin(x)D^2 \). Only used if derivative degree is None.

**Returns** Penalty matrix.

**Return type** numpy.array

```
Examples

>>> Constant((0,5)).penalty(0)
array([[5]])
>>> Constant().penalty(1)
array([[ 0.]])
```
References

```python
plot(chart=None, *, derivative=0, **kwargs)
Plot the basis object or its derivatives.

Parameters

- chart (figure object, axe or list of axes, optional) – figure over with
  the graphs are plotted or axis over where the graphs are plotted.
- derivative (int or tuple, optional) – Order of derivative to be plotted. De-
 faults 0.
- **kwargs – keyword arguments to be passed to the fdata.plot function.

Returns figure object in which the graphs are plotted.

Return type fig (figure)
```

```python
rbasis_of_product(other)
Multiplication of a Constant Basis with other Basis
```

```python
rescale(domain_range=None)
Return a copy of the basis with a new domain range, with the corresponding values rescaled to
the new bounds.

Parameters domain_range (tuple, optional) – Definition of the interval where the
basis defines a space. Defaults uses the same as the original basis.
```

```python
same_domain(other)
Returns if two basis are defined on the same domain range.

Parameters other (Basis) – Basis to check the domain range definition
```

1.1.3 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.

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

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

Class `skfda.representation.FData`

```python
class skfda.representation.FData(extrapolation, dataset_label, axes_labels, keepdims)
 Defines the structure of a functional data object.
```

- **n_samples**
  Number of samples.
  
  Type int

- **dim_domain**
  Dimension of the domain.
  
  Type int

- **dim_codomain**
  Dimension of the image.
**extrapolation**
Default extrapolation mode.

**dataset_label**
name of the dataset.

**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.

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

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>Initialize self. See <code>help(type(self))</code> for accurate signature.</td>
</tr>
<tr>
<td><code>argsort</code></td>
<td>Return the indices that would sort this array.</td>
</tr>
<tr>
<td><code>astype</code></td>
<td>Cast to a NumPy array with ‘dtype’.</td>
</tr>
<tr>
<td><code>compose</code></td>
<td>Composition of functions.</td>
</tr>
<tr>
<td><code>concatenate</code></td>
<td>Join samples from a similar FData object.</td>
</tr>
<tr>
<td><code>copy</code></td>
<td>Make a copy of the object.</td>
</tr>
<tr>
<td><code>derivative</code></td>
<td>Differentiate a FData object.</td>
</tr>
<tr>
<td><code>dropna</code></td>
<td>Return ExtensionArray without NA values.</td>
</tr>
<tr>
<td><code>evaluate</code></td>
<td>Evaluate the object or its derivatives at a list of values or a grid.</td>
</tr>
<tr>
<td><code>factorize</code></td>
<td>Encode the extension array as an enumerated type.</td>
</tr>
<tr>
<td><code>fillna</code></td>
<td>Fill NA/NaN values using the specified method.</td>
</tr>
<tr>
<td><code>isna</code></td>
<td>A 1-D array indicating if each value is missing.</td>
</tr>
<tr>
<td><code>mean</code></td>
<td>Compute the mean of all the samples.</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>Plot the FDatGrid object.</td>
</tr>
<tr>
<td><code>ravel</code></td>
<td>Return a flattened view on this array.</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>Repeat elements of a ExtensionArray.</td>
</tr>
<tr>
<td><code>searchsorted</code></td>
<td>Find indices where elements should be inserted to maintain order.</td>
</tr>
<tr>
<td><code>shift</code></td>
<td>Perform a shift of the curves.</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Take elements from an array.</td>
</tr>
<tr>
<td><code>to_basis</code></td>
<td>Return the basis representation of the object.</td>
</tr>
<tr>
<td><code>to_grid</code></td>
<td>Return the discrete representation of the object.</td>
</tr>
<tr>
<td><code>to_numpy</code></td>
<td>Returns a numpy array with the objects</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>Compute the ExtensionArray of unique values.</td>
</tr>
<tr>
<td><code>view</code></td>
<td>Return a view on the array.</td>
</tr>
</tbody>
</table>

`__init__` (extrapolation, dataset_label, axes_labels, keepdims)
Initialize self. See `help(type(self))` for accurate signature.
Attributes

- **axes_labels**
  Return the list of axes labels

- **coordinates**
  Return a component of the FDataGrid.

- **dim_codomain**
  Return number of dimensions of the codomain.

- **dim_domain**
  Return number of dimensions of the domain.

- **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.

- **n_samples**
  Return the number of samples.

- **nbytes**
  The number of bytes needed to store this object in memory.

- **ndim**
  Return number of dimensions of the functional data.

- **shape**
  Return a tuple of the array dimensions.

---

**argsort**

```
(ascending: bool = True, kind: str = 'quicksort', *args, **kwargs) → numpy.ndarray
```

Return the indices that would sort this array.

**Parameters**

- **ascending (bool, default True)** – Whether the indices should result in an ascending or descending sort.


- ****kwargs (**args,)** – passed through to `numpy.argsort()`.

**Returns**

Array of indices that sort `self`. If NaN values are contained, NaN values are placed at the end.

**Return type**

`ndarray`

**See also:**

- `numpy.argsort()` Sorting implementation used internally.

**astype**

```
dtype, copy=True)
```

Cast to a NumPy array with ‘dtype’.

**Parameters**

- **dtype (str or dtype)** – Typecode or data-type to which the array is cast.

- **copy (bool, default True)** – Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

**Returns**

`array` – NumPy ndarray with ‘dtype’ for its dtype.

**Return type**

`ndarray`

---

**axes_labels**

Return the list of axes labels

**compose**

```
f, *, eval_points=None, **kwargs)
```

Composition of functions.
Performs the composition of functions.

**Parameters**
- **fd (FData)** – FData object to make the composition. Should have the same number of samples and image dimension equal to the domain dimension of the object composed.
- **eval_points (array_like)** – Points to perform the evaluation.
- ****kwargs – Named arguments to be passed to the composition method of the specific functional object.

**concatenate(*others, as_coordinates=False)**
Join samples from a similar FData object.

Joins samples from another FData object if it has the same dimensions and has compatible representations.

**Parameters**
- **others (FData)** – other FData objects.
- **as_coordinates (boolean, optional)** – If False concatenates as new samples, else, concatenates the other functions as new components of the image. Defaults to False.

**Returns** FData object with the samples from the two original objects.

**Return type** FData

**coordinates**
Return a component of the FDataGrid.

If the functional object contains multivariate samples $f : \mathbb{R}^n \rightarrow \mathbb{R}^d$, this method returns an iterator of the vector $f = (f_1, ..., f_d)$.

**copy(**kwargs)**
Make a copy of the object.

**Parameters** **kwargs – named args with attributes to be changed in the new copy.

**Returns** A copy of the FData object.

**Return type** FData

**derivative(order=1)**
Differentiate a FData object.

**Parameters** **order (int, optional)** – Order of the derivative. Defaults to one.

**Returns** Functional object containing the derivative.

**Return type** FData

**dim_codomain**
Return number of dimensions of the codomain.

**Returns** Number of dimensions of the codomain.

**Return type** int

**dim_domain**
Return number of dimensions of the domain.

**Returns** Number of dimensions of the domain.
Return type int

domain_range
Return the domain range of the object

Returns List of tuples with the ranges for each domain dimension.

dropna()
Return ExtensionArray without NA values.

Returns valid
Return type ExtensionArray
dtype
An instance of 'ExtensionDtype'.
evaluate(eval_points, *, derivative=0, extrapolation=None, grid=False, aligned_evaluation=True, keepdims=None)
Evaluate the object or its derivatives at a list of values or a grid.

Parameters
- eval_points (array_like) – List of points where the functions are evaluated. If a matrix of shape nsample x eval_points is given each sample is evaluated at the values in the corresponding row in eval_points.
- derivative (int, optional) – Order of the derivative. Defaults to 0.
- extrapolation (str or Extrapolation, optional) – Controls the extrapolation mode for elements outside the domain range. By default it is used the mode defined during the instance of the object.
- grid (bool, optional) – Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays. If true the eval_points should be a list of size dim_domain with the corresponding times for each axis. The return matrix has shape n_samples x len(t1) x len(t2) x ... x len(t_dim_domain) x dim_codomain. If the domain dimension is 1 the parameter has no effect. Defaults to False.
- keepdims (bool, optional) – If the image dimension is equal to 1 and keepdims is True the return matrix has shape n_samples x eval_points x 1 else n_samples x eval_points. By default is used the value given during the instance of the object.

Returns Matrix whose rows are the values of the each function at the values specified in eval_points.

Return type (np.ndarray)
e extrapolation
Return default type of extrapolation.
e extrapolator_evaluator
Return the evaluator constructed by the extrapolator.
factorize(na_sentinel: int = -1) → Tuple[numpy.ndarray, pandas.core.dtypes.generic.ABCExtensionArray]
Encode the extension array as an enumerated type.

Parameters na_sentinel (int, default -1) – Value to use in the codes array to indicate missing values.

Returns
- **codes** (*ndarray*) – An integer NumPy array that’s an indexer into the original ExtensionArray.
- **uniques** (*ExtensionArray*) – An ExtensionArray containing the unique values of `self`.

**Note**: uniques will *not* contain an entry for the NA value of the ExtensionArray if there are any missing values present in `self`.

**See also:**

`factorize()` Top-level factorize method that dispatches here.

**Notes**

`pandas.factorize()` offers a *sort* keyword as well.

**fillna**(*value=None, method=None, limit=None*)

Fill NA/NaN values using the specified method.

*Parameters*

- **value** (*scalar, array-like*) – If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like ‘value’ can be given. It’s expected that the array-like have the same length as ‘self’.
- **method** ({'backfill', 'bfill', 'pad', 'ffill', None}, default None) – Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap.
- **limit** (*int, default None*) – If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

*Returns* With NA/NaN filled.

*Return type* ExtensionArray

**isna**()

A 1-D array indicating if each value is missing.

*Returns* Array full of False values.

*Return type* `na_values (np.ndarray)`

**mean**(*weights=None*)

Compute the mean of all the samples.

weights (array-like, optional): List of weights.

*Returns* A FData object with just one sample representing the mean of all the samples in the original object.

*Return type* `FData`

**n_samples**

Return the number of samples.
Returns Number of samples of the FData object.

Return type int

 nbytes
The number of bytes needed to store this object in memory.

 ndim
Return number of dimensions of the functional data. It is always 1, as each observation is con-
sidered a “scalar” object.

 Returns Number of dimensions of the functional data.

 Return type int

 plot(*args, **kwargs)
Plot the FDatGrid object.

 Parameters

 • chart (figure object, axe or list of axes, optional) – figure over with
the graphs are plotted or axis over where the graphs are plotted. If None and
ax is also None, the figure is initialized.

 • derivative (int or tuple, optional) – Order of derivative to be plotted. In
case of surfaces a tuple with the order of derivation in each direction can be passed.
See evaluate() to obtain more information. Defaults 0.

 • fig (figure object, optional) – figure over with the graphs are plotted in case
ax is not specified. If None and ax is also None, the figure is initialized.

 • ax (list of axis objects, optional) – axis over where the graphs are plotted.
If None, see param fig.

 • n_rows (int, optional) – designates the number of rows of the figure to plot the
different dimensions of the image. Only specified if fig and ax are None.

 • n_cols (int, optional) – designates the number of columns of the figure to plot
the different dimensions of the image. Only specified if fig and ax are None.

 • n_points (int or tuple, optional) – Number of points to evaluate in the plot.
In case of surfaces a tuple of length 2 can be passed with the number of points to
plot in each axis, otherwise the same number of points will be used in the two axes.
By default in unidimensional plots will be used 501 points; in surfaces will be used
30 points per axis, which makes a grid with 900 points.

 • domain_range (tuple or list of tuples, optional) – Range where the func-
tion will be plotted. In objects with unidimensional domain the domain range
should be a tuple with the bounds of the interval; in the case of surfaces a list with
2 tuples with the ranges for each dimension. Default uses the domain range of the
functional object.

 • group (list of int) – contains integers from [0 to number of labels) indicating
to which group each sample belongs to. Then, the samples with the same label are
plotted in the same color. If None, the default value, each sample is plotted in the
color assigned by matplotlib.pyplot.rcParams['axes.prop_cycle'].

 • group_colors (list of colors) – colors in which groups are represented, there
must be one for each group. If None, each group is shown with distinct colors in the
“Greys” colormap.
• **group_names** (*list of str*) – name of each of the groups which appear in a legend, there must be one for each one. Defaults to None and the legend is not shown.

• ****kwargs – if dim_domain is 1, keyword arguments to be passed to the matplotlib.pyplot.plot function; if dim_domain is 2, keyword arguments to be passed to the matplotlib.pyplot.plot_surface function.

**Returns** figure object in which the graphs are plotted.

**Return type** fig (figure object)

**ravel**(*order='C'*) → pandas.core.dtypes.generic.ABCExtensionArray

Return a flattened view on this array.

**Parameters**

- **order** (*{None, 'C', 'F', 'A', 'K'}, default 'C'*) –

**Returns**

**Return type** ExtensionArray

**Notes**

- Because ExtensionArrays are 1D-only, this is a no-op.
- The “order” argument is ignored, is for compatibility with NumPy.

**repeat**(*repeats, axis=None*)

Repeat elements of a ExtensionArray.

Returns a new ExtensionArray where each element of the current ExtensionArray is repeated consecutively a given number of times.

**Parameters**

- **repeats** (*int or array of ints*) – The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty ExtensionArray.
- **axis** (*None*) – Must be None. Has no effect but is accepted for compatibility with numpy.

**Returns** repeated_array – Newly created ExtensionArray with repeated elements.

**Return type** ExtensionArray

See also:

- **Series.repeat()** Equivalent function for Series.
- **Index.repeat()** Equivalent function for Index.
- **numpy.repeat()** Similar method for numpy.ndarray.
- **ExtensionArray.take()** Take arbitrary positions.

**Examples**
```python
cat = pd.Categorical(['a', 'b', 'c'])
cat
[a, b, c]
Categories (3, object): [a, b, c]
cat.repeat(2)
[a, a, b, b, c, c]
Categories (3, object): [a, b, c]
cat.repeat([1, 2, 3])
[a, b, b, c, c, c]
Categories (3, object): [a, b, c]
```

**searchsorted** *(value, side='left', sorter=None)*

Find indices where elements should be inserted to maintain order.

New in version 0.24.0.

Find the indices into a sorted array `self` (a) such that, if the corresponding elements in `value` were inserted before the indices, the order of `self` would be preserved.

Assuming that `self` is sorted:

<table>
<thead>
<tr>
<th>side</th>
<th>returned index <code>i</code> satisfies</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td><code>self[i-1] &lt; value &lt;= self[i]</code></td>
</tr>
<tr>
<td>right</td>
<td><code>self[i-1] &lt;= value &lt; self[i]</code></td>
</tr>
</tbody>
</table>

**Parameters**

- `value` *(array_like)* – Values to insert into `self`.
- `side` *({'left', 'right'}, optional)* – If ‘left’, the index of the first suitable location found is given. If ‘right’, return the last such index. If there is no suitable index, return either 0 or N (where N is the length of `self`).
- `sorter` *(1-D array_like, optional)* – Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

**Returns** Array of insertion points with the same shape as `value`.

**Return type** array of ints

See also:

- `numpy.searchsorted()` Similar method from NumPy.

**shape**

Return a tuple of the array dimensions.

**shift** *(shifts, *, restrict_domain=False, extrapolation=None, discretization_points=None, **kwargs)*

Perform a shift of the curves.

**Parameters**

- `shifts` *(array_like or numeric)* – List with the shift corresponding for each sample or numeric with the shift to apply to all samples.
- `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.

**discretization_points** *(array_like, optional)* – Set of points where the functions are evaluated to obtain the discrete representation of the object to operate. If an empty list is passed it calls np.linspace with bounds equal to the ones defined in *fd.domain_range* and the number of points the maximum between 201 and 10 times the number of basis plus 1.

Returns *FData* with the shifted functional data.

**take** *(indices, allow_fill=False, fill_value=None, axis=0)*

Take elements from an array.

Parameters

- **indices** *(sequence of integers)* – Indices to be taken.

- **allow_fill** *(bool, default False)* – How to handle negative values in *indices*.
  - False: negative values in *indices* indicate positional indices from the right (the default). This is similar to *numpy.take()*.
  - True: negative values in *indices* indicate missing values. These values are set to *fill_value*. Any other negative values raise a *ValueError*.

- **fill_value** *(any, optional)* – Fill value to use for NA-indices when *allow_fill* is True. This may be *None*, in which case the default NA value for the type, *self.dtype.na_value*, is used. For many ExtensionArrays, there will be two representations of *fill_value*: a user-facing “boxed” scalar, and a low-level physical NA value. *fill_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

Returns *FData*

Raises

- **IndexError** – When the indices are out of bounds for the array.

- **ValueError** – When *indices* contains negative values other than -1 and *allow_fill* is True.

**Notes**

ExtensionArray.take is called by *Series.__getitem__, .loc, iloc*, when *indices* is a sequence of values. Additionally, it’s called by *Series.reindex()*, or any other method that causes realignment, with a *fill_value*

See also:

*numpy.take*  *pandas.api.extensions.take*

**to_basis** *(basis, eval_points=None, **kwargs)*

Return the basis representation of the object.

Parameters

- **basis** *(Basis)* – basis object in which the functional data are going to be represented.

- ****kwargs – keyword arguments to be passed to *FDataBasis.from_data()*.
Returns Basis representation of the functional data object.

Return type `FDataBasis`

to_grid(eval_points=None)
Return the discrete representation of the object.

Parameters eval_points (array_like, optional) – Set of points where the functions are evaluated.

Returns Discrete representation of the functional data object.

Return type `FDataGrid`

to_numpy()
Returns a numpy array with the objects

unique()
Compute the ExtensionArray of unique values.

Returns uniques

Return type ExtensionArray

view(dtype=None) → Union[pandas.core.dtypes.generic.ABCExtensionArray, numpy.ndarray]
Return a view on the array.

Parameters dtype (str, np.dtype, or ExtensionDtype, optional) – Default None.

Returns A view of the ExtensionArray.

Return type ExtensionArray

1.1.4 Extrapolation

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

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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BoundaryExtrapolation</code></td>
<td>Extends the domain range using the boundary values.</td>
</tr>
<tr>
<td><code>ExceptionExtrapolation</code></td>
<td>Raise and exception.</td>
</tr>
<tr>
<td><code>FillExtrapolation</code></td>
<td>Values outside the domain range will be filled with a fixed value.</td>
</tr>
<tr>
<td><code>PeriodicExtrapolation</code></td>
<td>Extends the domain range periodically.</td>
</tr>
</tbody>
</table>

1.1. Representation of functional Data
BoundaryExtrapolation

class skfda.representation.extrapolation.BoundaryExtrapolation

Extends the domain range using the boundary values.

Examples

```python
>>> 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

```python
>>> fd.extrapolation = BoundaryExtrapolation()
>>> fd([-0.5, 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”.

```python
>>> fd.extrapolation = 'bounds'
>>> fd([-0.5, 0, 1.5]).round(3)
array([[ 0.976,  0.976,  0.797],
       [ 0.759,  0.759,  1.125]])
```

Methods

evaluator(fdata) Returns the evaluator used by FData.

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

evaluator(fdata) Returns the evaluator used by FData.

Returns

Evaluator of the periodic boundary.

Return type (Evaluator)

ExceptionExtrapolation

class skfda.representation.extrapolation.ExceptionExtrapolation

Raise and exception.

Examples

```python
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import (continues on next page)
```
ExceptionExtrapolation)

```python
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```python
>>> fd.extrapolation = ExceptionExtrapolation()
>>> try:
...    fd([-0.5, 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".

```python
>>> fd.extrapolation = 'exception'
>>> try:
...    fd([-0.5, 0, 1.5]).round(3)
... except ValueError as e:
...    print(e)
Attempt to evaluate 2 points outside the domain range.
```

Methods

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

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

evaluator(fdata)  Returns the evaluator used by FData.

   Returns  Evaluator of the periodic extrapolation.

   Return type  (Evaluator)
```

FillExtrapolation

class skfda.representation.extrapolation.FillExtrapolation(fill_value)

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

Examples

```python
>>> 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

```python
>>> fd.extrapolation = FillExtrapolation(0)
>>> fd([-0.5, 0, 1.5]).round(3)
```
The previous extrapolator is equivalent to the string “zeros”. In the same way FillExtrapolation(np.nan) is equivalent to “nan”.

```python
>>> fd.extrapolation = "nan"
>>> fd([-0.5, 0, 1.5]).round(3)
array([[ nan, 0.976, nan],
      [ nan, 0.759, nan]])
```

### Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(fill_value)</code></td>
<td>Returns the evaluator used by FData.</td>
</tr>
<tr>
<td><code>evaluator(fdata)</code></td>
<td>Construct an evaluator.</td>
</tr>
</tbody>
</table>

- **`__init__(fill_value)`**
  - Returns the evaluator used by FData.
  - **Returns** Evaluator of the periodic extrapolation.
  - **Return type** (Evaluator)

### Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fill_value</code></td>
<td>Returns the fill value of the extrapolation</td>
</tr>
</tbody>
</table>

- **`evaluator(fdata)`**
  - Construct an evaluator.
  - Builds the evaluator from an functional data object.
  - **Parameters** fdata (FData) – Functional object where the evaluator will be used.
  - **Returns** Evaluator of the fdata.
  - **Return type** (Evaluator)

- **`fill_value`**
  - Returns the fill value of the extrapolation

#### PeriodicExtrapolation

**class** `skfda.representation.extrapolation.PeriodicExtrapolation`

- 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([-0.5, 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([-0.5, 0, 1.5]).round(3)
array([[-0.724, 0.976, -0.724],
     [-1.086, 0.759, -1.086]])

Methods

evaluator(fdata) Returns the evaluator used by FData.

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

evaluator(fdata) Returns the evaluator used by FData.

Returns Evaluator of the periodic extrapolation.

Return type (Evaluator)

Custom Extrapolation

Custom extrapolators could be done subclassing EvaluatorConstructor.

kfda.representation.evaluator. Constructor of an evaluator.
EvaluatorConstructor
ekfda.representation.evaluator.Evaluator Structure of an evaluator.

EvaluatorConstructor

class kfda.representation.evaluator.EvaluatorConstructor
Constructor of an evaluator.

A constructor builds an Evaluator from a FData, which is used to the evaluation in the functional data object.

The evaluator constructor should have a method evaluator() which receives an fdata object and returns an Evaluator.
Methods

---

evaluator(fdata) Construct an evaluator.

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

evaluator(fdata) Construct an evaluator.

Builds the evaluator from an functional data object.

Parameters

- **fdata** (FData) – Functional object where the evaluator will be used.

Returns

Evaluator of the fdata.

Return type (Evaluator)

Evaluator

class skfda.representation.evaluator.Evaluator

Structure of an evaluator.

An evaluator defines how to evaluate points of a functional object, it can be used as extrapolator to evaluate points outside the domain range or as interpolator in a FDataGrid. The corresponding examples of Interpolation and Extrapolation shows the basic usage of this class.

The evaluator is called internally by **evaluate()**.

Should implement the methods **evaluate()** and **evaluate_composed()**.

Methods

---

evaluate(eval_points, *[, derivative]) Evaluation method.

evaluate_composed(eval_points, *[, derivative]) Evaluation method.

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

evaluate(eval_points, *, derivative=0) Evaluation method.

Evaluates the samples at the same evaluation points. The evaluation call will receive a 2-d array with the evaluation points. This method is called internally by **evaluate()** when the argument **aligned_evaluation** is True.

Parameters

- **eval_points** (*numpy.ndarray*) – Numpy array with shape (number_eval_points, dim_domain) with the evaluation points.

- **derivative** (*int, optional*) – Order of the derivative. Defaults to 0.

Returns

Numpy 3d array with shape (n_samples, number_eval_points,
dim_codomain) with the result of the evaluation. The entry (i,j,k) will contain the value k-th image dimension of the i-th sample, at the j-th evaluation point.

Return type (numpy.darray)

`evaluate_composed`:

Evaluates the samples at different evaluation points. The evaluation call will receive a 3-d array with the evaluation points for each sample. This method is called internally by `evaluate()` when the argument `aligned_evaluation` is False.

Parameters

- `eval_points (numpy.ndarray)` – Numpy array with shape `(n_samples, number_eval_points, dim_domain)` with the evaluation points for each sample.
- `derivative (int, optional)` – Order of the derivative. Defaults to 0.

Returns

Numpy 3d array with shape `(n_samples, number_eval_points, dim_codomain)` with the result of the evaluation. The entry (i,j,k) will contain the value k-th image dimension of the i-th sample, at the j-th evaluation point.

Return type (numpy.darray)

1.2 Preprocessing

Sometimes we need to preprocess the data prior to analyze it. The modules in this category deal with this problem.

1.2.1 Smoothing

Sometimes the functional observations are noisy. The noise can be reduced by smoothing the data.

This module provide several classes, called smoothers, that perform a smoothing transformation of the data. All of the smoothers follow the API of an scikit-learn transformer object.

The degree of smoothing is controlled in all smoothers by an smoothing parameter, named `smoothing_parameter`, that has different meaning for each smoother.

Kernel smoothers

Kernel smoothing methods compute the smoothed value at a point by considering the influence of each input point over it. For doing this, it considers a kernel function placed at the desired point. The influence of each input point will be related with the value of the kernel function at that input point.

There are several kernel smoothers provided in this library. All of them are also linear smoothers, meaning that they compute a smoothing matrix (or hat matrix) that performs the smoothing as a linear transformation.
skfda.preprocessing.smoothing.
kernel_smoothers.NadarayaWatsonSmoother(*)

Nadaraya-Watson smoothing method.

skfda.preprocessing.
smoothing.kernel_smoothers.
LocalLinearRegressionSmoother(*)

Local linear regression smoothing method.

skfda.preprocessing.smoothing.
kernel_smoothers.KNeighborsSmoother(*)

K-nearest neighbour kernel smoother.

NadarayaWatsonSmoother

class skfda.preprocessing.smoothing.kernel_smoothers.NadarayaWatsonSmoother(*, smoothing_parameter=None, kernel=<function normal>, weights=None, output_points=None)

Nadaraya-Watson smoothing method.

It is a linear kernel smoothing method. Uses an smoothing matrix \( \hat{H} \) for the discretisation points in \( \text{argvals} \) by the Nadaraya-Watson estimator. The smoothed values \( \hat{Y} \) can be calculated as \( \hat{Y} = \hat{H}Y \) where \( Y \) is the vector of observations at the points of discretisation \((x_1, x_2, ..., x_n)\).

\[
\hat{H}_{i,j} = \frac{K \left( \frac{x_i - x_j}{h} \right)}{\sum_{k=1}^{n} K \left( \frac{x_i - x_k}{h} \right)}
\]

where \( K(\cdot) \) is a kernel function and \( h \) the kernel window width or smoothing parameter.

Parameters

- **argvals** (`ndarray`) – Vector of discretisation points.
- **smoothing_parameter** (`float`, `optional`) – Window width of the kernel (also called \( h \) or bandwidth).
- **kernel** (`function`, `optional`) – kernel function. By default a normal kernel.
- **weights** (`ndarray`, `optional`) – Case weights matrix (in order to modify the importance of each point).
- **output_points** (`ndarray`, `optional`) – The output points. If omitted, the input points are used.

Examples

```python
>>> from skfda import FDataGrid
>>> fd = FDataGrid(sample_points=[1, 2, 4, 5, 7],
...                   data_matrix=[[1, 2, 3, 4, 5]])
>>> smoother = NadarayaWatsonSmoother(smoothing_parameter=3.5)
```
```python
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 2.42,
  2.61],
 [ 3.03],
 [ 3.24],
 [ 3.65]])

>>> smoother.hat_matrix().round(3)
array([[ 0.294, 0.282, 0.204, 0.153, 0.068],
 [ 0.249, 0.259, 0.22 , 0.179, 0.093],
 [ 0.165, 0.202, 0.238, 0.229, 0.165],
 [ 0.129, 0.172, 0.239, 0.249, 0.211],
 [ 0.073, 0.115, 0.221, 0.271, 0.319]])

>>> smoother = NadarayaWatsonSmoother(smoothing_parameter=2)
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.84,
  2.18],
 [ 3.09],
 [ 3.55],
 [ 4.28]])

>>> smoother.hat_matrix().round(3)
array([[ 0.425, 0.375, 0.138, 0.058, 0.005],
 [ 0.309, 0.35 , 0.212, 0.114, 0.015],
 [ 0.103, 0.193, 0.319, 0.281, 0.103],
 [ 0.046, 0.11 , 0.299, 0.339, 0.206],
 [ 0.006, 0.022, 0.163, 0.305, 0.503]])
```

The output points can be changed:

```python
>>> smoother = NadarayaWatsonSmoother(
...     smoothing_parameter=2,
...     output_points=[1, 2, 3, 4, 5, 6, 7])
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.84],
 [ 2.18],
 [ 2.61],
 [ 3.09],
 [ 3.55],
 [ 4.28]])

>>> smoother.hat_matrix().round(3)
array([[ 0.425, 0.375, 0.138, 0.058, 0.005],
 [ 0.309, 0.35 , 0.212, 0.114, 0.015],
 [ 0.103, 0.193, 0.319, 0.281, 0.103],
 [ 0.046, 0.11 , 0.299, 0.339, 0.206],
 [ 0.006, 0.022, 0.163, 0.305, 0.503]])
```
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(*[, smoothing_parameter, kernel, ...])</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>fit(X, y)</td>
<td>Compute the hat matrix for the desired output points.</td>
</tr>
<tr>
<td>fit_transform(X, y)</td>
<td>Fit to data, then transform it.</td>
</tr>
<tr>
<td>get_params(deep)</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td>hat_matrix([input_points, output_points])</td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td>score(X, y)</td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td>set_params(**params)</td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td>transform(X[, y])</td>
<td>Multiplies the hat matrix for the functions values to smooth them.</td>
</tr>
</tbody>
</table>

**__init__(*, smoothing_parameter=None, kernel=<function normal>, weights=None, output_points=None)**

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

**fit(X: skfda.representation.grid.FDataGrid, y=None)**

Compute the hat matrix for the desired output points.

### Parameters

- **X** (*FDataGrid*) – The data whose points are used to compute the matrix.
- **y** – Ignored

### Returns

self (object)

**fit_transform(X, y=None, **fit_params)**

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

### Parameters

- **X** (*numpy array of shape [n_samples, n_features]*) – Training set.
- **y** (*numpy array of shape [n_samples]*) – Target values.
- **fit_params (dict)** – Additional fit parameters.

### Returns

X_new – Transformed array.

### Return type

numpy array of shape [n_samples, n_features_new]

**get_params(deep=True)**

Get parameters for this estimator.

### Parameters

- **deep (bool, default=True)** – If True, will return the parameters for this estimator and contained subobjects that are estimators.

### Returns

params – Parameter names mapped to their values.

### Return type

mapping of string to any

**score(X, y)**

Returns the generalized cross validation (GCV) score.

### Parameters

- **X** (*FDataGrid*) – The data to smooth.
• $y$ (FDataGrid) – The target data. Typically the same as $X$.

**Returns** Generalized cross validation score.

**Return type** float

**set_params(**params**)**

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters** **params** (dict) – Estimator parameters.

**Returns** self – Estimator instance.

**Return type** object

**transform**(X: skfda.representation.grid.FDataGrid, y=None)

Multiplies the hat matrix for the functions values to smooth them.

**Parameters**

• $X$ (FDataGrid) – The data to smooth.

• $y$ – Ignored

**Returns** Functional data smoothed.

**Return type** FDataGrid

**Examples using skfda.preprocessing.smoothing.kernel_smoothers.NadarayaWatsonSmother**

• *Kernel Smoothing*

**LocalLinearRegressionSmother**

class skfda.preprocessing.smoothing.kernel_smoothers.LocalLinearRegressionSmother(*,
smoothing_parameter=None, kernel=<function normal>, weights=None, output_points=None)

Local linear regression smoothing method.

It is a linear kernel smoothing method. Uses an smoothing matrix $\hat{H}$ for the discretisation points in argvals by the local linear regression estimator. The smoothed values $\hat{Y}$ can be calculated as $\hat{Y} = \hat{H}Y$ where $Y$ is the vector of observations at the points of discretisation $(x_1, x_2, ..., x_n)$.

$$\hat{H}_{i,j} = \frac{b_i(x_j)}{\sum_{k=1}^{n} b_k(x_j)}$$
\[ b_i(x) = K \left( \frac{x_i - x}{h} \right) S_{n,2}(x) - (x_i - x)S_{n,1}(x) \]

\[ S_{n,k} = \sum_{i=1}^{n} K \left( \frac{x_i - x}{h} \right)^k \]

where \( K(\cdot) \) is a kernel function and \( h \) the kernel window width.

**Parameters**

- **argvals (ndarray)** – Vector of discretisation points.
- **smoothing_parameter (float, optional)** – Window width of the kernel (also called \( h \) or bandwidth).
- **kernel (function, optional)** – Kernel function. By default a normal kernel.
- **weights (ndarray, optional)** – Case weights matrix (in order to modify the importance of each point).
- **output_points (ndarray, optional)** – The output points. If omitted, the input points are used.

**Examples**

```python
>>> from skfda import FDataGrid
>>> fd = FDataGrid(sample_points=[1, 2, 4, 5, 7],
...                  data_matrix=[[1, 2, 3, 4, 5]])
>>> smoother = LocalLinearRegressionSmooother(smoothing_parameter=3.5)
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.13,
       1.36],
       [ 3.29,
       4.27],
       [ 5.08]])

>>> smoother.hat_matrix().round(3)
array([[ 0.614, 0.429, 0.077, -0.03 , -0.09 ],
       [ 0.381, 0.595, 0.168, -0. , -0.143],
       [-0.104, 0.112, 0.697, 0.398, -0.104],
       [-0.147, -0.036, 0.392, 0.639, 0.152],
       [-0.095, -0.079, 0.117, 0.308, 0.75 ]])
```

```python
>>> smoother = LocalLinearRegressionSmooother(smoothing_parameter=2)
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.11,
       1.41],
       [ 3.31,
       4.04],
       [ 5.04]])

>>> smoother.hat_matrix().round(3)
array([[ 0.714, 0.386, -0.037, -0.053, -0.01 ],
       [ 0.352, 0.724, 0.045, -0.081, -0.04 ],
       [-0.078, 0.052, 0.74 , 0.364, -0.078],
       [-0.07 , -0.067, 0.36 , 0.716, 0.061],
       [-0.012, -0.032, -0.025, 0.154, 0.915]])
```
The output points can be changed:

```python
>>> smoother = LocalLinearRegressionSmoother(
...     smoothing_parameter=2,
...     output_points=[1, 2, 3, 4, 5, 6, 7])
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.11],
       [ 1.41],
       [ 1.81],
       [ 3.31],
       [ 4.04],
       [ 5.35],
       [ 5.04]])
>>> smoother.hat_matrix().round(3)
array([[ 0.714, 0.386, -0.037, -0.053, -0.01 ],
       [ 0.352, 0.724, 0.045, -0.081, -0.04 ],
       [-0.084, 0.722, 0.722, -0.084, -0.278],
       [-0.078, 0.052, 0.74 , 0.364, -0.078],
       [-0.07 , -0.067, 0.36 , 0.716, 0.061],
       [-0.098, -0.202, -0.003, 0.651, 0.651],
       [-0.012, -0.032, -0.025, 0.154, 0.915]])
```

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(*, smoothing_parameter, kernel, ...)</td>
<td>Initialize self.</td>
</tr>
<tr>
<td><code>fit(X, y)</code></td>
<td>Compute the hat matrix for the desired output points.</td>
</tr>
<tr>
<td><code>fit_transform(X, y)</code></td>
<td>Fit to data, then transform it.</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>hat_matrix([input_points, output_points])</code></td>
<td></td>
</tr>
<tr>
<td><code>score(X, y)</code></td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>transform(X, y)</code></td>
<td>Multiplies the hat matrix for the functions values to smooth them.</td>
</tr>
</tbody>
</table>

**Private Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(*, smoothing_parameter=None, kernel=&lt;function normal&gt;, weights=None, output_points=None)</td>
<td>Initialize self. See help(type(self)) for accurate signature.</td>
</tr>
</tbody>
</table>

**fit(X: skfda.representation.grid.FDataGrid, y=None)**

Compute the hat matrix for the desired output points.

**Parameters**

- **X (FDataGrid)** – The data whose points are used to compute the matrix.
- **y** – Ignored

**Returns** self (object)

**fit_transform(X, y=None, **fit_params)**

Fit to data, then transform it.
Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- X (numpy array of shape [n_samples, n_features]) – Training set.
- y (numpy array of shape [n_samples]) – Target values.
- **fit_params (dict) – Additional fit parameters.

Returns X_new – Transformed array.

Return type numpy array of shape [n_samples, n_features_new]

get_params(deep=True)

Get parameters for this estimator.

Parameters deep (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type mapping of string to any

score(X, y)

Returns the generalized cross validation (GCV) score.

Parameters

- X (FDataGrid) – The data to smooth.
- y (FDataGrid) – The target data. Typically the same as X.

Returns Generalized cross validation score.

Return type float

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>_<parameter> so that it’s possible to update each component of a nested object.

Parameters **params (dict) – Estimator parameters.

Returns self – Estimator instance.

Return type object

transform(X: skfda.representation.grid.FDataGrid, y=None)

Multiplies the hat matrix for the functions values to smooth them.

Parameters

- X (FDataGrid) – The data to smooth.
- y – Ignored

Returns Functional data smoothed.

Return type FDataGrid
Examples using skfda.preprocessing.smoothing.kernel_smoothers.LocalLinearRegressionSmoother

- Kernel Smoothing

KNeighborsSmoother

class skfda.preprocessing.smoothing.kernel_smoothers.KNeighborsSmoother(*, smoothing_parameter=None, kernel=<function uniform>, weights=None, output_points=None)

K-nearest neighbour kernel smoother.

It is a linear kernel smoothing method. Uses an smoothing matrix $S$ for the discretisation points in argvals by the $k$ nearest neighbours estimator.

Usually used with the uniform kernel, it takes the average of the closest $k$ points to a given point.

Parameters

- **argvals** (*ndarray*) – Vector of discretisation points.
- **smoothing_parameter** (*int, optional*) – Number of nearest neighbours. By default it takes the 5% closest points.
- **kernel** (*function, optional*) – kernel function. By default a uniform kernel to perform a ‘usual’ $k$ nearest neighbours estimation.
- **weights** (*ndarray, optional*) – Case weights matrix (in order to modify the importance of each point).
- **output_points** (*ndarray, optional*) – The output points. If ommited, the input points are used.

Examples

```python
>>> from skfda import FDataGrid
>>> fd = FDataGrid(sample_points=[1, 2, 4, 5, 7],
...                 data_matrix=[[1, 2, 3, 4, 5]])
>>> smoother = KNeighborsSmoother(smoothing_parameter=2)
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.5],
       [ 1.5],
       [ 3.5],
       [ 3.5],
       [ 4.5]])
```

```python
>>> smoother.hat_matrix().round(3)
array([[ 0.5, 0.5, 0., 0., 0.],
       [ 0.5, 0.5, 0., 0., 0.],
       [ 0., 0., 0.5, 0.5, 0.],
       [ 0., 0., 0.5, 0.5, 0.],
       [ 0., 0., 0.5, 0.5, 0.]],
       (continues on next page)
```
In case there are two points at the same distance it will take both.

```python
>>> fd = FDataGrid(sample_points=[1, 2, 3, 5, 7],
...                  data_matrix=[[1, 2, 3, 4, 5]])
>>> smoother = KNeighborsSmoother(smoothing_parameter=2)
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.5],
       [ 2. ],
       [ 2.5],
       [ 4. ],
       [ 4.5]])
>>> smoother.hat_matrix().round(3)
array([[ 0.5 , 0.5 , 0. , 0. , 0. ],
       [ 0.333, 0.333, 0.333, 0. , 0. ],
       [ 0. , 0.5 , 0.5 , 0. , 0. ],
       [ 0. , 0. , 0.333, 0.333, 0.333],
       [ 0. , 0. , 0. , 0.5 , 0.5 ]])
```

The output points can be changed:

```python
>>> smoother = KNeighborsSmoother(
...                   smoothing_parameter=2,
...                   output_points=[1, 2, 3, 4, 5, 6, 7])
>>> fd_smoothed = smoother.fit_transform(fd)
>>> fd_smoothed.data_matrix.round(2)
array([[ 1.5],
       [ 2. ],
       [ 2.5],
       [ 3.5],
       [ 4. ],
       [ 4.5]])
>>> smoother.hat_matrix().round(3)
array([[ 0.5 , 0.5 , 0. , 0. , 0. ],
       [ 0.333, 0.333, 0.333, 0. , 0. ],
       [ 0. , 0.5 , 0.5 , 0. , 0. ],
       [ 0. , 0. , 0.333, 0.333, 0.333],
       [ 0. , 0. , 0. , 0.5 , 0.5 ],
       [ 0. , 0. , 0. , 0.5 , 0.5 ]])
```

### Methods

```python
__init__(*, smoothing_parameter, kernel, ...)
```
Initialize self.

(continued from previous page)
Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit(X[, y])</code></td>
<td>Compute the hat matrix for the desired output points.</td>
</tr>
<tr>
<td><code>fit_transform(X[, y])</code></td>
<td>Fit to data, then transform it.</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>hat_matrix([input_points, output_points])</code></td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td><code>score(X, y)</code></td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>transform(X[, y])</code></td>
<td>Multiplies the hat matrix for the functions values to smooth them.</td>
</tr>
</tbody>
</table>

```python
__init__(*, smoothing_parameter=None, kernel=<function uniform>, weights=None, output_points=None)
```

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

```python
fit(X: skfda.representation.grid.FDataGrid, y=None)
```

Compute the hat matrix for the desired output points.

**Parameters**

- `X` ([FDataGrid]) – The data whose points are used to compute the matrix.
- `y` – Ignored

**Returns** `self` (object)

```python
fit_transform(X, y=None, **fit_params)
```

Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

**Parameters**

- `X` ([numpy array of shape [n_samples, n_features]]) – Training set.
- `y` ([numpy array of shape [n_samples]]) – Target values.
- `**fit_params` ([dict]) – Additional fit parameters.

**Returns** `X_new` – Transformed array.

**Return type** numpy array of shape [n_samples, n_features_new]

```python
get_params(deep=True)
```

Get parameters for this estimator.

**Parameters**

- `deep` ([bool], `default=True`) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** `params` – Parameter names mapped to their values.

**Return type** mapping of string to any

```python
score(X, y)
```

Returns the generalized cross validation (GCV) score.

**Parameters**

- `X` ([FDataGrid]) – The data to smooth.
- `y` ([FDataGrid]) – The target data. Typically the same as `X`.

**Returns** Generalized cross validation score.
Return type  float

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters  **params (dict) – Estimator parameters.

Returns  self – Estimator instance.

Return type  object

transform(X: skfda.representation.grid.FDataGrid, y=None)

Multiplies the hat matrix for the functions values to smooth them.

Parameters

- X (FDataGrid) – The data to smooth.
- y – Ignored

Returns  Functional data smoothed.

Return type  FDataGrid

Examples using skfda.preprocessing.smoothing.kernel_smoothers.KNeighborsSmoother

- *Kernel Smoothing*

**Basis smoother**

The basis smoother smooths the data by means of expressing it in a truncated basis expansion. The data can be further smoothed penalizing its derivatives, using a linear differential operator. This has the effect of reducing the curvature of the function and/or its derivatives.

This smoother is also a linear smoother, although if the QR or Cholesky methods are used, the matrix does not need to be explicitly computed.

```
skfda.preprocessing.smoothing.BasisSmoother(...)  
```

BasisSmoother

```
class skfda.preprocessing.smoothing.BasisSmoother(basis, *, smoothing_parameter: float = 0, weights=None, penalty: Union[int, Iterable[float], LinearDifferentialOperator] = None, penalty_matrix=None, output_points=None, method='cholesky', return_basis=False)  
```

Transform raw data to a smooth functional form.

Transform raw data to a smooth functional form.

Takes functional data in a discrete form and makes an approximates it to the closest function that can be generated by the basis.a.
The fit is made so as to reduce the penalized sum of squared errors \([RS05-5-2-6]\):

\[
PENSSSE(c) = (y - \Phi c)'W(y - \Phi c) + \lambda c'Rc
\]

where \(y\) is the vector or matrix of observations, \(\Phi\) the matrix whose columns are the basis functions evaluated at the sampling points, \(c\) the coefficient vector or matrix to be estimated, \(\lambda\) a smoothness parameter and \(c'Rc\) the matrix representation of the roughness penalty \(\int [L(x(s))]^2 ds\) where \(L\) is a linear differential operator.

Each element of \(R\) has the following close form:

\[
R_{ij} = \int L\phi_i(s)L\phi_j(s)ds
\]

By deriving the first formula we obtain the closed formed of the estimated coefficients matrix:

\[
\hat{c} = (\Phi'W\Phi + \lambda R)^{-1}\Phi'Wy
\]

The solution of this matrix equation is done using the cholesky method for the resolution of a LS problem. If this method throughs a rounding error warning you may want to use the QR factorisation that is more numerically stable despite being more expensive to compute. \([RS05-5-2-8]\)

**Parameters**

- **basis** – (Basis): Basis used.
- **weights** *(array_like, optional)* – Matrix to weight the observations. Defaults to the identity matrix.
- **smoothing_parameter** *(int or float, optional)* – Smoothing parameter. Trying with several factors in a logarithm scale is suggested. If 0 no smoothing is performed. Defaults to 0.
- **penalty** *(int, iterable or LinearDifferentialOperator)* – If it is an integer, it indicates the order of the derivative used in the computing of the penalty matrix. For instance 2 means that the differential operator is \(f''(x)\). If it is an iterable, it consists on coefficients representing the differential operator used in the computing of the penalty matrix. For instance the tuple \((1, 0, \text{numpy.sin})\) means \(1 + \sin(x)D^2\). It is possible to supply directly the LinearDifferentialOperator object. If not supplied this defaults to 2. Only used if penalty_matrix is \(\text{None}\).
- **penalty_matrix** *(array_like, optional)* – Penalty matrix. If supplied the differential operator is not used and instead the matrix supplied by this argument is used.
- **method** *(str)* – Algorithm used for calculating the coefficients using the least squares method. The values admitted are ‘cholesky’, ‘qr’ and ‘matrix’ for Cholesky and QR factorisation methods, and matrix inversion respectively. The default is ‘cholesky’.
- **output_points** *(ndarray, optional)* – The output points. If omitted, the input points are used. If return_basis is True, this parameter is ignored.
- **return_basis** *(boolean)* – If False (the default) returns the smoothed data as an FDataGrid, like the other smoothers. If True returns a FDataBasis object.

**Examples**

By default, this smoother returns a FDataGrid, like the other smoothers:
```python
>>> import numpy as np
>>> import skfda

>>> t = np.linspace(0, 1, 5)
>>> x = np.sin(2 * np.pi * t) + np.cos(2 * np.pi * t)
>>> x
array([ 1., 1., -1., -1., 1.])

>>> fd = skfda.FDataGrid(data_matrix=x, sample_points=t)
>>> basis = skfda.representation.basis.Fourier((0, 1), n_basis=3)
>>> smoother = skfda.preprocessing.smoothing.BasisSmoother(basis, method='cholesky')
>>> fd_smooth = smoother.fit_transform(fd)
>>> fd_smooth.data_matrix.round(2)
array([[ 1.,
        1.,
       -1.,
       -1.,
        1.]], dtype=float32)

However, the parameter `return_basis` can be used to return the data in basis form, by default, without extra smoothing:

```python
def = skfda.FDataGrid(data_matrix=x, sample_points=t)
basis = skfda.representation.basis.Fourier((0, 1), n_basis=3)
smoother = skfda.preprocessing.smoothing.BasisSmoother(basis, method='cholesky', return_basis=True)
fd_basis = smoother.fit_transform(fd)
fd_basis.coefficients.round(2)
array([[ 0. , 0.71, 0.71]])
```
```python
>>> fd = skfda.FDataGrid(data_matrix=x, sample_points=t)
>>> basis = skfda.representation.basis.Fourier((0, 1), n_basis=3)
>>> smoother = skfda.preprocessing.smoothing.BasisSmoother(
...     basis, method='cholesky',
...     smoothing_parameter=1,
...     penalty=LinearDifferentialOperator(weights=[3, 5]),
...     return_basis=True)
>>> fd_basis = smoother.fit_transform(fd)
>>> fd_basis.coefficients.round(2)
array([[ 0.18,  0.07,  0.09]])
```

```python
>>> from skfda.misc import LinearDifferentialOperator
>>> fd = skfda.FDataGrid(data_matrix=x, sample_points=t)
>>> basis = skfda.representation.basis.Fourier((0, 1), n_basis=3)
>>> smoother = skfda.preprocessing.smoothing.BasisSmoother(
...     basis, method='qr',
...     smoothing_parameter=1,
...     penalty=LinearDifferentialOperator(weights=[3, 5]),
...     return_basis=True)
>>> fd_basis = smoother.fit_transform(fd)
>>> fd_basis.coefficients.round(2)
array([[ 0.18,  0.07,  0.09]])
```

```python
>>> from skfda.misc import LinearDifferentialOperator
>>> fd = skfda.FDataGrid(data_matrix=x, sample_points=t)
>>> basis = skfda.representation.basis.Fourier((0, 1), n_basis=3)
>>> smoother = skfda.preprocessing.smoothing.BasisSmoother(
...     basis, method='matrix',
...     smoothing_parameter=1,
...     penalty=LinearDifferentialOperator(weights=[3, 5]),
...     return_basis=True)
>>> fd_basis = smoother.fit_transform(fd)
>>> fd_basis.coefficients.round(2)
array([[ 0.18,  0.07,  0.09]])
```

References

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(basis, *, smoothing_parameter, ...)</td>
<td>Initialize self.</td>
</tr>
<tr>
<td><code>fit(X[, y])</code></td>
<td>Compute the hat matrix for the desired output points.</td>
</tr>
<tr>
<td><code>fit_transform(X[, y])</code></td>
<td>Compute the hat matrix for the desired output points.</td>
</tr>
<tr>
<td><code>get_params</code>(deep)</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>hat_matrix</code>(input_points, output_points)</td>
<td>Returns the generalized cross validation (GCV) score.</td>
</tr>
<tr>
<td><code>score</code>(X, y)</td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 36 – continued from previous page

transform(X, y) Apply the smoothing.

__init__(basis, *, smoothing_parameter: float = 0, weights=None, penalty: Union[int, Iterable[float], LinearDifferentialOperator] = None, penalty_matrix=None, output_points=None, method='cholesky', return_basis=False)
Initialize self. See help(type(self)) for accurate signature.

class SolverMethod
An enumeration.

fit(X: skfda.representation.grid.FDataGrid, y=None)
Compute the hat matrix for the desired output points.

Parameters
• X (FDataGrid) – The data whose points are used to compute the matrix.
• y – Ignored

Returns self (object)

fit_transform(X: skfda.representation.grid.FDataGrid, y=None)
Compute the hat matrix for the desired output points.

Parameters
• X (FDataGrid) – The data whose points are used to compute the matrix.
• y – Ignored

Returns self (object)

get_params(deep=True)
Get parameters for this estimator.

Parameters
depth (bool, default=True) – If True, will return the parameters for this
 estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type mapping of string to any

score(X, y)
Returns the generalized cross validation (GCV) score.

Parameters
• X (FDataGrid) – The data to smooth.
• y (FDataGrid) – The target data. Typically the same as X.

Returns Generalized cross validation score.

Return type float

set_params(**params)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

Parameters **params (dict) – Estimator parameters.

Returns self – Estimator instance.
Return type object

transform(X: skfda.representation.grid.FDataGrid, y=None)

Apply the smoothing.

Parameters
- X (FDataGrid) – The data to smooth.
- y – Ignored

Returns self (object)

Validation

It is necessary to measure how good is the smoothing to prevent undersmoothing and oversmoothing. The following classes follow the scikit-learn API for a scorer object, and measure how good is the smoothing. In both of them, the target object y should also be the original data. These scorers need that the smoother is linear, as they use internally the hat matrix.

<table>
<thead>
<tr>
<th>sklearn.preprocessing.smoothing.validation.LinearSmootherLeaveOneOutScorer</th>
<th>Leave-one-out cross validation scoring method for linear smoothers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sklearn.preprocessing.smoothing.validation.LinearSmootherGeneralizedCVScorer(...)</td>
<td>Generalized cross validation scoring method for linear smoothers.</td>
</tr>
</tbody>
</table>

LinearSmootherLeaveOneOutScorer

class sklearn.preprocessing.smoothing.validation.LinearSmootherLeaveOneOutScorer

Leave-one-out cross validation scoring method for linear smoothers.

It calculates the cross validation score for every sample in a FDataGrid object given a linear smoother with a smoothing matrix $H^\nu$ calculated with a parameter $\nu$:

$$CV(\nu) = \frac{1}{n} \sum_i \left( y_i - \hat{y}_i^{\nu(-i)} \right)^2$$

Where $\hat{y}_i^{\nu(-i)}$ is the adjusted $y_i$ when the the pair of values $(x_i, y_i)$ are excluded in the smoothing. This would require to recalculate the smoothing matrix $n$ times. Fortunately the above formula can be expressed in a way where the smoothing matrix does not need to be calculated again.

$$CV(\nu) = \frac{1}{n} \sum_i \left( \frac{y_i - \hat{y}_i^{\nu}}{1 - H_i^{\nu}} \right)^2$$

Parameters
- estimator (Estimator) – Linear smoothing estimator.
- X (FDataGrid) – Functional data to smooth.
- y (FDataGrid) – Functional data target. Should be the same as X.

Returns Cross validation score, with negative sign, as it is a penalization.

Return type float

__init__()

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

class skfda.preprocessing.smoothing.validation.LinearSmotherGeneralizedCVScorer(
    penalization_function=None)

Generalized cross validation scoring method for linear smoothers.

It calculates the general cross validation score for every sample in a FDataGrid object given a smoothing matrix $H^\nu$ calculated with a parameter $\nu$:

$$GCV(\nu) = \Xi(\nu, n) \frac{1}{n} \sum_i (y_i - \hat{y}_i^\nu)^2$$

Where $\hat{y}_i^\nu$ is the adjusted $y_i$ and $\Xi$ is a penalization function. By default the penalization function is:

$$\Xi(\nu, n) = \left(1 - \frac{tr(H^\nu)}{n}\right)^{-2}$$

but others such as the Akaike’s information criterion can be considered.

Parameters

- estimator (Estimator) – Linear smoothing estimator.
- X (FDataGrid) – Functional data to smooth.
- y (FDataGrid) – Functional data target. Should be the same as X.

Returns Cross validation score, with negative sign, as it is a penalization.

Return type float

Methods

__init__([penalization_function]) Initialize self.

__init__(penalization_function=None) Initialize self. See help(type(self)) for accurate signature.

The LinearSmotherGeneralizedCVScorer object accepts also an optional penalization_function, used instead of the default one. The available ones are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>skfda.preprocessing.smoothing.validation.akaike_information_criterion(...)</td>
<td>Akaike’s information criterion for cross validation.</td>
</tr>
<tr>
<td>skfda.preprocessing.smoothing.validation.finite_prediction_error(...)</td>
<td>Finite prediction error for cross validation.</td>
</tr>
<tr>
<td>skfda.preprocessing.smoothing.validation.shibata(...)</td>
<td>Shibata’s model selector for cross validation.</td>
</tr>
<tr>
<td>skfda.preprocessing.smoothing.validation.rice(...)</td>
<td>Rice’s bandwidth selector for cross validation.</td>
</tr>
</tbody>
</table>
akaike_information_criterion

skfda.preprocessing.smoothing.validation.akaike_information_criterion(hat_matrix)

Akaike’s information criterion for cross validation.

\[ \Xi(\nu, n) = \exp \left( 2 \cdot \frac{\text{tr}(\hat{H}^\nu)}{n} \right) \]

**Parameters**
- **hat_matrix** (*numpy.darray*) – Smoothing matrix whose penalization score is desired.

**Returns**
- penalization given by the Akaike’s information criterion.

**Return type**
- float

finite_prediction_error

skfda.preprocessing.smoothing.validation.finite_prediction_error(hat_matrix)

Finite prediction error for cross validation.

\[ \Xi(\nu, n) = 1 + \frac{\text{tr}(\hat{H}^\nu)}{n} \]

**Parameters**
- **hat_matrix** (*numpy.darray*) – Smoothing matrix whose penalization score is desired.

**Returns**
- penalization given by the finite prediction error.

**Return type**
- float

shibata

skfda.preprocessing.smoothing.validation.shibata(hat_matrix)

Shibata’s model selector for cross validation.

\[ \Xi(\nu, n) = 1 + 2 \cdot \frac{\text{tr}(\hat{H}^\nu)}{n} \]

**Parameters**
- **hat_matrix** (*numpy.darray*) – Smoothing matrix whose penalization score is desired.

**Returns**
- penalization given by the Shibata’s model selector.

**Return type**
- float

rice

skfda.preprocessing.smoothing.validation.rice(hat_matrix)

Rice’s bandwidth selector for cross validation.

\[ \Xi(\nu, n) = \left( 1 - 2 \cdot \frac{\text{tr}(\hat{H}^\nu)}{n} \right)^{-1} \]

**Parameters**
- **hat_matrix** (*numpy.darray*) – Smoothing matrix whose penalization score is desired.
Returns penalization given by the Rice’s bandwidth selector.

Return type float

An utility class is also provided, which inherits from the scikit-learn class `GridSearchCV` and performs a grid search using the scorers to find the best `smoothing_parameter` from a list.

```
skfda.preprocessing.smoothing.validation.SmoothingParameterSearch(...)
```

Chooses the best smoothing parameter and performs smoothing.

**SmoothingParameterSearch**

```
class skfda.preprocessing.smoothing.validation.SmoothingParameterSearch(estimator, param_values, *, scoring=None, n_jobs=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan)
```

Chooses the best smoothing parameter and performs smoothing.

Performs the smoothing of a FDataGrid object choosing the best parameter of a given list using a cross validation scoring method.

**Note:** This is similar to fitting a scikit-learn GridSearchCV over the data, using the cv_method as a scorer.

**Parameters**

- **estimator** (*smoother estimator*) – scikit-learn compatible smoother.
- **param_values** (*iterable*) – iterable containing the values to test for `smoothing_parameter`.
- **scoring** (*scoring method*) – scoring method used to measure the performance of the smoothing. If `None` (the default) the `score` method of the estimator is used.
- **n_jobs** (*int or None, optional (default=None]*) – Number of jobs to run in parallel. `None` means 1 unless in a joblib.parallel_backend context. `-1` means using all processors. See scikit-learn Glossary for more details.
- **pre_dispatch** (*int, or string, optional*) – Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:
  - None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
  - An int, giving the exact number of total jobs that are spawned
  - A string, giving an expression as a function of `n_jobs`, as in ‘2*n_jobs’
• `verbose (integer)`: Controls the verbosity: the higher, the more messages.
• `error_score ('raise' or numeric)`: Value to assign to the score if an error occurs in estimator fitting. If set to `raise`, the error is raised. If a numeric value is given, `FitFailedWarning` is raised. This parameter does not affect the refit step, which will always raise the error. Default is `np.nan`.

Examples

Creates a `FDataGrid` object of the function $y = x^2$ and performs smoothing by means of the k-nearest neighbours method.

```python
>>> import skfda
>>> from skfda.preprocessing.smoothing import kernel_smoothers
>>> x = np.linspace(-2, 2, 5)
>>> fd = skfda.FDataGrid(x ** 2, x)
>>> grid = SmoothingParameterSearch(...
    kernel_smoothers.KNeighborsSmotherer(), [2, 3])
>>> _ = grid.fit(fd)
>>> np.array(grid.cv_results_['mean_test_score']).round(2)
array([-11.67, -12.37])
>>> round(grid.best_score_, 2)
-11.67
>>> grid.best_params_['smoothing_parameter']
2
>>> grid.best_estimator_.hat_matrix().round(2)
array([[ 0.5 , 0.5 , 0. , 0. , 0. ],
       [ 0.33, 0.33, 0.33, 0. , 0. ],
       [ 0. , 0.33, 0.33, 0.33, 0. ],
       [ 0. , 0. , 0.33, 0.33, 0.33],
       [ 0. , 0. , 0. , 0.5 , 0.5 ]])
>>> grid.transform(fd).round(2)
FDataGrid(
    array([[ [ 2.5 ],
             [ 1.67],
             [ 0.67],
             [ 1.67],
             [ 2.5 ]]]),
    sample_points=array([-2., -1., 0., 1., 2.]),
    domain_range=array([-2., 2.]),
    ...
)
```

Other validation methods can be used such as cross-validation or general cross validation using other penalization functions.

```python
>>> grid = SmoothingParameterSearch(...
    kernel_smoothers.KNeighborsSmotherer(), [2, 3],
    scoring=LinearSmotherLeaveOneOutScorer())
>>> _ = grid.fit(fd)
>>> np.array(grid.cv_results_['mean_test_score']).round(2)
array([-4.2, -5.5])
```

(continues on next page)
Different output points can also be used. In that case the value used as a target is still the smoothed value at the input points:

```python
>>> output_points = np.linspace(-2, 2, 9)
>>> grid = SmoothingParameterSearch(
...     kernel_smoothers.KNeighborsSmooother(output_points=output_points), [2,3])
>>> _ = grid.fit(fd)
>>> np.array(grid.cv_results_['mean_test_score']).round(2)
array([-11.67, -12.37])
```

## Methods

Methods
**__init__**(estimator, param_values, *, ...) Initialize self.

**decision_function**(X) Call decision_function on the estimator with the best found parameters.

**fit**(X, y, groups) Run fit with all sets of parameters.

**get_params**(deep) Get parameters for this estimator.

**inverse_transform**(Xt) Call inverse_transform on the estimator with the best found params.

**predict**(X) Call predict on the estimator with the best found parameters.

**predict_log_proba**(X) Call predict_log_proba on the estimator with the best found parameters.

**predict_proba**(X) Call predict_proba on the estimator with the best found parameters.

**score**(X[, y]) Returns the score on the given data, if the estimator has been refitted.

**set_params**(**params**) Set the parameters of this estimator.

**transform**(X) Call transform on the estimator with the best found parameters.

---

**__init__**(estimator, param_values, *, scoring=None, n_jobs=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan) Initialize self. See help(type(self)) for accurate signature.

### Attributes

**classes_**

**decision_function**(X) Call decision_function on the estimator with the best found parameters. Only available if refit=True and the underlying estimator supports decision_function.

Parameters

- **X** *(indexable, length n_samples)* – Must fulfill the input assumptions of the underlying estimator.

**fit**(X, y=None, groups=None, **fit_params) Run fit with all sets of parameters.

Parameters

- **X** *(array-like of shape (n_samples, n_features))* – Training vector, where n_samples is the number of samples and n_features is the number of features.

- **y** *(array-like of shape (n_samples, n_output) or (n_samples,), optional)* – Target relative to X for classification or regression; None for unsupervised learning.

- **groups** *(array-like, with shape (n_samples,), optional)* – Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a “Group” cv instance (e.g., GroupKFold).

- ****fit_params**(dict of string -> object) – Parameters passed to the fit method of the estimator

---

1.2. Preprocessing
get_params(deep=True)
Get parameters for this estimator.

Parameters deep (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type mapping of string to any

inverse_transform(Xt)
Call inverse_transform on the estimator with the best found params.
Only available if the underlying estimator implements inverse_transform and refit=True.

Parameters Xt (indexable, length n_samples) – Must fulfill the input assumptions of the underlying estimator.

predict(X)
Call predict on the estimator with the best found parameters.
Only available if refit=True and the underlying estimator supports predict.

Parameters X (indexable, length n_samples) – Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(X)
Call predict_log_proba on the estimator with the best found parameters.
Only available if refit=True and the underlying estimator supports predict_log_proba.

Parameters X (indexable, length n_samples) – Must fulfill the input assumptions of the underlying estimator.

predict_probas(X)
Call predict_probas on the estimator with the best found parameters.
Only available if refit=True and the underlying estimator supports predict_probas.

Parameters X (indexable, length n_samples) – Must fulfill the input assumptions of the underlying estimator.

score(X, y=None)
Returns the score on the given data, if the estimator has been refit.

This uses the score defined by scoring where provided, and the best_estimator_.score method otherwise.

Parameters
- X (array-like of shape (n_samples, n_features)) – Input data, where n_samples is the number of samples and n_features is the number of features.
- y (array-like of shape (n_samples, n_output) or (n_samples,), optional) – Target relative to X for classification or regression; None for unsupervised learning.

Returns score

Return type float

set_params(**params)
Set the parameters of this estimator.
The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Parameters**

**params** (*dict*) – Estimator parameters.

**Returns**

`self` – Estimator instance.

**Return type**

`object`

**transform** (*X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

**Parameters**

`X` (*indexable, length n_samples*) – Must fulfill the input assumptions of the underlying estimator.

Examples using `skfda.preprocessing.smoothing.validation.SmoothingParameterSearch`

- *Kernel Smoothing*

**References**


### 1.2.2 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 is shown the basic usage of this method.

```python
skfda.preprocessing.registration.ShiftRegistration([...])
```

Register a functional dataset using shift alignment.

```python
class skfda.preprocessing.registration.ShiftRegistration(max_iter=5, tol=0.01, template='mean', extrapolation=None, step_size=1, restrict_domain=False, initial='zeros', output_points=None)
```

Register a functional dataset using shift alignment.
Realizes the registration of a set of curves using a shift alignment \cite{RaSi2005-7-2}. Let \( \{x_i(t)\}_{i=1}^N \) be a functional dataset, calculates \( \delta_i \) for each sample such that \( x_i(t + \delta_i) \) minimizes the least squares criterion:

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

Estimates each shift parameter \( \delta_i \) iteratively by using a modified Newton-Raphson algorithm, updating the template \( \mu \) in each iteration as is described in detail in \cite{RaSi2005-7-9-1}.

Method only implemented for univariate functional data.

**Parameters**

- `max_iter` (*int, optional*) – Maximum number of iterations. Defaults sets to 5. Generally 2 or 3 iterations are sufficient to obtain a good alignment.

- `tol` (*float, optional*) – Tolerance allowable. The process will stop if \( \max_i |\delta_i^{(r)} - \delta_i^{(r-1)}| < tol \). Default sets to 1e-2.

- `template` (*str, callable or FData, optional*) – Template to use in the least squares criterion. If template=”mean” it is use the functional mean as in the original paper. The template can be a callable that will receive an FDataGrid with the samples and will return another FDataGrid as a template, such as any of the means or medians of the module `skfda.exploratory.stats`. If the template is an FData is used directly as the final template to the registration, if it is a callable or “mean” the template is computed iteratively constructing a temporal template in each iteration. In \cite{RaSi2005-7-9-1} is described in detail this procedure. Defaults to “mean”.

- `extrapolation` (*str or Extrapolation, optional*) – Controls the extrapolation mode for points outside the domain range. By default uses the method defined in the data to be transformed. See the `extrapolation` documentation to obtain more information.

- `step_size` (*int or float, optional*) – Parameter to adjust the rate of convergence in the Newton-Raphson algorithm, see \cite{RaSi2005-7-9-1}. Defaults to 1.

- `restrict_domain` (*bool, optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation, in which case only the fit_transform method will be available, as training and transformation must be done together. Defaults to False.

- `initial` (*str or array_like, optional*) – Array with an initial estimation of shifts. Default uses a list of zeros for the initial shifts.

- `output_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.

**Template**

- `template_` – Template \( \mu \) learned during the fitting used to the transformation.

  Type `FData`

**Deltas**

- `deltas_` – List of shifts \( \delta_i \) applied during the last transformation.

  Type `numpy.ndarray`

**Iterations**

- `n_iter_` – Number of iterations performed during the last transformation.
Type  int

Note:  Due to the use of derivatives for the estimation of the shifts, the samples to be registered may be smooth for the correct convergence of the method.

Examples

```python
>>> from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.basis import Fourier

Registration and creation of dataset in discretized form:

```python
>>> fd = make_sinusoidal_process(n_samples=10, error_std=0,
... random_state=1)
>>> reg = ShiftRegistration(extrapolation="periodic")
>>> fd_registered = reg.fit_transform(fd)
>>> fd_registered
FDataGrid(...)
```

Shifts applied during the transformation

```python
>>> reg.deltas_.round(3)
array([-0.126, 0.190, 0.029, 0.036, -0.104, 0.116, ..., -0.058])
```

Registration and creation of a dataset in basis form using the transformation previously fitted:

```python
>>> fd = make_sinusoidal_process(n_samples=2, error_std=0,
... random_state=2)
>>> fd_basis = fd.to_basis(Fourier())
>>> reg.transform(fd_basis)
FDataBasis(...)
```

References

Methods

```python
__init__([max_iter, tol, template, ...]) Initialize self.
fit(X[, y]) Fit the estimator.
fit_transform(X[, y]) Fit the estimator and transform the data.
get_params([deep]) Get parameters for this estimator.
inverse_transform(X[, y]) Applies the inverse transformation.
score(X[, y]) Returns the percentage of total variation removed.
set_params(**params) Set the parameters of this estimator.
transform(X[, y]) Register the data.
```
__init__(max_iter=5, tol=0.01, template='mean', extrapolation=None, step_size=1, restrict_domain=False, initial='zeros', output_points=None)
Initialize self. See help(type(self)) for accurate signature.

fit(X: skfda.representation._functional_data.FData, y=None)
Fit the estimator.

Parameters
• X (FData) – Functional dataset used to construct the template for the alignment.
• y (ignored) – not used, present for API consistency by convention.

Returns
self
Return type RegistrationTransformer
Raises AttributeError – If this method is call when restrict_domain=True.

fit_transform(X: skfda.representation._functional_data.FData, y=None)
Fit the estimator and transform the data.

Parameters
• X (FData) – Functional dataset to be transformed.
• y (ignored) – not used, present for API consistency by convention.

Returns
Functional data registered.
Return type FData

get_params(deep=True)
Get parameters for this estimator.

Parameters
depth (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.
Return type
dict: \(\text{mapping of string to any}\)

inverse_transform(X: skfda.representation._functional_data.FData, y=None)
Applies the inverse transformation.

Applies the opposite shift used in the last call to transform.

Parameters
• X (FData) – Functional dataset to be transformed.
• y (ignored) – not used, present for API consistency by convention.

Returns
Functional data registered.
Return type FData

Examples:
Creates a synthetic functional dataset.

```python
>>> from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process
>>> fd = make_sinusoidal_process(error_std=0, random_state=1)
>>> fd.extrapolation = 'periodic'
```

Dataset registration and centering.
Reverse the translation applied during the registration.

```python
>>> reg = ShiftRegistration()
>>> fd_registered = reg.fit_transform(fd)
>>> fd_centered = fd_registered - fd_registered.mean()
```

```python
>>> reg.inverse_transform(fd_centered)
FDataGrid(...)
```

`score(X: skfda.representation.functional_data.FData, y=None)`

Returns the percentage of total variation removed.

Computes the squared multiple correlation index of the proportion of the total variation due to phase, defined as:

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

where $\text{MSE}_{\text{total}}$ is the mean squared error and $\text{MSE}_{\text{phase}}$ is the mean squared error due to the phase explained by the registration procedure. See `AmplitudePhaseDecomposition` for a detailed explanation.

**Parameters**
- `X` (`FData`) – Functional data to be registered
- `y` (`Ignored`) – Ignored, only for API conventions.

**Returns** float.

See also:
- `AmplitudePhaseDecomposition LeastSquares SobolevLeastSquares PairwiseCorrelation`

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters** **params (`dict`) – Estimator parameters.

**Returns** `self` – Estimator instance.

**Return type** object

`transform(X: skfda.representation.functional_data.FData, y=None)`

Register the data.

Transforms the data using the template previously learned during fitting.

**Parameters**
- `X` (`FData`) – Functional dataset to be transformed.
- `y` (`Ignored`) – not used, present for API consistency by convention.

**Returns** Functional data registered.

**Return type** `FData`

** Raises AttributeError – If this method is called when restrict_domain=True.**
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.

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

**landmark_shift**

```python
skfda.preprocessing.registration.landmark_shift(fd, landmarks, location=None, *, restrict_domain=False, extrapolation=None, eval_points=None, **kwargs)
```

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 aligned. If a numeric value is passed the landmarks will be aligned 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 $\frac{1}{2}(\max(\text{landmarks}) + \min(\text{landmarks}))$ which 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

```python
>>> 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

```python
>>> 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

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

Examples using `skfda.preprocessing.registration.landmark_shift`

- **Landmark shift**

`landmark_shift_deltas`

`skfda.preprocessing.registration.landmark_shift_deltas(fd, landmarks, location=None)`

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 $\delta_i$ such 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 $\frac{1}{2}(max(landmarks) + min(landmarks))$ which 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

```python
>>> 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

```python
>>> 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

```python
>>> 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

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

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skfda.preprocessing.registration.landmark_registration(fd, ...)</code></td>
<td>Perform landmark registration of the curves.</td>
</tr>
<tr>
<td><code>skfda.preprocessing.registration.landmark_registration_warping(fd, ...)</code></td>
<td>Calculate the transformation used in landmark registration.</td>
</tr>
</tbody>
</table>

**landmark_registration**

`skfda.preprocessing.registration.landmark_registration(fd, landmarks, *, location=None, eval_points=None)`

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 aligned. 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:

Examples

```python
>>> 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

```python
>>> 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

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

This method will work for FDataBasis as for FDataGrids

```python
>>> fd = fd.to_basis(BSpline(n_basis=12))
>>> landmark_registration(fd, landmarks)
FDataBasis(...)
```

Examples using `skfda.preprocessing.registration.landmark_registration`

- Landmark registration

landmark_registration_warping

```python
skfda.preprocessing.registration.landmark_registration_warping(fd, landmarks, *, location=None, eval_points=None)
```

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 aligned. 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:

**Examples**

```python
>>> 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

```python
>>> 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

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

The registered function can be obtained using function composition

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

**Examples using** skfda.preprocessing-registration.landmark_registration_warping

• **Landmark registration**

**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.
ElasticRegistration

class skfda.preprocessing.registration.ElasticRegistration:

Align a FDatagrid using the SRSF framework.

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

\[
\gamma^* = \arg\min_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)
\]

Where \( d_\lambda \) denotes the extended Fisher-Rao distance with a penalty term, used to control the amount of warping.

\[
d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\gamma} - SRSF(g)\|_{L^2}^2 + \lambda R(\gamma)
\]

In the implementation it is used as penalty term

\[
R(\gamma) = \|\sqrt{\gamma} - 1\|_{L^2}^2
\]

Which 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, also known as elastic mean, which is the local minimum of the sum of squares of elastic distances. See \texttt{elastic_mean()}.

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

**Parameters**

- **template** (str, FDatagrid or callable, 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 elastic mean, in which case \texttt{elastic_mean()} is called.
- **penalty_term** (float, optional) – Controls the amount of elasticity. Defaults to 0.
- **output_points** (array_like, optional) – Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid which will be transformed.
- **grid_dim** (int, optional) – Dimension of the grid used in the DP alignment algorithm. Defaults 7.

**template_**

Template learned during fitting, used for alignment in \texttt{transform()}.

**Type** FDatagrid

**warping_**

Warping applied during the last transformation.

**Type** FDatagrid
Elastic registration of with train/test sets.

```python
>>> from skfda.preprocessing.registration import ElasticRegistration
>>> from skfda.datasets import make_multimodal_samples
>>> X_train = make_multimodal_samples(n_samples=15, random_state=0)
>>> X_test = make_multimodal_samples(n_samples=3, random_state=1)
```

Fit the transformer, which learns the elastic mean of the train set as template.

```python
>>> elastic_registration = ElasticRegistration()
>>> elastic_registration.fit(X_train)
ElasticRegistration(...)
```

Registration of the test set.

```python
>>> elastic_registration.transform(X_test)
FDataGrid(...)
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>(template, penalty, output_points, ...) - Initializes the registration transformer</td>
</tr>
<tr>
<td><code>fit</code></td>
<td>(X, y) - Fit the transformer.</td>
</tr>
<tr>
<td><code>fit_transform</code></td>
<td>(X, y) - Fit to data, then transform it.</td>
</tr>
<tr>
<td><code>get_params</code></td>
<td>(deep) - Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>inverse_transform</code></td>
<td>(X, y) - Reverse the registration procedure previously applied.</td>
</tr>
<tr>
<td><code>score</code></td>
<td>(X, y) - Returns the percentage of total variation removed.</td>
</tr>
<tr>
<td><code>set_params</code></td>
<td>(**params) - Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>(X, y) - Apply elastic registration to the data.</td>
</tr>
</tbody>
</table>

```python
__init__ (template=’elastic mean’, penalty=0.0, output_points=None, grid_dim=7)
```

Initializes the registration transformer

```python
fit(X: skfda.representation.grid.FDataGrid = None, y=None)
```

Fit the transformer.

Learns the template used during the transformation.

**Parameters**

- **X** (FDataGrid, optional) – Functional samples used as training samples. If the template provided it is an FDataGrid this samples are it is not need to construct the template from the samples and this argument is ignored.

- **y** (Ignored) – Present for API conventions.

**Returns** self.
Return type RegistrationTransformer

`fit_transform(X, y=None, **fit_params)`
Fit to data, then transform it.
Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- `y` (numpy array of shape `[n_samples]`) – Target values.
- **fit_params** (dict) – Additional fit parameters.

Returns `X_new` – Transformed array.

Return type numpy array of shape `[n_samples, n_features_new]`

`get_params(deep=True)`
Get parameters for this estimator.

Parameters

- `deep` (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns `params` – Parameter names mapped to their values.

Return type mapping of string to any

`inverse_transform(X: skfda.representation.grid.FDataGrid, y=None)`
Reverse the registration procedure previously applied.
Let `\gamma(t)` the warping applied to construct a registered functional datum `f^*(t) = f(\gamma(t))`.
Given a functional datum `f^*(t)` it is computed
:math:`\gamma^{-1}(t)` to reverse the registration procedure `f(t) = f^*(\gamma^{-1}(t))`.

Parameters

- `X` (FDataGrid) – Functional data to apply the reverse transform.
- `y` (Ignored) – Present for API conventions.

Returns Functional data compose by the inverse warping.

Return type FDataGrid

Raises

- `ValueError` – If the warpings `\gamma` were not build via
- `transform()` or if the number of samples of `X` is different
- than the number of samples of the dataset previosly transformed.

Examples

Center the datasets taking into account the misalignment.

```python
>>> from skfda.preprocessing.registration import \
  ElasticRegistration
... >>> from skfda.datasets import make_multimodal_samples
>>> X = make_multimodal_samples(random_state=0)
```

1.2. Preprocessing
Registration of the dataset.

```python
>>> elastic_registration = ElasticRegistration()
>>> X = elastic_registration.fit_transform(X)
```

Subtract the elastic mean build as template during the registration and reverse the transformation.

```python
>>> X = X - elastic_registration.template_
>>> X_center = elastic_registration.inverse_transform(X)
```

See also:

invert_warping()

```python
def score(X: skfda.representation._functional_data.FData, y=None)
Returns the percentage of total variation removed.
```

Computes the squared multiple correlation index of the proportion of the total variation due to phase, defined as:

\[ R^2 = \frac{\text{MSE}_{\text{phase}}}{\text{MSE}_{\text{total}}} \]

where MSE_{total} is the mean squared error and MSE_{phase} is the mean squared error due to the phase explained by the registration procedure. See AmplitudePhaseDecomposition for a detailed explanation.

Parameters

- **X** (*FData*) – Functional data to be registered
- **y** (*Ignored*) – Ignored, only for API conventions.

Returns float.

See also:

AmplitudePhaseDecomposition LeastSquares SobolevLeastSquares PairwiseCorrelation

```python
set_params(**params)
Set the parameters of this estimator.
```

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>_<parameter>` so that it’s possible to update each component of a nested object.

Parameters **params** (*dict*) – Estimator parameters.

Returns self – Estimator instance.

Return type object

```python
def transform(X: skfda.representation.grid.FDataGrid, y=None)
Apply elastic registration to the data.
```

Parameters

- **X** (*FDataGrid*) – Functional data to be registered.
- **y** (*ignored*) – Present for API conventions.

Returns Registered samples.
Return type FDataGrid

Examples using skfda.preprocessing.registration.ElasticRegistration

- Elastic registration
- Pairwise alignment

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.
elastic_mean(...)
```

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 \( \Gamma \) denotes the space of warping functions. The karcher mean in this space is defined as

\[
[\mu_q] = \arg\min_{[q] \in \mathcal{A}} \sum_{i=1}^{n} d^2_{\lambda}([q], [q_i])
\]

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.
- `penalty` (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.
- `max_iter` (int) – Maximum number of iterations. Defaults to 20.
- `tol` (float) – Convergence criterion, the algorithm will stop if \( |\mu_\nu - \mu_{\nu-1}| \leq \text{tol} \).
- `initial` (float) – Value of the mean at the starting point. By default takes the average of the initial points of the samples.
grid_dim (int, optional) – Dimension of the grid used in the alignment algorithm. Defaults to 7.

kwargs (**) – Named options to be passed 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

Examples using skfda.preprocessing-registration.elastic.elastic_mean

- Elastic registration

warping_mean

skfda.preprocessing-registration.elastic.warping_mean(warping, *, max_iter=100, tol=1e-06, step_size=0.3)

Compute the karcher mean of a set of warpings.

Let $\gamma_i = 1...n$ be a set of warping functions $\gamma_i : [a, b] \rightarrow [a, b]$ in $\Gamma$, 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].

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

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.
- max_iter (int) – Maximum number of interations. Defaults to 100.
- tol (float) – Convergence criterion, if the norm of the mean of the shooting vectors, $|v| < tol$, the algorithm will stop. Defaults to 1e-5.
- step_size (float) – Step size $\epsilon$ used to update the mean. Default to 1.

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

SRSF

class skfda.preprocessing-registration.elastic.SRSF(output_points=None, initial_value=None)

Square-Root Slope Function (SRSF) transform.
Let \( f : [a, b] \to \mathbb{R} \) be an absolutely continuous function, the SRSF transform is defined as

\[
SRSF(f(t)) = sgn(f(t))\sqrt{|\dot{f}(t)|} = q(t)
\]

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

The inverse SRSF transform is defined as

\[
f(t) = f(a) + \int_a^t q(t)q(t)dt.
\]

This transformation is a mapping up to constant. Given the SRSF and the initial value \( f(a) \) the original function can be obtained, for this reason it is necessary to store the value \( f(a) \) during the fit, which is dropped due to derivation. If it is applied the inverse transformation without fit the estimator it is assumed that \( f(a) = 0 \).

**eval_points**

Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.

  **Type** array_like, optional

**initial_value**

Initial value to apply in the inverse transformation. If \( None \) there are stored the initial values of the functions during the transformation to apply during the inverse transformation. Defaults \( None \).

  **Type** float, optional

**Note:** Due to the use of derivatives it is recommended that the samples are sufficiently smooth, or have passed a smoothing preprocessing before, in order to achieve good results.

### References

### Examples

Create a toy dataset and apply the transformation and its inverse.

```python
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.preprocessing.registration.elastic import SRSF
>>> fd = make_sinusoidal_process(error_std=0, random_state=0)
>>> srsf = SRSF()
>>> q = srsf.fit_transform(fd)
```

Fits the estimator (to apply the inverse transform) and apply the SRSF

```python
>>> q = srsf.fit_transform(fd)
```

Apply the inverse transform.

```python
>>> fd_pull_back = srsf.inverse_transform(q)
```

The original and the pull back \( fd \) are almost equal
>>> zero = fd - fd_pull_back
>>> zero.data_matrix.flatten().round(3)
array([ 0. , 0. , 0. , ... ])

Methods

__init__((output_points, initial_value)) Initializes the transformer.

fit([X, y]) This transformer do not need to be fitted.

fit_transform(X[, y]) Fit to data, then transform it.

get_params([deep]) Get parameters for this estimator.

inverse_transform(X[, y]) Computes the inverse SRSF transform.

set_params(**params) Set the parameters of this estimator.

transform(X[, y]) Computes the square-root slope function (SRSF) transform.

__init__(output_points=None, initial_value=None) Initializes the transformer.

Parameters

- eval_points – (array_like, optional): Set of points where the functions are evaluated, by default uses the sample points of the FDataGrid transformed.

- initial_value (float, optional) – Initial value to apply in the inverse transformation. If None there are stored the initial values of the functions during the transformation to apply during the inverse transformation. Defaults None.

fit(X=None, y=None) This transformer do not need to be fitted.

Parameters

- X (Ignored) – Present for API conventions.

- y (Ignored) – Present for API conventions.

Returns self

Return type (Estimator)

fit_transform(X, y=None, **fit_params) Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- X (numpy array of shape [n_samples, n_features]) – Training set.

- y (numpy array of shape [n_samples]) – Target values.

- **fit_params (dict) – Additional fit parameters.

Returns X__new – Transformed array.

Return type numpy array of shape [n_samples, n_features_new]

get_params(deep=True) Get parameters for this estimator.
**Parameters** `deep` *(bool, default=True)* – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** `params` – Parameter names mapped to their values.

**Return type** mapping of string to any

`inverse_transform(X: skfda.representation.grid.FDataGrid, y=None)`

Computes the inverse SRSF transform.

Given the srsf and the initial value the original function can be obtained as [SK16-4-6-2] : 

\[
    f(t) = f(a) + \int_a^t q(t) |q(t)| dt
\]

where \( q(t) = SRSF(f(t)) \).

If it is applied this inverse transformation without fitting the estimator it is assumed that \( f(a) = 0 \).

**Parameters**

- `X` *(FDataGrid)* – SRSF to be transformed.

- `y` *(Ignored)* – Present for API conventions.

**Returns** Functions in the original space.

**Return type** FDataGrid

**Raises** `ValueError` – If functions are multidimensional.

**References**

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters** `**params` *(dict)* – Estimator parameters.

**Returns** `self` – Estimator instance.

**Return type** object

`transform(X: skfda.representation.grid.FDataGrid, y=None)`

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

Let \( f : [a, b] \to \mathbb{R} \) be an absolutely continuous function, the SRSF transform is defined as [SK16-4-6-1]:

\[
    SRSF(f(t)) = sgn(f(t)) \sqrt{f(t)} = q(t)
\]

**Parameters**

- `X` *(FDataGrid)* – Functions to be transformed.

- `y` *(Ignored)* – Present for API conventions.

**Returns** SRSF functions.

**Return type** FDataGrid

**Raises** `ValueError` – If functions are not univariate.
References

Validation

This module contains several classes methods for the quantification and validation of the registration procedure.

- `skfda.preprocessing.registration.validation.AmplitudePhaseDecomposition(...)`: Compute mean square error measures for amplitude and phase variation.
- `skfda.preprocessing.registration.validation.LeastSquares(...)`: Cross-validated measure of the registration procedure.
- `skfda.preprocessing.registration.validation.SobolevLeastSquares(...)`: Cross-validated measure of the registration procedure.
- `skfda.preprocessing.registration.validation.PairwiseCorrelation(...)`: Cross-validated measure of pairwise correlation between functions.

AmplitudePhaseDecomposition

```python
class skfda.preprocessing.registration.validation.AmplitudePhaseDecomposition(
    return_stats=False,
    eval_points=None)
```

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 and returns 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

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

The measures of amplitude and phase mean square error are

\[
\text{MSE}_{\text{amp}} = C_R \frac{1}{N} \sum_{i=1}^{N} \int [y_i(t) - \bar{y}(t)]^2 dt
\]

\[
\text{MSE}_{\text{phase}} = \int [C_R \bar{y}^2(t) - \bar{x}^2(t)] dt
\]

where the constant \( C_R \) is defined as

\[
C_R = 1 + \frac{\frac{1}{N} \sum_{i=1}^{N} \int [Dh_i(t) - \bar{Dh}(t)][y_i^2(t) - \bar{y}^2(t)] dt}{\frac{1}{N} \sum_{i=1}^{N} \int y_i^2(t) dt}
\]

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 total mean square error is decomposed in the two sources of variability.

\[
\text{MSE}_{\text{total}} = \text{MSE}_{\text{amp}} + \text{MSE}_{\text{phase}}
\]
The squared multiple correlation index of the proportion of the total variation due to phase is defined as:

\[ R^2 = \frac{\text{MSE}_{\text{phase}}}{\text{MSE}_{\text{total}}} \]

See [KR08-3] for a detailed explanation.

**return_stats**

If `true` returns a named tuple with four values: \( R^2, \text{MSE}_{\text{amp}}, \text{MSE}_{\text{pha}} \) and \( C_R \). Otherwise the squared correlation index \( R^2 \) is returned. Default `False`.

**eval_points**

Set of points where the functions are evaluated to obtain a discrete representation and perform the calculation.

**Parameters**

- `estimator` (*RegistrationTransformer*) – Registration transformer.
- `X` (*FData*) – Unregistered functions.
- `y` (*FData*, optional) – Target data, generally the same as `X`. By default ‘None’, which uses `X` as target.

**Returns**

Squared correlation index \( R^2 \) if `return_stats` is `False`. Otherwise a named tuple containing:

- `r_squared`: Squared correlation index \( R^2 \).
- `mse_amp`: Mean square error of amplitude \( \text{MSE}_{\text{amp}} \).
- `mse_pha`: Mean square error of phase \( \text{MSE}_{\text{pha}} \).
- `c_r`: Constant \( C_R \).

**Return type** *(float or NamedTuple)*

**Raises** `ValueError` – If the functional data is not univariate.

**Examples**

Calculate the score of the shift registration of a sinusoidal process synthetically generated.

```python
>>> from skfda.preprocessing.registration.validation import AmplitudePhaseDecomposition
>>> from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process

>>> X = make_sinusoidal_process(error_std=0, random_state=0)
```

Fit the registration procedure.
```python
>>> shift_registration = ShiftRegistration()
>>> shift_registration.fit(X)
ShiftRegistration(...)
```

Compute the $R^2$ correlation index

```python
>>> scorer = AmplitudePhaseDecomposition()
>>> score = scorer(shift_registration, X)
>>> round(score, 3)
0.972
```

Also it is possible to get all the values of the decomposition.

```python
>>> scorer = AmplitudePhaseDecomposition(return_stats=True)
>>> stats = scorer(shift_registration, X)
>>> round(stats.r_squared, 3)
0.972
>>> round(stats.mse_amp, 3)
0.007
>>> round(stats.mse_pha, 3)
0.227
>>> round(stats.c_r, 3)
1.0
```

See also:
- LeastSquares
- SobolevLeastSquares
- PairwiseCorrelation

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__([return_stats, eval_points])</code></td>
<td>Initialize the transformer</td>
</tr>
<tr>
<td><code>score_function(X, y, *, warping)</code></td>
<td>Compute the score of the transformation performed.</td>
</tr>
</tbody>
</table>

```python
LeastSquares
```

```python
class skfda.preprocessing-registration-validation.LeastSquares(return_stats=False, eval_points=None)
```

Cross-validated measure of the registration procedure.
Computes a cross-validated measure of the level of synchronization [James07]:

\[
l_s = 1 - \frac{1}{N} \sum_{i=1}^{N} \int \left( \tilde{f}_i(t) - \frac{1}{N-1} \sum_{j \neq i} \tilde{f}_j(t) \right)^2 dt = \frac{1}{N} \sum_{i=1}^{N} \int \left( \tilde{f}_i(t) - \frac{1}{N-1} \sum_{j \neq i} f_j(t) \right)^2 dt
\]

where \( f_i \) and \( \tilde{f}_i \) are the original and the registered data respectively.

The \( l_s \) measures the total cross-sectional variance of the aligned functions, relative to the original value. A value of 1 would indicate an identical shape for all registered curves, while zero corresponds to no improvement in the synchronization. It can be negative because the model can be arbitrarily worse.

**eval_points**

Set of points where the functions are evaluated to obtain a discrete representation and perform the calculation.

**Type** array_like, optional

**Parameters**

- **estimator** (RegistrationTransformer) – Registration transformer.
- **X** (FData) – Original functional data.
- **y** (FData) – Registered functional data.

**Note:** The original least square measure used in [S11-5-2-1] is defined as \( 1 - l_s \), but has been modified according to the scikit-learn scorers, where higher values correspond to better cross-validated measures.

**References**

**Examples**

Calculate the score of the shift registration of a sinusoidal process synthetically generated.

```python
>>> from skfda.preprocessing.registration.validation import LeastSquares
... from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process
>>> X = make_sinusoidal_process(error_std=0, random_state=0)
Fit the registration procedure.

```Python
>>> shift_registration = ShiftRegistration()
>>> shift_registration.fit(X)
ShiftRegistration(...)
```

Compute the least squares score. ```scorer = LeastSquares() ```score = scorer(shift_registration, X) ```
```Python
>> round(score, 3) 0.796
```

**See also:**

AmplitudePhaseDecomposition SobolevLeastSquares PairwiseCorrelation

1.2. Preprocessing
Methods

```python
__init__(return_stats=False, eval_points=None)
    Initialize the transformer

score_function(X, y)
    Compute the score of the transformation performed.
```

```python
__init__(return_stats=False, eval_points=None)
    Initialize the transformer

score_function(X, y)
    Compute the score of the transformation performed.
```

**Parameters**


**Returns**

Score of the transformation.

**Return type** float

### SobolevLeastSquares

class skfda.preprocessing.registration.validation.SobolevLeastSquares(eval_points=None)

Cross-validated measure of the registration procedure.

Computes a cross-validated measure of the level of synchronization [S11-5-2-3]:

\[
sls = 1 - \frac{\sum_{i=1}^{N} \int \left( \frac{\dot{f}_i(t)}{N} - \frac{1}{N} \sum_{j=1}^{N} \dot{f}_j \right)^2 dt}{\sum_{i=1}^{N} \int \left( \frac{\dot{f}_i(t)}{N} - \frac{1}{N} \sum_{j=1}^{N} \dot{f}_j \right)^2 dt}
\]

where \( \dot{f}_i \) and \( \dot{\tilde{f}}_i \) are the derivatives of the original and the registered data respectively.

This criterion measures the total cross-sectional variance of the derivatives of the aligned functions, relative to the original value. A value of 1 would indicate an identical shape for all registered curves, while zero corresponds to no improvement in the registration. It can be negative because the model can be arbitrarily worse.

**eval_points**

Set of points where the functions are evaluated to obtain a discrete representation and perform the calculation.

**Type** array_like, optional

**Parameters**

- `estimator` ([RegistrationTransformer](https://docs.scikit-fda.org/en/latest/api.html#scikit-fda-registration-transformers)) – Registration transformer.

**Note:** The original sobolev least square measure used in [S11-5-2-3] is defined as \(1 - sls\), but has been modified according to the scikit-learn scorers, where higher values correspond to better cross-validated measures.
Examples

Calculate the score of the shift registration of a sinusoidal process synthetically generated.

```python
>>> from skfda.preprocessing.registration.validation import ...
... SobolevLeastSquares
>>> from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process

>>> X = make_sinusoidal_process(error_std=0, random_state=0)

Fit the registration procedure.

>>> shift_registration = ShiftRegistration()
>>> shift_registration.fit(X)

Compute the sobolev least squares score.

```python
>>> scorer = SobolevLeastSquares()
>>> score = scorer(shift_registration, X)
>>> round(score, 3)
0.762
``` 

See also:

AmplitudePhaseDecomposition LeastSquares PairwiseCorrelation

Methods

```
__init__(eval_points) Initialize the transformer

score_function(X, y) Compute the score of the transformation performed.
```

```
__init__(eval_points=None)

Initialize the transformer

score_function(X, y)

Compute the score of the transformation performed.

Parameters

- X (FData) – Original functional data.
- y (FData) – Functional data registered.

Returns

Score of the transformation.

Return type

float

PairwiseCorrelation

```
class skfda.preprocessing.registration.validation.PairwiseCorrelation(eval_points=None)

Cross-validated measure of pairwise correlation between functions.

Computes a cross-validated pairwise correlation between functions to compare registration methods
\[ p_c = \frac{\sum_{i \neq j} cc(\tilde{f}_i(t), \tilde{f}_j(t))}{\sum_{i \neq j} cc(f_i(t), f_j(t))} \]

where \( f_i \) and \( \tilde{f}_i \) are the original and registered data respectively and \( cc(f, g) \) is the pairwise Pearson’s correlation between functions.

The larger the value of \( p_c \), the better the alignment between functions in general.

**eval_points**

Set of points where the functions are evaluated to obtain a discrete representation and perform the calculation.

**Type**: array_like, optional

**Parameters**

- **estimator** (`RegistrationTransformer`) – Registration transformer.
- **X** (`FData`) – Original functional data.
- **y** (`FData`) – Registered functional data.

**Note**: Pearson’s correlation between functions is calculated assuming the samples are equiespaciated.

**References**

**Examples**

Calculate the score of the shift registration of a sinusoidal process synthetically generated.

```python
>>> from skfda.preprocessing.registration.validation import ...
   PairwiseCorrelation
>>> from skfda.preprocessing.registration import ShiftRegistration
>>> from skfda.datasets import make_sinusoidal_process
>>> X = make_sinusoidal_process(error_std=0, random_state=0)
```

Fit the registration procedure.

```python
>>> shift_registration = ShiftRegistration()
>>> shift_registration.fit(X)
```

Compute the pairwise correlation score.

```python
>>> scorer = PairwiseCorrelation()
>>> score = scorer(shift_registration, X)
>>> round(score, 3)
1.816
```

**See also**:

AmplitudePhaseDecomposition LeastSquares SobolevLeastSquares

**Methods**
__init__(eval_points)
Initialize the transformer

score_function(X, y)
Compute the score of the transformation performed.

__init__(eval_points=None)
Initialize the transformer

score_function(X, y)
Compute the score of the transformation performed.

Parameters
- X (FData) – Original functional data.
- y (FData) – Functional data registered.

Returns Score of the transformation.
Return type float

Warping utils
This module contains some functions related with the warping of functional data.

skfda.preprocessing.registration.invert_warping(...) Compute the inverse of a diffeomorphism.

skfda.preprocessing.registration.normalize_warping(warping) Rescale a warping to normalize their domain.

invert_warping

skfda.preprocessing.registration.invert_warping(fdatagrid, *, output_points=None)
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
We will construct the warping \( \gamma : [0, 1] \rightarrow [0, 1] \) which maps \( t \) to \( t^3 \).

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

We will compute the inverse.

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

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

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

**Examples using skfda.preprocessing.registration.invert_warping**

- **Pairwise alignment**

**normalize_warping**

skfda.preprocessing-registration.normalize_warping(warping, domain_range=None)

Rescale a warping to normalize their domain.

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

**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)

**References**

1.2.3 Smoothing

If the functional data observations are noisy, smoothing the data allows a better representation of the true underlying functions. You can learn more about the smoothing methods provided by scikit-fda here.

1.2.4 Registration

Sometimes, the functional data may be misaligned, or the phase variation should be ignored in the analysis. To align the data and eliminate the phase variation, we need to use registration methods. Here you can learn more about the registration methods available in the library.

1.3 Exploratory analysis

The exploratory analysis package deals with techniques to summarize, interpret and visualize functional data.

1.3.1 Visualization

The visualization package provides tools to show different views of the functional data, that highlight several important aspects of it.

Functional Data Boxplot

Classes to construct the functional data boxplot. Only supported for functional data with domain dimension 1 or 2 and as many dimensions on the image as required.

If the dimension of the domain is 1, the following class must be used. See the Boxplot example for detailed explanation.

```
```

Boxplot

```
class skfda.exploratory.visualization.Boxplot(fdatagrid, depth_method=<function modified_band_depth>, prob=[0.5], factor=1.5)
```

Representation of the functional boxplot.

Class implementing the functional boxplot which is an informative exploratory tool for visualizing functional data, as well as its generalization, the enhanced functional boxplot. Only supports 1 dimensional domain functional data.

Based on the center outward ordering induced by a depth measure for functional data, the descriptive statistics of a functional boxplot are: the envelope of the 50% central region, the median curve, and the maximum non-outlying envelope. In addition, outliers can be detected in a functional boxplot by the
1.5 times the 50% central region empirical rule, analogous to the rule for classical boxplots.

**fdatagrid**
Object containing the data.

**median**
contains the median/s.

**central_envelope**
contains the central envelope/s.

**non_outlying_envelope** (array, (fdatagrid.dim_codomain, 2, nsample_points)): contains the non-outlying envelope/s.

**colormap**
Colormap from which the colors to represent the central regions are selected.

**envelopes** (array, (fdatagrid.dim_codomain * ncentral_regions, 2, nsample_points)): contains the region envelopes.

**outliers**
contains the outliers.

**barcol**
Color of the envelopes and vertical lines.

**outliercol**
Color of the outliers.

**mediancol**
Color of the median.

**show_full_outliers**
If False (the default) then only the part outside the box is plotted. If True, complete outlying curves are plotted.

**Example**

Function \( f : \mathbb{R} \rightarrow \mathbb{R} \).

```python
>>> from skfda import FDataGrid
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
...                 [0.5, 0.5, 1, 2, 1.5, 1],
...                 [-1.0, -1.0, -0.5, 1.0, 1.0, 0.5],
...                 [-0.5, -0.5, -0.5, -1.0, -1.0, -1.0]]
```
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = FDataGrid(data_matrix, sample_points, dataset_label="dataset",
... axes_labels=["x_label", "y_label"]
>>> Boxplot(fd)

Boxplot(
    FDataGrid=FDataGrid(
        array([[ 1. ],
               [ 2. ],
               [ 3. ],
               [ 2.5],
               [ 2. ]],
               [[ 0.5],
                [ 0.5],
                [ 1. ],
                [ 2. ],
                [ 1.5],
                [ 1. ]],
               [[-1. ],
                [-1. ],
                [-0.5],
                [ 1. ],
                [ 1. ],
                [ 0.5]],
               [[-0.5],
                [-0.5],
                [-0.5],
                [-1. ],
                [-1. ],
                [-1. ]]),
        sample_points=array([0, 2, 4, 6, 8, 10]),
        domain_range=array([[0, 10]]),
        dataset_label='dataset',
        axes_labels=['x_label', 'y_label'],
        extrapolation=None,
        interpolator=SplineInterpolator(interpolation_order=1,
        smoothness_parameter=0.0, monotone=False),
        keepdims=False),
    median=array([[ 0.5],
                  [ 0.5],
                  [ 1. ],
                  [ 2. ],
                  [ 1.5],
                  [ 1. ]]),
    central_envelope=(array([[-1. ],
                              [-1. ],
                              [-0.5],
                              [ 1. ],
                              [ 1. ],
                              [ 0.5]]), array([[ 0.5],
                   [ 0.5],
                   [ 1. ]]),

(continues on next page)
Methods

```python
__init__(fdatagrid[, depth_method, prob, factor])
```
Initialization of the Boxplot class.

```python
plot([chart, fig, axes, n_rows, n_cols])
```
Visualization of the functional boxplot of the `fdatagrid` (dim_domain=1).

```python
__init__(fdatagrid, depth_method=<function modified_band_depth>, prob=[0.5], factor=1.5)
```
Initialization of the Boxplot class.

Parameters

- **fdatagrid** (FDataGrid) – Object containing the data.
- **depth_method** (depth measure, optional) – Method used to order the data. Defaults to modified band depth.
- **prob** (list of float, optional) – List with float numbers (in the range from 1 to 0) that indicate which central regions to represent. Defaults to [0.5] which represents the 50% central region.
- **factor** (double) – Number used to calculate the outlying envelope.
Attributes

```
central_envelope
colormap
envelopes
factor
fdatagrid
median
non_outlying_envelope
outliers
show_full_outliers
```

```
plot(chart=None, *, fig=None, axes=None, n_rows=None, n_cols=None)
```

Visualization of the functional boxplot of the fdatagrid (dim_domain=1).

Parameters

- **fig (figure object, optional)** – figure over with the graphs are plotted in case ax is not specified. If None and ax is also None, the figure is initialized.
- **axes (list of axis objects, optional)** – axis over where the graphs are plotted. If None, see param fig.
- **n_rows (int, optional)** – designates the number of rows of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- **n_cols (int, optional)** – designates the number of columns of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.

Returns

- **fig (figure)**

Examples using `skfda.exploratory.visualization.Boxplot`

- **Surface Boxplot**
- **Boxplot**

If the dimension of the domain is 2, this one. See the **Surface Boxplot** example for detailed explanation.

```
skfda.exploratory.visualization.SurfaceBoxplot(...)  Representation of the surface boxplot.
```

### SurfaceBoxplot

```
class skfda.exploratory.visualization.SurfaceBoxplot(fdatagrid, method=<function modified_band_depth>, factor=1.5)
```

Representation of the surface boxplot.

Class implementing the surface boxplot. Analogously to the functional boxplot, it is an informative exploratory tool for visualizing functional data with domain dimension 2. Nevertheless, it does not implement the enhanced surface boxplot.

Based on the center outward ordering induced by a **depth measure** for functional data, it represents...
the envelope of the 50% central region, the median curve, and the maximum non-outlying envelope.

**fdatagrid**  
Object containing the data.  
Type *FDataGrid*

**median**  
contains the median/s.  
Type array, *(fdatagrid.dim_codomain, lx, ly)*

**central_envelope**  
contains the central envelope/s.  
Type array, *(fdatagrid.dim_codomain, 2, lx, ly)*

**non_outlying_envelope**  
contains the non-outlying envelope/s.  
Type array, *(fdatagrid.dim_codomain, 2, lx, ly)*

**colormap**  
Colormap from which the colors to represent the central regions are selected.  
Type *matplotlib.colors.LinearSegmentedColormap*

**boxcol**  
Color of the box, which includes median and central envelope.  
Type string

**outcol**  
Color of the outlying envelope.  
Type string

**Example**

Function \( f : \mathbb{R}^p \mapsto \mathbb{R} \).  

```python  
>>> from skfda import FDataGrid  
>>> data_matrix = [[[1], [0.7], [1]],  
... [[4], [0.4], [5]],  
... [[2], [0.5], [2]],  
... [[3], [0.6], [3]]]  
>>> sample_points = [[2, 4], [3, 6, 8]]  
>>> fd = FDataGrid(data_matrix, sample_points, dataset_label="dataset",  
... axes_labels=["x1_label", "x2_label", "y_label"])  
>>> SurfaceBoxplot(fd)  
SurfaceBoxplot(  
    FDataGrid=FDataGrid(  
        array([[ 1.],  
                [ 0.7],  
                [ 1.]],  
                [[ 4.],  
                [ 0.4],  
                [ 5.]],  
                [[ 2.],  
```

(continues on next page)
[0.5],
[2.],
[[3.],
[0.6],
[3.]]),
sample_points=[array([[2, 4]], array([[3, 6, 8]]),
domain_range=array([[2, 4],
[3, 8]]),
dataset_label='dataset',
axes_labels=['x1_label', 'x2_label', 'y_label'],
extrapolation=None,
interpolator=SplineInterpolator(interpolation_order=1,
smoothness_parameter=0.0, monotone=False),
keepdims=False),
median=array([[1.],
[0.7],
[1.]],
[[4.],
[0.4],
[5.]]),
central envelope=(array([[1.],
[0.7],
[1.]],
[[4.],
[0.4],
[5.]]),
array([[1.],
[0.7],
[1.]],
[[4.],
[0.4],
[5.]])),
outlying envelope=(array([[1.],
[0.7],
[1.]],
[[4.],
[0.4],
[5.]]),
array([[1.],
[0.7],
[1.]],
[[4.],
[0.4],
[5.]])))

Methods

**__init__**(fdatagrid[, method, factor]) Initialization of the functional boxplot.

**plot**(chart, fig, axes, n_rows, n_cols) Visualization of the surface boxplot of the fdatagrid (dim_domain=2).

1.3. Exploratory analysis
__init__ (fdatagrid, method=<function modified_band_depth>, factor=1.5)

Initialization of the functional boxplot.

Parameters

- **fdatagrid (FDataGrid)** – Object containing the data.
- **method (depth measure, optional)** – Method used to order the data. Defaults to modified band depth.
- **prob (list of float, optional)** – List with float numbers (in the range from 1 to 0) that indicate which central regions to represent. Defaults to [0.5] which represents the 50% central region.
- **factor (double)** – Number used to calculate the outlying envelope.

Attributes

- **boxcol**
- **central_envelope**
- **colormap**
- **factor**
- **fdatagrid**
- **median**
- **non_outlying_envelope**
- **outcol**

plot (chart=None, *, fig=None, axes=None, n_rows=None, n_cols=None)

Visualization of the surface boxplot of the fdatagrid (dim_domain=2).

Args:

- **fig (figure object, optional)**: figure over with the graphs are plotted in case ax is not specified. If None and ax is also None, the figure is initialized.
- **axes (list of axis objects, optional)**: axis over where the graphs are plotted. If None, see param fig.
- **n_rows (int, optional)**: designates the number of rows of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- **n_cols (int, optional)**: designates the number of columns of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.

Returns figure object in which the graphs are plotted.

Return type fig (figure)

Examples using skfda.exploratory.visualization.SurfaceBoxplot

- **Surface Boxplot**

Magnitude-Shape Plot

The Magnitude-Shape Plot is implemented in the MagnitudeShapePlot class.
The `MagnitudeShapePlot` needs both the mean and the variation of the directional outlyingness of the samples, which is calculated using `directional_outlyingness_stats()`.

Once the points assigned to each of the samples are obtained from the above function, an outlier detection method is implemented. The results can be shown calling the `plot()` method of the class.

```python
skfda.exploratory.visualization.MagnitudeShapePlot(...)
```

**Implementation of the magnitude-shape plot**

This plot, which is based on the calculation of the directional outlyingness of each of the samples, serves as a visualization tool for the centrality of curves. Furthermore, an outlier detection procedure is included.

The norm of the mean of the directional outlyingness ($\|\text{MO}\|$) is plotted in the x-axis, and the variation of the directional outlyingness ($V_O$) in the y-axis.

The outliers are detected using an instance of `DirectionalOutlierDetector`.

**fdatagrid**

Object to be visualized.

_Type_ `FDataGrid`

**depth_method**

Method used to order the data. Defaults to modified band depth.

_Type_ `depth measure`, optional

**pointwise_weights**

an array containing the weights of each points of discretisation where values have been recorded.

_Type_ `array_like`, optional

**alpha**

Denotes the quantile to choose the cutoff value for detecting outliers. Defaults to 0.993, which is used in the classical boxplot.

_Type_ `float`, optional

**points**

2-dimensional matrix where each row contains the points plotted in the graph.

_Type_ `numpy.ndarray`

**outliers**

Contains 1 or 0 to denote if a sample is an outlier or not, respectively.

_Type_ `1-D array`, (fdatagrid.n_samples,)

**colormap**

Colormap from which the colors of the plot are extracted. Defaults to ‘seismic’.

_Type_ `matplotlib.pyplot.LinearSegmentedColormap`, optional

**color**

Tone of the colormap in which the nonoutlier points are plotted. Defaults to 0.2.
**Type** float, optional

`outliercol`
Tone of the colormap in which the outliers are plotted. Defaults to 0.8.

**Type** float, optional

`xlabel`
Label of the x-axis. Defaults to ‘MO’, mean of the directional outlyingness.

**Type** string, optional

`ylabel`
Label of the y-axis. Defaults to ‘VO’, variation of the directional outlyingness.

**Type** string, optional

`title`
Title of the plot. defaults to ‘MS-Plot’.

**Type** string, optional

**Example**

```python
>>> import skfda
>>> from skfda.exploratory.depth import modified_band_depth
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
...                 [0.5, 0.5, 1, 2, 1.5, 1],
...                 [-1, -1, -0.5, 1, 1, 0.5],
...                 [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> MagnitudeShapePlot(fd)
MagnitudeShapePlot(
    FDataGrid=FDataGrid(
        array([[ 1.],
                [ 2.],
                [ 3.],
                [ 2.5],
                [ 2.]],
           [[ 0.5],
             [ 0.5],
             [ 1.],
             [ 2.],
             [ 1.5],
             [ 1.]],
           [[-1. ],
            [-1. ],
            [-0.5],
            [ 1. ],
            [ 1. ],
            [ 0.5]],
           [[-0.5],
            [-0.5],
            [-0.5],
            [-0.5],
            [-0.5]],
```
Methods

```python
__init__(fdatagrid, **kwargs) Initialization of the MagnitudeShapePlot class.

plot([chart, fig, axes]) Visualization of the magnitude shape plot of the fdatagrid.

__init__(fdatagrid, **kwargs) Initialization of the MagnitudeShapePlot class.
```

**Parameters**

- `fdatagrid` (`FDataGrid`) – Object containing the data.
- `depth_method` (`depth measure`, optional) – Method used to order the data. Defaults to projection depth.
- `pointwise_weights` (`array_like`, optional) – an array containing the weights of each points of discretisati on where values have been recorded.
- `alpha` (`float`, optional) – Denotes the quantile to choose the cutoff value for detecting outliers. Defaults to 0.993, which is used in the classical boxplot.
- `assume_centered` (`boolean`, optional) – If True, the support of the robust location and the covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, default value, the robust location and covariance are directly computed with the FastMCD algorithm.

---

1.3. Exploratory analysis 125
without additional treatment.

- **support_fraction** (float, 0 < support_fraction < 1, optional) – The proportion of points to be included in the support of the raw MCD estimate. Default is None, which implies that the minimum value of support_fraction will be used within the algorithm: \([n_{\text{sample}} + n_{\text{features}} + 1] / 2\)

- **random_state** (int, RandomState instance or None, optional) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. By default, it is 0.

**Attributes**

- alpha
- color
- colormap
- depth_method
- fdatagrid
- outliercol
- outliers
- points
- pointwise_weights

**plot** (chart=None, *, fig=None, axes=None)
Visualization of the magnitude shape plot of the fdatagrid.

- Parameters ax (axes object, optional) – axes over where the graph is plotted. Defaults to matplotlib current axis.
- Returns figure object in which the graph is plotted.
- Return type fig (figure object)

**Examples using** skfda.exploratory.visualization.MagnitudeShapePlot

- **Magnitude-Shape Plot**
- **Magnitude-Shape Plot synthetic example**

**Clustering Plots**

In order to show the results of the cluster algorithms in a visual way, this module is implemented. It contains the following methods:

- **skfda.exploratory.visualization.clustering.plot_clusters(...)**
  Plot of the FDataGrid samples by clusters.

- **skfda.exploratory.visualization.clustering.plot_cluster_lines(...)**
  Implementation of the plotting of the results of the Fuzzy K-Means method.

- **skfda.exploratory.visualization.clustering.plot_cluster_bars(...)**
  Implementation of the plotting of the results of the Fuzzy K-Means method.
plot_clusters

skfda.exploratory.visualization.clustering.plot_clusters(estimator, X, chart=None, fig=None, axes=None, n_rows=None, n_cols=None, sample_labels=None, cluster_colors=None, cluster_labels=None, center_colors=None, center_labels=None, center_width=3, colormap=matplotlib.colors.LinearSegmentedColormap object)

Plot of the FDataGrid samples by clusters.

The clusters are calculated with the estimator passed as a parameter. If the estimator is not fitted, the fit method is called. Once each sample is assigned a label the plotting can be done. Each group is assigned a color described in a legend.

Parameters

- **estimator** (*BaseEstimator object*) – estimator used to calculate the clusters.
- **X** (*FDataGrd object*) – contains the samples which are grouped into different clusters.
- **fig** (*figure object*) – figure over which the graphs are plotted in case ax is not specified. If None and ax is also None, the figure is initialized.
- **axes** (*list of axis objects*) – axis over where the graphs are plotted. If None, see param fig.
- **n_rows** (*int*) – designates the number of rows of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- **n_cols** (*int*) – designates the number of columns of the figure to plot the different dimensions of the image. Only specified if fig and ax are None.
- **sample_labels** (*list of str*) – contains in order the labels of each sample.
- **cluster_colors** (*list of colors*) – contains in order the colors of each cluster the samples of the fdatagrid are classified into.
- **cluster_labels** (*list of str*) – contains in order the names of each cluster the samples of the fdatagrid are classified into.
- **center_colors** (*list of colors*) – contains in order the colors of each centroid of the clusters the samples of the fdatagrid are classified into.
- **center_labels** (*list of str*) – contains in order the labels of each centroid of the clusters the samples of the fdatagrid are classified into.
- **center_width** (*int*) – width of the centroid curves.
- **colormap** (*colormap*) – colormap from which the colors of the plot are taken. Defaults to *rainbow*.

Returns

tuple containing:
fig (figure object): figure object in which the graphs are plotted in case ax is None.

ax (axes object): axes in which the graphs are plotted.

Return type (tuple)

Examples using `skfda.exploratory.visualization.clustering.plot_clusters`

- Clustering

`plot_cluster_lines`

```
skfda.exploratory.visualization.clustering.plot_cluster_lines(estimator, X, chart=None, fig=None, axes=None, sample_colors=None, sample_labels=None, cluster_labels=None, colormap=<matplotlib.colormap object>, x_label=None, y_label=None, title=None)
```

Implementation of the plotting of the results of the Fuzzy K-Means method.

A kind of Parallel Coordinates plot is generated in this function with the membership values obtained from the algorithm. A line is plotted for each sample with the values for each cluster. See Clustering Example.

Parameters

- `estimator (BaseEstimator object)` – estimator used to calculate the clusters.
- `X (FDaGrd object)` – contains the samples which are grouped into different clusters.
- `fig (figure object, optional)` – figure over which the graph is plotted in case ax is not specified. If None and ax is also None, the figure is initialized.
- `axes (axes object, optional)` – axis over where the graph is plotted. If None, see param fig.
- `sample_colors (list of colors, optional)` – contains in order the colors of each sample of the fdatagrid.
- `sample_labels (list of str, optional)` – contains in order the labels of each sample of the fdatagrid.
- `cluster_labels (list of str, optional)` – contains in order the names of each cluster the samples of the fdatagrid are classified into.
- `colormap (colormap, optional)` – colormap from which the colors of the plot are taken.
- `x_label (str)` – Label for the x-axis. Defaults to “Cluster”.
- `y_label (str)` – Label for the y-axis. Defaults to “Degree of membership”.
- `title (str, optional)` – Title for the figure where the clustering results are plotted. Defaults to “Degrees of membership of the samples to each cluster”.

128 Chapter 1. API Reference
Returns
tuple containing:

fig (figure object): figure object in which the graphs are plotted in
case ax is None.

ax (axes object): axes in which the graphs are plotted.

Return type  (tuple)

Examples using skfda.exploratory.visualization.clustering.plot_cluster_lines

- Clustering

plot_cluster_bars

skfda.exploratory.visualization.clustering.plot_cluster_bars(estimator,  X,  chart=None,
fig=None,  axes=None,
sort=-1,  sample_labels=None,
cluster_colors=None,  cluster_labels=None,
colormap=<matplotlib.colors.LinearSegmentedColormap object>,  x_label=None,
y_label=None,  title=None)

Implementation of the plotting of the results of the Fuzzy K-Means method.

A kind of barplot is generated in this function with the membership values obtained from the algorithm.
There is a bar for each sample whose height is 1 (the sum of the membership values of a sample add to 1), and the part proportional to each cluster is coloured with the corresponding color. See Clustering Example.

Parameters

- estimator (BaseEstimator object) – estimator used to calculate the clusters.
- X (FDataGrd object) – contains the samples which are grouped into different clusters.
- fig (figure object, optional) – figure over which the graph is plotted in case ax
  is not specified. If None and ax is also None, the figure is initialized.
- axes (axes object, optional) – axes over where the graph is plotted. If None,
  see param fig.
- sort (int, optional) – Number in the range [-1, n_clusters) designating the cluster
  whose labels are sorted in a decrementsing order. Defaults to -1, in this case, no
  sorting is done.
- sample_labels (list of str, optional) – contains in order the labels of each
  sample of the fdatagrid.
- cluster_labels (list of str, optional) – contains in order the names of each
  cluster the samples of the fdatagrid are classified into.
- **cluster_colors** *(list of colors)* – contains in order the colors of each cluster the samples of the fdatagrid are classified into.
- **colormap** *(colormap, optional)* – colormap from which the colors of the plot are taken.
- **x_label** *(str)* – Label for the x-axis. Defaults to “Sample”.
- **y_label** *(str)* – Label for the y-axis. Defaults to “Degree of membership”.
- **title** *(str)* – Title for the figure where the clustering results are plotted. Defaults to “Degrees of membership of the samples to each cluster”.

**Returns**

tuple containing:

- **fig** *(figure object)*: figure object in which the graph is plotted in case ax is None.
- **ax** *(axis object)*: axis in which the graph is plotted.

**Return type** *(tuple)*

**Examples using** `skfda.exploratory.visualization.clustering.plot_cluster_bars`

- **Clustering**

  In the first one, the samples of the FDataGrid are divided by clusters which are assigned different colors. The following functions, are only valid for the class `FuzzyKMeans` to see the results graphically in the form of a parallel coordinates plot or a barplot respectively.

  See Clustering Example for detailed explanation.

**1.3.2 Depth Measures**

Functions to order functional data.

Each sample of the dataset is assigned a number between 0 and 1. Larger values correspond to more centered samples and smaller ones to those samples more outward.

```python
skfda.exploratory.depth.band_depth(fdatagrid, *)
```

Implementation of Band Depth for functional data.

```python
skfda.exploratory.depth.modified_band_depth(*)
```

Implementation of Modified Band Depth for functional data.

```python
skfda.exploratory.depth.fraiman_muniz_depth(*)
```

Implementation of Fraiman and Muniz (FM) Depth for functional data.

**band_depth**

```python
skfda.exploratory.depth.band_depth(fdatagrid, *, pointwise=False)
```

Implementation of Band Depth for functional data.

The band depth of each sample is obtained by computing the fraction of the bands determined by two sample curves containing the whole graph of the first one. In the case the fdatagrid domain dimension is 2, instead of curves, surfaces determine the bands. In larger dimensions, the hyperplanes determine the bands.
Parameters

- `fdatagrid (FDataGrid)` – Object over whose samples the band depth is going to be calculated.
- `pointwise (boolean, optional)` – Indicates if the pointwise depth is returned instead. Defaults to False.

Returns

Array containing the band depth of the samples, or the band depth of the samples at each point of discretization if pointwise equals to True.

Return type `depth (numpy.darray)`

Examples

```python
>>> import skfda

>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
...                 [0.5, 0.5, 1, 2, 1.5, 1],
...                 [-1, -1, -0.5, 1, 1, 0.5],
...                 [-0.5, -0.5, -0.5, -1, -1, -1]]

>>> sample_points = [0, 2, 4, 6, 8, 10]

>>> fd = skfda.FDataGrid(data_matrix, sample_points)

>>> band_depth(fd)
array([ 0.5 , 0.83333333, 0.5 , 0.5 ])
```

Examples using `skfda.exploratory.depth.band_depth`

- Boxplot

`modified_band_depth`

`skfda.exploratory.depth.modified_band_depth(fdatagrid, *, pointwise=False)`

Implementation of Modified Band Depth for functional data.

The band depth of each sample is obtained by computing the fraction of time its graph is contained in the bands determined by two sample curves. In the case the fdatagrid domain dimension is 2, instead of curves, surfaces determine the bands. In larger dimensions, the hyperplanes determine the bands.

Parameters

- `fdatagrid (FDataGrid)` – Object over whose samples the modified band depth is going to be calculated.
- `pointwise (boolean, optional)` – Indicates if the pointwise depth is returned instead. Defaults to False.

Returns

Array containing the modified band depth of the samples, or the modified band depth of the samples at each point of discretization if pointwise equals to True.

Return type `depth (numpy.darray)`
Examples

```python
>>> import skfda
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
    ... [0.5, 0.5, 1, 2, 1.5, 1],
    ... [-1, -1, -0.5, 1, 1, 0.5],
    ... [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> depth = modified_band_depth(fd)
>>> depth.round(2)
array([ 0.5 , 0.83, 0.72, 0.67])
>>> pointwise = modified_band_depth(fd, pointwise = True)
>>> pointwise.round(2)
array([[[ 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
        [ 0.83, 0.83, 0.83, 0.83, 0.83, 0.83]],
        [ 0.5 , 0.5 , 0.83, 0.83, 0.83, 0.83],
        [ 0.83, 0.83, 0.83, 0.5 , 0.5 , 0.5 ]])
```

Examples using `skfda.exploratory.depth.modified_band_depth`

- **Magnitude-Shape Plot**

fraiman_muniz_depth

`skfda.exploratory.depth.fraiman_muniz_depth(fdatagrid, *, pointwise=False)`

Implementation of Fraiman and Muniz (FM) Depth for functional data.

Each column is considered as the samples of an aleatory variable. The univariate depth of each of the samples of each column is calculated as follows:

\[
D(x) = 1 - \left| \frac{1}{2} - F(x) \right|
\]

Where \( F \) stands for the marginal univariate distribution function of each column.

The depth of a sample is the result of integrating the previously computed depth for each of its points and normalizing dividing by the length of the interval.

Parameters

- `fdagrid (FDataGrid)`: Object over whose samples the FM depth is going to be calculated.
- `pointwise (boolean, optional)`: Indicates if the pointwise depth is returned instead. Defaults to False.

Returns

Array containing the Fraiman-Muniz depth of the samples, or the Fraiman-Muniz of the samples at each point of discretization if pointwise equals to True.

Return type depth (numpy.array)
Examples

Currently, this depth function can only be used for univariate functional data:

```python
>>> import skfda

>>> data_matrix = 
[[1, 1, 2, 3, 2.5, 2],
 ... [0.5, 0.5, 1, 2, 1.5, 1],
 ... [-1, -1, -0.5, 1, 1, 0.5],
 ... [-0.5, -0.5, -0.5, -1, -1, -1]]

>>> sample_points = [0, 2, 4, 6, 8, 10]

>>> fd = skfda.FDataGrid(data_matrix, sample_points)

>>> fraiman_muniz_depth(fd)
array([0.5 , 0.75 , 0.925, 0.875])
```

You can use `pointwise` to obtain the pointwise depth, before the integral is applied.

```python
>>> pointwise = fraiman_muniz_depth(fd, pointwise=True)

>>> pointwise
array([[ 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
       [ 0.75, 0.75, 0.75, 0.75, 0.75, 0.75],
       [ 0.75, 0.75, 1. , 1. , 1. , 1. ],
       [ 1. , 1. , 1. , 0.75, 0.75, 0.75]])
```

Examples using `skfda.exploratory.depth.fraiman_muniz_depth`

- Boxplot
- Magnitude-Shape Plot

The possibility of obtaining the ordering of each point of the sample (compared to the other samples) is given if a parameter is specified in the functions.

All of them support multivariate functional data, with more than one dimension on the image and on the domain.

Outlyingness conversion to depth

The concepts of depth and outlyingness are (inversely) related. A deeper datum is less likely an outlier. Conversely, a datum with very low depth is possibly an outlier. In order to convert an outlying measure to a depth measure the following convenience function is provided.

```python
skfda.exploratory.depth.outlyingness_to_depth
```

```
Convert outlyingness function to depth function.

```

**outlyingness_to_depth**

```
skfda.exploratory.depth.outlyingness_to_depth(outlyingness, *, supreme=None)
```

Convert outlyingness function to depth function.
An outlyingness function $O(x)$ can be converted to a depth function as

$$D(x) = \frac{1}{1 + O(x)}$$

if $O(x)$ is unbounded or as

$$D(x) = 1 - \frac{O(x)}{\sup O(x)}$$

if $O(x)$ is bounded ([Se06]).

**Parameters**

- `outlyingness (Callable)` – Outlyingness function.
- `supreme (float, optional)` – Supreme value of the outlyingness function.

**Returns** The corresponding depth function.

**Return type** Callable

**References**

**Multivariate depths**

Some utilities, such as the `MagnitudeShapePlot` require computing a non-functional (multivariate) depth pointwise. Thus we also provide some multivariate depth functions.

```python
skfda.exploratory.depth.multivariate.projection_depth(X, *)
```

Returns the projection depth.

**projection_depth**

```python
skfda.exploratory.depth.multivariate.projection_depth(X, *, pointwise=False)
```

Returns the projection depth.

The projection depth is the depth function associated with the Stagel-Donoho outlyingness.

### 1.3.3 Outlier detection

Functional outlier detection is the identification of functions that do not seem to behave like the others in the dataset. There are several ways in which a function may be different from the others. For example, a function may have a different shape than the others, or its values could be more extreme. Thus, outlyingness is difficult to categorize exactly as each outlier detection method looks at different features of the functions in order to identify the outliers.

Each of the outlier detection methods in scikit-fda has the same API as the outlier detection methods of scikit-learn.

**Interquartile Range Outlier Detector**

One of the most common ways of outlier detection is given by the functional data boxplot. An observation is marked as an outlier if it has points $1.5 \cdot IQR$ times outside the region containing the deepest 50% of the
curves (the central region), where \( IQR \) is the interquartile range.

### skfda.exploratory.outliers.

**IQROutlierDetector**

**class skfda.exploratory.outliers.IQROutlierDetector(*, depth_method=<function modified_band_depth>, factor=1.5)**

Outlier detector using the interquartile range.

Detects as outliers functions that have one or more points outside \( \text{factor} \) times the interquartile range plus or minus the central envelope, given a functional depth measure. This corresponds to the points selected as outliers by the functional boxplot.

**Parameters**

- **depth_method** (*Callable*) – The functional depth measure used.
- **factor** (*float*) – The number of times the IQR is multiplied.

**Example**

Function \( f : \mathbb{R} \rightarrow \mathbb{R} \).

```python
>>> import skfda
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
                 ... [0.5, 0.5, 1, 2, 1.5, 1],
                 ... [-1, -1, -0.5, 1, 1, 0.5],
                 ... [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> out_detector = IQROutlierDetector()
>>> out_detector.fit_predict(fd)
array([-1, 1, 1, -1])
```

**Methods**

```python
__init__(*[, depth_method, factor]) Initialize self.
fit(X[, y]) Perform fit on X and returns labels for X.
fit_predict(X[, y]) Perform fit on X and returns labels for X.
get_params([deep]) Get parameters for this estimator.
predict(X) Predict targets for X.
set_params(**params) Set the parameters of this estimator.
```

**__init__(*[, depth_method, factor])** Initialize self. See help(type(self)) for accurate signature.

**fit_predict(X, y=None)** Perform fit on X and returns labels for X.

1.3. Exploratory analysis 135
Parameters

- **X** (*ndarray, shape (n_samples, n_features)*) – Input data.
- **y** (*Ignored*) – Not used, present for API consistency by convention.

Returns **y** – 1 for inliers, -1 for outliers.

Return type *ndarray, shape (n_samples,)*

**get_params**(deep=True)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type *mapping of string to any*

**set_params**(**params**)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

Parameters **params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type *object*

**DirectionalOutlierDetector**

Other more novel way of outlier detection takes into account the magnitude and shape of the curves. Curves which have a very different shape or magnitude are considered outliers.

**skfda.exploratory.outliers.**

**DirectionalOutlierDetector**(())

Outlier detector using directional outlyingness.

**class skfda.exploratory.outliers.DirectionalOutlierDetector**((), **depth_method=<function projection_depth>, pointwise_weights=0, assume_centered=False, support_fraction=None, num_resamples=1000, random_state=0, alpha=0.993, _force_asymptotic=False)

Outlier detector using directional outlyingness.

Considering **Y** = \( \left( \text{MO}^T, \text{VO} \right)^T \), the outlier detection method is implemented as described below.

First, the square robust Mahalanobis distance is calculated based on a sample of size \( h \leq fdatagrid.n_{samples} \):

\[
RMD^2 (Y, Y^*_j) = (Y - Y^*_j)^T S_j^{-1} (Y - Y^*_j)
\]
where $J$ denotes the group of $h$ samples that minimizes the determinant of the corresponding covariance matrix, $Y^*_{J} = h^{-1} \sum_{i \in J} Y_i$ and $S^*_{J} = h^{-1} \sum_{i \in J} (Y_i - Y^*_{J}) (Y_i - Y^*_{J})^T$. The sub-sample of size $h$ controls the robustness of the method.

Then, the tail of this distance distribution is approximated as follows:

$$\frac{c(m - p)}{m(p + 1)} \text{RMD}^2(Y, Y^*_{J}) \sim F_{p+1, m-p}$$

where $p$ is the dimension of the image plus one, and $c$ and $m$ are parameters determining the degrees of freedom of the $F$-distribution and the scaling factor, given by empirical results and an asymptotic formula.

Finally, we choose a cutoff value to determine the outliers, $C$, as the $\alpha$ quantile of $F_{p+1, m-p}$. We set $\alpha = 0.993$, which is used in the classical boxplot for detecting outliers under a normal distribution.

**Parameters**

- **depth_method** (*depth measure*, optional) – Method used to order the data. Defaults to projection depth.
- **pointwise_weights** (*array_like*, optional) – an array containing the weights of each points of discretisati on where values have been recorded.
- **alpha** (*float*, optional) – Denotes the quantile to choose the cutoff value for detecting outliers Defaults to 0.993, which is used in the classical boxplot.
- **assume_centered** (*boolean*, optional) – If True, the support of the robust location and the covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, default value, the robust location and covariance are directly computed with the FastMCD algorithm without additional treatment.
- **support_fraction** (*float, 0 < support_fraction < 1, optional*) – The proportion of points to be included in the support of the raw MCD estimate. Default is None, which implies that the minimum value of support_fraction will be used within the algorithm: $[n\_sample + n\_features + 1] / 2$
- **random_state** (*int, RandomState instance or None, optional*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. By default, it is 0.

**Example**

Function $f : \mathbb{R} \rightarrow \mathbb{R}$.

```python
>>> import skfda
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
                 [0.5, 0.5, 1, 2, 1.5, 1],
                 [-1, -1, -0.5, 1, 1, 0.5],
                 [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> out_detector = DirectionalOutlierDetector()
>>> out_detector.fit_predict(fd)
array([[1, 1, 1, 1]])
```
## References


## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(*)[, depth_method, ...]</code></td>
<td>Initialize self.</td>
</tr>
<tr>
<td><code>fit_predict(X[, y])</code></td>
<td>Perform fit on X and returns labels for X.</td>
</tr>
<tr>
<td><code>get_params(deep)</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

__init__(*)

```
* depth_method=<function projection_depth>,
  pointwise_weights=None,
  assume_centered=False,
  support_fraction=None,
  num_resamples=1000,
  random_state=0,
  alpha=0.993,
  _force_asymptotic=False
```

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

**`fit_predict(X, y=None)`**

Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

**Parameters**

- **X** *(ndarray, shape (n_samples, n_features)) – Input data.*
- **y** *(Ignored) – Not used, present for API consistency by convention.*

**Returns**

- **y** – 1 for inliers, -1 for outliers.
- **Return type** ndarray, shape (n_samples,)

**`get_params(deep=True)`**

Get parameters for this estimator.

**Parameters**

- **deep** *(bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.*

**Returns**

- **params** – Parameter names mapped to their values.
- **Return type** mapping of string to any

**`set_params(**params)`**

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>_<parameter>` so that it’s possible to update each component of a nested object.

**Parameters**

- ****params** *(dict) – Estimator parameters.*

**Returns**

- **self** – Estimator instance.
- **Return type** object

For this method, it is necessary to compute the mean and variation of the directional outlyingness, which can be done with the following function.
`skfda.exploratory.outliers.directional_outlyingness_stats(...)` Computes the directional outlyingness of the functional data.

**directional_outlyingness_stats**

`skfda.exploratory.outliers.directional_outlyingness_stats(fdatagrid: skfda.representation.grid.FDataGrid, *depth_method=<function projection_depth>, pointwise_weights=None) → skfda.exploratory.outliers._directional_outlyingness.DirectionalOutlyingnessStats`

Computes the directional outlyingness of the functional data.

Furthermore, it calculates functional, mean and the variational directional outlyingness of the samples in the data set, which are also returned.

The functional directional outlyingness can be seen as the overall outlyingness, analog to other functional outlyingness measures.

The mean directional outlyingness, describes the relative position (including both distance and direction) of the samples on average to the center curve; its norm can be regarded as the magnitude outlyingness.

The variation of the directional outlyingness, measures the change of the directional outlyingness in terms of both norm and direction across the whole design interval and can be regarded as the shape outlyingness.

Firstly, the directional outlyingness is calculated as follows:

$$O(X(t), F_X(t)) = \left\{ \frac{1}{d(X(t), F_X(t))} - 1 \right\} \cdot v(t)$$

where $X$ is a stochastic process with probability distribution $F$, $d$ a depth function and $v(t) = \{X(t) - Z(t)\} / \|X(t) - Z(t)\|$ is the spatial sign of $\{X(t) - Z(t)\}$, $Z(t)$ denotes the median and $\|\cdot\|$ denotes the $L_2$ norm.

From the above formula, we define the mean directional outlyingness as:

$$MO(X, F_X) = \int_I O(X(t), F_X(t)) \cdot w(t) dt;$$

and the variation of the directional outlyingness as:

$$VO(X, F_X) = \int_I \|O(X(t), F_X(t)) - MO(X, F_X)\|^2 \cdot w(t) dt$$

where $w(t)$ a weight function defined on the domain of $X$, $I$.

Then, the total functional outlyingness can be computed using these values:

$$FO(X, F_X) = \|MO(X, F_X)\|^2 + VO(X, F_X).$$

**Parameters**

- `fdatagrid (FDataGrid)` – Object containing the samples to be ordered according to the directional outlyingness.
- **depth_method** (*depth measure*, optional) – Method used to order the data. Defaults to modified band depth.

- **pointwise_weights** (*array_like*, *optional*) – an array containing the weights of each point of discretisation where values have been recorded. Defaults to the same weight for each of the points: 1/len(interval).

**Returns** DirectionalOutlyingnessStats object.

**Example**

```python
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
                 ...,  [0.5, 0.5, 1, 2, 1.5, 1],
                 ...,  [-1, -1, -0.5, 1, 1, 0.5],
                 ...,  [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> stats = directional_outlyingness_stats(fd)
>>> stats.directional_outlyingness
array([[ 0.89932101,
        0.89932101,
        1.57381177,
        1.01173614,
        1.12415127,
        1.12415127],
       [ 0.        ,
        0.        ,
        0.        ,
        0.        ,
        0.        ,
        0.        ],
       [-0.89932101,
        -0.89932101,
        -0.67449076,
        -0.33724538,
        -0.22483025,
        -0.22483025],
       [-0.44966051,
        -0.44966051,
        -0.67449076,
        -1.6862269 ,
        -2.02347228,
        -1.57381177]], dtype=float32)
>>> stats.functional_directional_outlyingness
array([ 2.99742218,  2.93929124,  3.01966359,  3.36873005])
>>> stats.mean_directional_outlyingness
array([[ 1.12415127,  0.        ,  -0.53959261,  -1.17661166]],
       dtype=float32)
```
**References**


### 1.4 Datasets

Functions to obtain datasets.

#### 1.4.1 Fetching real datasets

The following functions are used to retrieve specific functional datasets:

```python
skfda.datasets.fetch_growth(return_X_y=False)
```

Load the Berkeley Growth Study dataset.

The data is obtained from the R package ‘fda’, which takes it from the Berkeley Growth Study. The Berkeley Growth Study (Tuddenham and Snyder, 1954) recorded the heights of 54 girls and 39 boys between the ages of 1 and 18 years. Heights were measured at 31 ages for each child, and the standard error of these measurements was about 3mm, tending to be larger in early childhood and lower in later years.

**References**


**Parameters**

- `return_X_y` – Return only the data and target as a tuple.

**Examples using `skfda.datasets.fetch_growth`**

- *Elastic registration*
Representation of functional data
K-nearest neighbors classification

fetch_phoneme

skfda.datasets.fetch_phoneme(return_X_y: bool = False)
Load the phoneme dataset.

The data is obtained from the R package ‘ElemStatLearn’, which takes it from the dataset in https://web.stanford.edu/~hastie/ElemStatLearn/.

These data arose from a collaboration between Andreas Buja, Werner Stuetzle and Martin Maechler, and it is used as an illustration in the paper on Penalized Discriminant Analysis by Hastie, Buja and Tibshirani (1995).

The data were extracted from the TIMIT database (TIMIT Acoustic-Phonetic Continuous Speech Corpus, NTIS, US Dept of Commerce) which is a widely used resource for research in speech recognition. A dataset was formed by selecting five phonemes for classification based on digitized speech from this database. The phonemes are transcribed as follows: “sh” as in “she”, “dcl” as in “dark”, “iy” as the vowel in “she”, “aa” as the vowel in “dark”, and “ao” as the first vowel in “water”. From continuous speech of 50 male speakers, 4509 speech frames of 32 msec duration were selected, approximately 2 examples of each phoneme from each speaker. Each speech frame is represented by 512 samples at a 16kHz sampling rate, and each frame represents one of the above five phonemes. The breakdown of the 4509 speech frames into phoneme frequencies is as follows:

<table>
<thead>
<tr>
<th>phoneme</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>695</td>
</tr>
<tr>
<td>ao</td>
<td>1022</td>
</tr>
<tr>
<td>dcl</td>
<td>757</td>
</tr>
<tr>
<td>iy</td>
<td>1163</td>
</tr>
<tr>
<td>sh</td>
<td>872</td>
</tr>
</tbody>
</table>

From each speech frame, a log-periodogram was computed, which is one of several widely used methods for casting speech data in a form suitable for speech recognition. Thus the data used in what follows consist of 4509 log-periodograms of length 256, with known class (phoneme) memberships.

The data contain curves sampled at 256 points, a response variable, and a column labelled “speaker” identifying the different speakers.

References


Parameters

return_X_y – Return only the data and target as a tuple.

Examples using skfda.datasets.fetch_phoneme

Kernel Smoothing

fetch_tecator

skfda.datasets.fetch_tecator(return_X_y: bool = False)
Load the Tecator dataset.

The data is obtained from the R package ‘fda.usc’, which takes it from http://lib.stat.cmu.edu/datasets/tecator.
This is the Tecator data set: The task is to predict the fat content of a meat sample on the basis of its near infrared absorbance spectrum.

1. Statement of permission from Tecator (the original data source)

These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 - 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents.

If results from these data are used in a publication we want you to mention the instrument and company name (Tecator) in the publication. In addition, please send a preprint of your article to

Karin Thente, Tecator AB, Box 70, S-263 21 Hoganas, Sweden

The data are available in the public domain with no responsibility from the original data source. The data can be redistributed as long as this permission note is attached.

For more information about the instrument - call Perstorp Analytical’s representative in your area.

2. Description of the data

For each meat sample the data consists of a 100 channel spectrum of absorbances and the contents of moisture (water), fat and protein. The absorbance is -log10 of the transmittance measured by the spectrometer. The three contents, measured in percent, are determined by analytic chemistry.

There are 215 samples.

Parameters return_X_y – Return only the data and target as a tuple.

Examples using skfda.datasets.fetch_tecator

- Exploring data

fetch_medflies

skfda.datasets.fetch_medflies(return_X_y: bool = False)

Load the Medflies dataset, where the flies are separated in two classes according to their longevity.

The data is obtained from the R package ‘ddalpha’, which is a modification of the dataset in http://www.stat.ucdavis.edu/~wang/data/medfly1000.htm.

The data set medfly1000.dat consists of number of eggs laid daily for each of 1000 medflies (Mediterranean fruit flies, Ceratitis capitata) until time of death. Data were obtained in Dr. Carey’s laboratory. A description of the experiment which was done by Professor Carey of UC Davis and collaborators in a medfly rearing facility in Mexico is in Carey et al.(1998) below. The main questions are to explore the relationship of age patterns of fecundity to mortality, longevity and lifetime reproduction.

A basic finding was that individual mortality is associated with the time-dynamics of the egg-laying trajectory. An approximate parametric model of the egg laying process was developed and used in Müller et al. (2001). Nonparametric approaches which extend principal component analysis for curve data to the situation when covariates are present have been developed and discussed in Chiu, Müller and Wang (2003) and Chiu et al. (2003).

References


**Parameters**

- `return_X_y` – Return only the data and target as a tuple.

**Examples using `skfda.datasets.fetch_medflies`**

- *Representation of functional data*

**fetch_weather**

`skfda.datasets.fetch_weather(return_X_y: bool = False)`

Load the Canadian Weather dataset.

The data is obtained from the R package ‘fda’ from CRAN.

Daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994.

**References**


**Parameters**

- `return_X_y` – Return only the data and target as a tuple.

**Examples using `skfda.datasets.fetch_weather`**

- *Neighbors Functional Regression*
- *Boxplot*
- *Neighbors Scalar Regression*
- *Magnitude-Shape Plot*
- *Clustering*

**fetch_aemet**

`skfda.datasets.fetch_aemet(return_X_y: bool = False)`

Load the Spanish Weather dataset.

The data is obtained from the R package ‘fda.usc’ from CRAN.

Series of daily summaries of 73 spanish weather stations selected for the period 1980-2009. The dataset contains the geographic information of each station and the average for the period 1980-2009

Authors: Manuel Febrero Bande, Manuel Oviedo de la Fuente <manuel.oviedo@usc.es>

Source: The data were obtained from the FTP of AEMET in 2009.

Parameters return_X_y – Return only the data and target as a tuple.

fetch_octane

skfda.datasets.fetch_octane(return_X_y: bool = False)
Load near infrared spectra of gasoline samples.

This function fetchs the octane dataset from the R package ‘mrfDepth’ from CRAN.

Near infrared (NIR) spectra of gasoline samples, with wavelengths ranging from 1102nm to 1552nm with measurements every two nm. This dataset contains six outliers to which ethanol was added, which is required in some states. See [RDEH2006] and [HuRS2015] for further details.

The data is labeled according to this different composition.


References

Parameters return_X_y – Return only the data and target as a tuple.

Those functions return a dictionary with at least a “data” field containing the instance data, and a “target” field containing the class labels or regression values, if any.

In addition datasets can be downloaded from CRAN and the UCR:

| skfda.datasets.fetch_cran(name, package_name, *| Fetch a dataset from CRAN. |
| skfda.datasets.fetch_ucr(name, **kwargs) | Fetch a dataset from the UCR. |

fetch_cran

skfda.datasets.fetch_cran(name, package_name, *, converter=None, **kwargs)
Fetch a dataset from CRAN.

Parameters

- name – Dataset name.
- package_name – Name of the R package containing the dataset.

fetch_ucr

skfda.datasets.fetch_ucr(name, **kwargs)
Fetch a dataset from the UCR.

Parameters name – Dataset name.
Note: Functional multivariate datasets are not yet supported.

References


Datasets from CRAN are not in a standardized format. Datasets from the UCR are in the same format as the specific datasets, but often have an explicit test set, accessible as “data_test” and “target_test”.

1.4.2 Making synthetic datasets

The following functions are used to make synthetic functional datasets:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skfda.datasets.make_gaussian_process(..., ...)</code></td>
<td>Generate Gaussian process trajectories.</td>
</tr>
<tr>
<td><code>skfda.datasets.make_sinusoidal_process(...)</code></td>
<td>Generate sinusoidal process.</td>
</tr>
<tr>
<td><code>skfda.datasets.make_multimodal_samples(...)</code></td>
<td>Generate multimodal samples.</td>
</tr>
<tr>
<td><code>skfda.datasets.make_random_warping(...[, ...])</code></td>
<td>Generate random warping functions.</td>
</tr>
</tbody>
</table>

**make_gaussian_process**

`skfda.datasets.make_gaussian_process(n_samples: int = 100, n_features: int = 100, *, start: float = 0.0, stop: float = 1.0, mean=0, cov=None, noise: float = 0.0, random_state=None)`

Generate Gaussian process trajectories.

**Parameters**

- `n_samples` – The total number of trajectories.
- `n_features` – The total number of trajectories.
- `start` – Starting point of the trajectories.
- `stop` – Ending point of the trajectories.
- `mean` – The mean function of the process. Can be a callable accepting a vector with the locations, or a vector with length `n_features`.
- `cov` – The covariance function of the process. Can be a callable accepting two vectors with the locations, or a matrix with size `n_features x n_features`.
- `noise` – Standard deviation of Gaussian noise added to the data.
- `random_state` – Random state.

**Returns** `FDataGrid` object comprising all the trajectories.

**Examples using skfda.datasets.make_gaussian_process**

- Surface Boxplot
- **Magnitude-Shape Plot synthetic example**

**make_sinusaloidal_process**

```python
skfda.datasets.make_sinusaloidal_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)
```

Generate sinusoidal process.

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 $\alpha_i$ and phases $\phi_i$ 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.

**Examples using** `skfda.datasets.make_sinusaloidal_process`

- **Shift Registration**
- **Interpolation**
- **Radius neighbors classification**
- **Extrapolation**
make_multimodal_samples

skfda.datasets.make_multimodal_samples(n_samples: int = 15, *, n_modes: int = 1, points_per_dim: int = 100, dim_domain: int = 1, dim_codomain: 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)

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.

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

Where $\mu_n = \text{mode_location}_n + \epsilon$ and $\epsilon$ is normally distributed, with mean $\kappa$ and standard deviation given by the parameter $\text{std}$.

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 $\text{points_per_dim}^{\text{dim_domain}}$ points of discretization.
- **dim_domain** – Number of dimensions of the domain.
- **dim_codomain** – 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 $\sigma$ 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.

Examples using skfda.datasets.make_multimodal_samples

- Elastic registration
- Landmark registration
- Representation of functional data
- Landmark shift
- Pairwise alignment
**make_multimodal_landmarks**

```python
skfda.datasets.make_multimodal_landmarks(n_samples: int = 15, *, n_modes: int = 1, dim_domain: int = 1, dim_codomain: int = 1, start: float = -1, stop: float = 1, std: float = 0.05, random_state=None)
```

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 generated 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.
- **dim_domain** – Number of dimensions of the domain.
- **dim_codomain** – Number of dimensions of the codomain.
- **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.

**Examples using skfda.datasets.make_multimodal_landmarks**

- **Landmark registration**
- **Landmark shift**

**make_random_warping**

```python
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)
```

Generate random warping functions.

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

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

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

The random warping it is constructed making an exponential map to \( \Gamma \).
\[ \gamma(t) = \int_0^t \left( \frac{\sin(\|v\|)}{\|v\|} v(s) + \cos(\|v\|) \right)^2 ds \]

An affine translation is used to define the warping in \([a, b]\).

The smoothing and shape of the warpings can be controlling changing \(N\), \(\sigma\) 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 \(\gamma_{id}\). 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.

### 1.5 Miscellaneous

Miscellaneous functions and objects.

#### 1.5.1 Metrics

This module contains multiple functional distances and norms.

**Lp Spaces**

The following functions computes the norms and distances used in Lp spaces.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skfda.misc.metrics.norm_lp(fdatalog[, p, p2])</code></td>
<td>Calculate the norm of all the samples in a FDataGrid object.</td>
</tr>
<tr>
<td><code>skfda.misc.metrics.lp_distance(fdatalog)</code></td>
<td>Lp distance for FDataGrid objects.</td>
</tr>
</tbody>
</table>

**norm_lp**

`skfda.misc.metrics.norm_lp(fdatalog, p=2, p2=2)`

Calculate the norm of all the samples in a FDataGrid object.
For each sample sample $f$ the $L^p$ norm is defined as:

$$
\|f\| = \left( \int_D |f|^p dx \right)^{\frac{1}{p}}
$$

Where $D$ is the domain over which the functions are defined.

The integral is approximated using Simpson's rule.

In general, if $f$ is a multivariate function $(f_1, ..., f_d)$, and $D \subset \mathbb{R}^n$, it is applied the following generalization of the $L^p$ norm.

$$
\|f\| = \left( \int_D \|f\|^p_* dx \right)^{\frac{1}{p}}
$$

Where $\| \cdot \|_*$ denotes a vectorial norm. See `vectorial_norm()` to more information.

For example, if $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, and $\| \cdot \|_*$ is the euclidean norm $\| (x,y) \|_* = \sqrt{x^2 + y^2}$, the $L^p$ norm applied is

$$
\|f\| = \left( \int \int_D \left( \sqrt{|f_1(x,y)|^2 + |f_2(x,y)|^2} \right)^p dx dy \right)^{\frac{1}{p}}
$$

**Parameters**

- `fdatagrid` (FDataGrid) – FDataGrid object.
- `p` (*int*, *optional*) – $p$ of the $L^p$ norm. Must be greater or equal than 1. If $p='inf'$ or $p=np.inf$ it is used the $L$ infinity metric. Defaults to 2.

**Returns** Matrix with as many rows as samples in the first object and as many columns as samples in the second one. Each element $(i, j)$ of the matrix is the inner product of the $i$th sample of the first object and the $j$th sample of the second one.

**Return type** numpy.darray

**Examples**

Calculates the norm of a FDataGrid containing the functions $y = 1$ and $y = x$ defined in the interval $[0,1]$.

```python
>>> x = np.linspace(0,1,1001)
>>> fd = FDataGrid([np.ones(len(x)), x] ,x)
>>> norm_lp(fd).round(2)
array([1. , 0.58])
```

The $L^p$ norm is only defined if $p \geq 1$.

```python
>>> norm_lp(fd, p = 0.5)
Traceback (most recent call last):
    ...
ValueError: p must be equal or greater than 1.
```
lp_distance

skfda.misc.metrics.lp_distance(fdata1, fdata2, p=2, p2=2, *, eval_points=None, _check=True)

Lp distance for FDataGrid objects.

Calculates the distance between all possible pairs of one sample of the first FDataGrid object and one of the second one.

For each pair of samples f and g the distance between them is defined as:

\[ d(f, g) = d(f, g) = \| f - g \| \]

The norm is specified as a parameter but defaults to the l2 norm.

Parameters

- **fdatagrid** (FDataGrid) – FDataGrid object.
- **p** (int, optional) – p of the lp norm. Must be greater or equal than 1. If p='inf' or p=np.inf it is used the L infinity metric. Defaults to 2.
- **p2** (int, optional) – p index of the vectorial norm applied in case of multivariate objects. Defaults to 2. See norm_lp().

Examples

Computes the distances between an object containing functional data corresponding to the functions \( y = 1 \) and \( y = x \) defined over the interval \([0, 1]\) and another ones containing data of the functions \( y = 0 \) and \( y = x/2 \). The result then is an array 2x2 with the computed l2 distance between every pair of functions.

```python
>>> x = np.linspace(0, 1, 1001)
>>> fd = FDataGrid([np.ones(len(x))], x)
>>> fd2 = FDataGrid([np.zeros(len(x))], x)
>>> lp_distance(fd, fd2).round(2)
1.0
```

If the functional data are defined over a different set of points of discretisation the functions returns an exception.

```python
>>> x = np.linspace(0, 2, 1001)
>>> fd2 = FDataGrid([np.zeros(len(x)), x/2 + 0.5], x)
>>> lp_distance(fd, fd2)
Traceback (most recent call last):
  ...
ValueError: Domain ranges for both objects must be equal
```

Examples using skfda.misc.metrics.lp_distance

- **Radius neighbors classification**

Elastic distances

The following functions implements multiple distances used in the elastic analysis and registration of functional data.
### skfda.misc.metrics.fisher_rao_distance

*skfda.misc.metrics.fisher_rao_distance*(...)  
Compute the Fisher-Rao distance between two functional objects.

**Parameters**
- `fdata1` (*FData*) – First FData object.  
- `fdata2` (*FData*) – Second FData object.  
- `eval_points` (*array_like*, *optional*) – Array with points of evaluation.  

**Returns**  
Fisher rao distance.

**Raises**  
ValueError – If the objects are not unidimensional.

### skfda.misc.metrics.amplitude_distance

*skfda.misc.metrics.amplitude_distance*(...)  
Compute the amplitude distance between two functional objects.

**Parameters**
- `fdata1` (*FData*) – First FData object.  
- `fdata2` (*FData*) – Second FData object.  
- `eval_points` (*array_like*, *optional*) – Array with points of evaluation.  
- `lam` (*float*, *optional*) – Penalty term.

**Returns**  
Amplitude distance.

**Raises**  
ValueError – If the objects are not unidimensional.

### skfda.misc.metrics.phase_distance

*skfda.misc.metrics.phase_distance*(...[,])  
Compute the phase distance between two functional objects.

### skfda.misc.metrics.warping_distance

*skfda.misc.metrics.warping_distance*(...[,])  
Compute the distance between warpings functions.

---

**fisher_rao_distance**

*skfda.misc.metrics.fisher_rao_distance*(fdata1, fdata2, *, eval_points=None, _check=True)  
Compute the Fisher-Rao distance between two functional objects.

Let \( f_i \) and \( f_j \) be two functional observations, and let \( q_i \) and \( q_j \) be the corresponding SRSF (see SRSF), the fisher rao distance is defined as

\[
 d_{FR}(f_i, f_j) = \|q_i - q_j\|_2 = \left( \int_0^1 sgn(f_i(t)) \sqrt{|\dot{f}_i(t)|} - sgn(f_j(t)) \sqrt{|\dot{f}_j(t)|} \, dt \right)^{\frac{1}{2}}
\]

If the observations are distributions of random variables the distance will match with the usual fisher-rao distance in non-parametric form for probability distributions [S11-2].

If the samples are defined in a domain different than \((0,1)\) their domains are normalized to this interval with an affine transformation.

**Parameters**
- `fdata1` (*FData*) – First FData object.  
- `fdata2` (*FData*) – Second FData object.  
- `eval_points` (*array_like*, *optional*) – Array with points of evaluation.  

**Returns**  
Fisher rao distance.

**Raises**  
ValueError – If the objects are not unidimensional.

**References:**

---

**amplitude_distance**

*skfda.misc.metrics.amplitude_distance*(fdata1, fdata2, *, lam=0.0, eval_points=None, _check=True, **kwargs)  
Compute the amplitude distance between two functional objects.

Let \( f_i \) and \( f_j \) be two functional observations, and let \( q_i \) and \( q_j \) be the corresponding SRSF (see SRSF), the amplitude distance is defined as

\[
 d_A(f_i, f_j) = \min_{\gamma \in \Gamma} d_{FR}(f_i \circ \gamma, f_j)
\]

A penalty term could be added to restrict the ammount of elasticity in the alignment used.

\[
 d_A^2(f_i, f_j) = \min_{\gamma \in \Gamma} \{ d_{FR}^2(f_i \circ \gamma, f_j) + \lambda \mathcal{R}(\gamma) \}
\]

Where \( d_{FR} \) is the Fisher-Rao distance and the penalty term is given by

\[
 \mathcal{R}(\gamma) = \| \sqrt{\gamma} - 1 \|_2^2
\]
See [SK16-4-10-1] for a detailed explanation.

If the samples are defined in a domain different than (0,1) their domains are normalized to this interval with an affine transformation.

**Parameters**
- `fdata1 (FData)` – First FData object.
- `fdata2 (FData)` – Second FData object.
- `lam (float, optional)` – Penalty term to restrict the elasticity.
- `eval_points (array_like, optional)` – Array with points of evaluation.
- `**kwargs (dict)` – Name arguments to be passed to `elastic_registration_warping()`.

**Returns** Elastic distance.

**Return type** float

**Raises** `ValueError` – If the objects are not unidimensional.

**References:**

`phase_distance`

```
scikit-fda.misc.metrics.phase_distance(fdata1, fdata2, *, lam=0.0, eval_points=None, _check=True, **kwargs)
```

Compute the phase distance between two functional objects.

Let $f_i$ and $f_j$ be two functional observations, and let $\gamma_{ij}$ the corresponding warping used in the elastic registration to align $f_i$ to $f_j$ (see `elastic_registration()`). The phase distance between $f_i$ and $f_j$ is defined as

$$d_P(f_i, f_j) = d_{FR}(\gamma_{ij}, \gamma_{id}) = \arccos\left( \int_0^1 \sqrt{\dot{\gamma}_{ij}(t)} dt \right)$$

See [SK16-4-10-2] for a detailed explanation.

If the samples are defined in a domain different than (0,1) their domains are normalized to this interval with an affine transformation.

**Parameters**
- `fdata1 (FData)` – First FData object.
- `fdata2 (FData)` – Second FData object.
- `lambda (float, optional)` – Penalty term to restrict the elasticity.
- `eval_points (array_like, optional)` – Array with points of evaluation.
- `**kwargs (dict)` – Name arguments to be passed to `elastic_registration_warping()`.

**Returns** Phase distance between the objects.

**Return type** float

**Raises** `ValueError` – If the objects are not unidimensional.

**References:**
warping_distance

```
scikit-fda.misc.metrics.warping_distance(warping1, warping2, *, eval_points=None, _check=True)
```

Compute the distance between warpings functions.

Let \( \gamma_i \) and \( \gamma_j \) be two warpings, defined in \( \gamma_i : [a, b] \rightarrow [a, b] \). The distance in the space of warping functions, \( \Gamma \), with the riemannian metric given by the fisher-rao inner product can be computed using the structure of hilbert sphere in their ssfs's.

\[
d_{\Gamma}(\gamma_i, \gamma_j) = \cos^{-1}\left(\int_0^1 \sqrt{\dot{\gamma}_i(t) \dot{\gamma}_j(t)} \, dt\right)
\]

See [SK16-4-11-2] for a detailed explanation.

If the warpings are not defined in \([0,1]\), an affine transformation is made to change the domain.

**Parameters**

- `fdata1 (FData)` – First warping.
- `fdata2 (FData)` – Second warping.
- `eval_points (array_like, optional)` – Array with points of evaluation.

**Returns** Distance between warpings:

**Return type** `float`

**Raises** `ValueError` – If the objects are not unidimensional.

**References:**

**Utils**

- `skfda.misc.metrics.vectorial_norm(fdatagrid)` – Apply a vectorial norm to a multivariate function.
- `skfda.misc.metrics.distance_from_norm(norm, ...)` – Returns the distance induced by a norm.

**vectorial_norm**

```
scikit-fda.misc.metrics.vectorial_norm(fdatagrid, p=2)
```

Apply a vectorial norm to a multivariate function.

Given a multivariate function \( f : \mathbb{R}^n \rightarrow \mathbb{R}^d \) applies a vectorial norm \( \| \cdot \| \) to produce a function \( \| f \| : \mathbb{R}^n \rightarrow \mathbb{R} \).

For example, let \( f : \mathbb{R} \rightarrow \mathbb{R}^2 \) be \( f(t) = (f_1(t), f_2(t)) \) and \( \| \cdot \|_2 \) the euclidian norm.

\[
\| f \|_2(t) = \sqrt{|f_1(t)|^2 + |f_2(t)|^2}
\]

In general if \( p \neq \pm \infty \) and \( f : \mathbb{R}^n \rightarrow \mathbb{R}^d \)

\[
\| f \|_p(x_1, \ldots, x_n) = \left( \sum_{k=1}^d |f_k(x_1, \ldots, x_n)|^p \right)^{1/p}
\]

**Parameters**
• `fdatagrid` (`FDatagrid`) – Functional object to be transformed.
• `p` (int, optional) – Exponent in the lp norm. If p is a number then it is applied \(\text{sum}(\text{abs}(x)^{**p})**(1./p))\), if p is inf then \(\text{max}(|x|)\), and if p is -inf it is applied \(\text{min}(|x|)\). See `numpy.linalg.norm` to more information. Defaults to 2.

**Returns** FDatagrid with image dimension equal to 1.

**Return type** (FDatagrid)

**Examples**

```python
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.misc.metrics import vectorial_norm

First we will construct an example dataset with curves in \(\mathbb{R}^2\).
```  
```python
>>> fd = make_multimodal_samples(dim_codomain=2, random_state=0)
>>> fd.dim_codomain
2
```

We will apply the euclidean norm

```python
>>> fd = vectorial_norm(fd, p=2)
>>> fd.dim_codomain
1
```

**distance_from_norm**

`skfda.misc.metrics.distance_from_norm(norm, **kwargs)`

Returns the distance induced by a norm.

Given a norm \(\| \cdot \| : X \to \mathbb{R}\), returns the distance \(d : X \times X \to \mathbb{R}\) induced by the norm:

\[ d(f, g) = \|f - g\| \]

**Parameters**

• `norm` (Function) – Norm function `norm(fdata, **kwargs)`.

• `**kwargs` (dict, optional) – Named parameters to be passed to the norm function.

**Returns** Distance function `norm_distance(fdata1, fdata2)`.

**Return type** Function

**Examples**

Computes the \(L^2\) distance between an object containing functional data corresponding to the function \(y(x) = x\) defined over the interval \([0, 1]\) and another one containing data of the function \(y(x) = x/2\).

Firstly we create the functional data.

```python
>>> x = np.linspace(0, 1, 1001)
>>> fd = FDataGrid([x], x)
>>> fd2 = FDataGrid([x/2], x)
```
To construct the $L^2$ distance it is used the $L^2$ norm which it is used to compute the distance.

```python
>>> l2_distance = distance_from_norm(norm_lp, p=2)
>>> d = l2_distance(fd, fd2)
>>> float('%.3f' % d)
0.289
```

**pairwise_distance**

`skfda.misc.metrics.pairwise_distance(distance, **kwargs)`

Return pairwise distance for FDataGrid objects.

Given a distance returns the corresponding pairwise distance function.

The pairwise distance calculates the distance between all possible pairs of one sample of the first FDataGrid object and one of the second one.

The matrix returned by the pairwise distance is a matrix with as many rows as samples in the first object and as many columns as samples in the second one. Each element $(i, j)$ of the matrix is the distance between the $i$th sample of the first object and the $j$th sample of the second one.

**Parameters**

- `distance` (Function) – Distance functions between two functional objects `distance(fd1, fd2, **kwargs)`.

- `**kwargs` (dict, optional) – Parameters dictionary to be passed to the distance function.

**Returns**

Pairwise distance function, which accepts two functional data objects and returns the pairwise distance matrix.

**Return type** Function

**Examples using** `skfda.misc.metrics.pairwise_distance` 

- Radius neighbors classification

### 1.6 Machine Learning

This module contains classes compatible with the scikit-learn estimators and utilities for solving machine learning problems. It consists of three sub-modules: *Classification*, *Clustering* and *Regression*.

#### 1.6.1 Classification

Module with classes to perform classification of functional data.
Nearest Neighbors

This module contains nearest neighbors estimators to perform classification. In the examples K-nearest neighbors classification and Radius neighbors classification it is explained the basic usage of these estimators.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>skfda.ml.classification.KNeighborsClassifier(...)</code></td>
<td>Classifier implementing the k-nearest neighbors vote.</td>
</tr>
<tr>
<td><code>skfda.ml.classification.RadiusNeighborsClassifier(...)</code></td>
<td>Classifier implementing a vote among neighbors within a given radius.</td>
</tr>
</tbody>
</table>

**KNeighborsClassifier**

class `skfda.ml.classification.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, n_jobs=1, multivariate_metric=False)

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 L2 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.

• multivariate_metric (boolean, optional (default = False)) – Indicates if the metric used is a sklearn distance between vectors (see DistanceMetric) or a functional metric of the module skfda.misc.metrics if False.

Examples

Firstly, we will create a toy dataset with 2 classes

```python
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

def make_sinusoidal_process(phase_mean=1.8, error_std=0.,
    phase_std=.25, random_state=0):
    X = np.linspace(0, 10.1, 101)
    y = make_sin('sin', phase_mean=phase_mean, error_std=error_std,
        phase_std=phase_std, random_state=random_state)(X)
    return MultivariateSeries(X, y.reshape(-1, 1))

def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

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

```python
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

from skfda.datasets import make_sin
from skfda.datasets import make_sinusoidal_process
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

from skfda.datasets import make_sin
from skfda.datasets import make_sinusoidal_process
make_sin = make_sin('sin', **kwargs)
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

from skfda.datasets import make_sin
from skfda.datasets import make_sinusoidal_process
make_sin = make_sin('sin', **kwargs)
```

We can predict the class of new samples

```python
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

from skfda.datasets import make_sin
from skfda.datasets import make_sinusoidal_process
def make_sin(name, **kwargs):
    a = getattr(math, name)
    return lambda time: a(time, **kwargs)

from skfda.datasets import make_sin
from skfda.datasets import make_sinusoidal_process
make_sin = make_sin('sin', **kwargs)
```

```python
>>> from skfda.datasets import make_sin
>>> y = 15*[0] + 15*[1]
```

### 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.

Methods

__init__((n_neighbors, weights, algorithm, ...)) 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.

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-Neighbors for points in X.

predict(X) Predict the class labels for the provided data.

predict_proba(X) Return probability estimates for the test data X.

score(X, y[, sample_weight]) Return the mean accuracy on the given test data and labels.

set_params(**params) Set the parameters of this estimator.

__init__(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, n_jobs=1, multivariate_metric=False) Initialize the classifier.

fit(X, y=None) Fit the model using X as training data and y as target values.

Parameters

- X (FDataGrid, array_matrix) – Training data. FDataGrid with the training data or array matrix with shape [n_samples, n_samples] if metric='precomputed'.
- y (array-like or sparse matrix) – Target values of shape = [n_samples] or [n_samples, n_outputs]. In the case of unsupervised search, this parameter is ignored.

Note: This method wraps the corresponding sklearn routine in the module sklearn.neighbors.

get_params(deep=True) Get parameters for this estimator.

Parameters deep (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type mapping of string to any

kneighbors(X=None, n_neighbors=None, return_distance=True) Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

- X (FDataGrid or matrix) – FDatagrid with the query functions or matrix (n_query, n_indexed) if metric == ‘precomputed’. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- n_neighbors (int) – Number of neighbors to get (default is the value passed to the constructor).

return_distance (boolean, optional): Defaults to True. If False, distances will not be returned.
Returns

array  Array representing the lengths to points, only present if return_distance=True

ind  [array] Indices of the nearest points in the population matrix.

Return type  dist

Examples

Firstly, we will create a toy dataset.

```python
>>> 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

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors()
>>> neigh.fit(fd)
```

Now we can query the k-nearest neighbors.

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

```python
>>> 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 ]])
```

Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

`kneighbors_graph(X=None, n_neighbors=None, mode='connectivity')`

Computes the (weighted) graph of k-Neighbors for points in X

Parameters

- **X** (FDataGrid or matrix) – FDataGrid with the query functions or matrix (n_query, n_indexed) if metric == ‘precomputed’. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- **n_neighbors** (int) – Number of neighbors to get (default is the value passed to the constructor).
- **mode** (‘connectivity’ or ‘distance’, optional) – Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are distance between points.
Returns

Sparse matrix in CSR format, shape = [n_samples, n_samples_fit] \( n_{samples\_fit} \) is the number of samples in the fitted data \( A[i, j] \) is assigned the weight of edge that connects \( i \) to \( j \).

Examples

Firstly, we will create a toy dataset.

```python
>>> 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.

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors()
>>> neigh.fit(fd)
```

Now we can obtain the graph of k-neighbors of a sample.

```python
>>> graph = neigh.kneighbors_graph(fd[0])
>>> print(graph)
(0, 0) 1.0
(0, 7) 1.0
(0, 6) 1.0
(0, 11) 1.0
(0, 2) 1.0
```

Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**predict\( (X) \)**

Predict the class labels for the provided data.

**Parameters**

\( X \) (FDataGrid or array-like) – FDataGrid with the test samples or array
\( (n\_query, n\_indexed) \) if metric == ‘precomputed’.

**Returns**

\( y \) : array of shape [n_samples] or [n_samples, n_outputs] with class labels for each data sample.

**Return type**

(np.array)

Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**predict_proba\( (X) \)**

Return probability estimates for the test data \( X \).

**Parameters**

\( X \) (FDataGrid or array-like) – FDataGrid with the test samples or array
\( (n\_query, n\_indexed) \) if metric == ‘precomputed’.
Returns

\( p \) [array of shape = [n_samples, n_classes], or a list of n_outputs] of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

score\((X, y, \text{sample_weight=}None)\)
Return the mean accuracy on the given test data and labels.
In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- \( X \) (array-like of shape \((n_{samples}, n_{features})\)) – Test samples.
- \( y \) (array-like of shape \((n_{samples},)\) or \((n_{samples}, n_{outputs})\)) – True labels for \( X \).
- sample_weight (array-like of shape \((n_{samples},)\), default=None) – Sample weights.

Returns score – Mean accuracy of self.predict\((X)\) wrt. \( y \).

Return type float

set_params\(**params\)
Set the parameters of this estimator.
The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form \(<\text{component}>\_<\text{parameter}>\) so that it’s possible to update each component of a nested object.

Parameters **params (dict) – Estimator parameters.

Returns self – Estimator instance.

Return type object

Examples using skfda.ml.classification.KNeighborsClassifier

- K-nearest neighbors classification

RadiusNeighborsClassifier

class skfda.ml.classification.RadiusNeighborsClassifier\((radius=1.0, \text{weights=}\text{uniform'}, \text{algorithm=}\text{auto'}, \text{leaf_size=}30, \text{metric=}\text{l2'}, \text{metric_params=}None, \text{outlier_label=}None, \text{n_jobs=}1, \text{multivariate_metric=}False)\)
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.

[callsable]: 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 L2 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.

- **multivariate_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.

```python
>>> 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.

```python
>>> 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.

```python
>>> 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, 1])
```

See also:

- KNeighborsClassifier
- NearestCentroid
- KNeighborsRegressor
- RadiusNeighborsRegressor
- NearestNeighbors

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`.


Methods

- `__init__` ([radius, weights, algorithm, …]) 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.
- `radius_neighbors(X[, radius, return_distance])` Finds the neighbors within a given radius of a fdatagrid or fdatagrids.
- `radius_neighbors_graph(X, radius, mode)` Computes the (weighted) graph of Neighbors for points in X Neighborhoods are restricted the points at a distance lower than radius.
- `score(X, y[, sample_weight])` Return the mean accuracy on the given test data and labels.
- `set_params(**params)` Set the parameters of this estimator.

```python
__init__(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, outlier_label=None, n_jobs=1, multivariate_metric=False)
```

Initialize the classifier.

```python
fit(X, y=None)
```

Fit the model using X as training data and y as target values.

Parameters

- `X` (FDataGrid, array_matrix) – Training data. FDataGrid with the training data or array matrix with shape \([n\_samples, n\_samples]\) if metric='precomputed'.
- `y` (array-like or sparse matrix) – Target values of shape \([n\_samples]\) or \([n\_samples, n\_outputs]\). In the case of unsupervised search, this parameter is ignored.

**Note:** This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

```python
get_params(deep=True)
```
Get parameters for this estimator.

**Parameters**
- `deep (bool, default=True)` – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**
- `params` – Parameter names mapped to their values.

**Return type**
- mapping of string to any

**predict(X)**
Predict the class labels for the provided data.

**Parameters**
- `X (FDataGrid or array-like)` – FDataGrid with the test samples or array (n_query, n_indexed) if metric == ‘precomputed’.

**Returns**
- `y : array of shape [n_samples] or [n_samples, n_outputs] with class labels for each data sample.`

**Return type**
- (np.array)

**Notes**
This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**radius_neighbors(X=None, radius=None, return_distance=True)**
Finds the neighbors within a given radius of a fdatagrid or fdatagrids. Return the indices and distances of each point from the dataset lying in a ball with size radius around the points of the query array. Points lying on the boundary are included in the results. The result points are not necessarily sorted by distance to their query point.

**Parameters**
- `X (FDataGrid, optional)` – fdatagrid with the sample or samples whose neighbors will be returned. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- `radius (float, optional)` – Limiting distance of neighbors to return. (default is the value passed to the constructor).
- `return_distance (boolean, optional)` – distances will not be returned

**Returns**

( array, shape (n_samples): dist [array of arrays representing the] distances to each point, only present if return_distance=True. The distance values are computed according to the metric constructor parameter.)

( array, shape (n_samples,): An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size radius around the query points.)

**Examples**
Firstly, we will create a toy dataset.

```python
>>> 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.,
(continues on next page)```
We will fit a Nearest Neighbors estimator.

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
```

Now we can query the neighbors in the radius.

```python
>>> distances, index = neigh.radius_neighbors(fd[:2])
>>> index[0]  # Neighbors of sample 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:**

`kneighbors`

**Notes**

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

`radius_neighbors_graph(X=None, radius=None, mode='connectivity')`

Computes the (weighted) graph of Neighbors for points in X Neighborhoods are restricted the points at a distance lower than radius.

**Parameters**

- `X (FDataGrid)` – The query sample or samples. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- `radius (float)` – Radius of neighborhoods. (default is the value passed to the constructor).
- `mode (‘connectivity’ or ’distance’, optional)` – Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ’distance’ the edges are distance between points.

**Returns** sparse matrix in CSR format, shape = [n_samples, n_samples] A[i, j] is assigned the weight of edge that connects i to j.

**Notes**

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.
score(X, y, sample_weight=None)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (array-like of shape (n_samples, n_features)) – Test samples.
- **y** (array-like of shape (n_samples, ) or (n_samples, n_outputs)) – True labels for X.
- **sample_weight** (array-like of shape (n_samples, ), default=None) – Sample weights.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it’s possible to update each component of a nested object.

Parameters **params (dict) – Estimator parameters.

Returns **self** – Estimator instance.

Return type object

Examples using skfda.ml.classification.RadiusNeighborsClassifier

- **Radius neighbors classification**

NearestCentroid

class skfda.ml.classification.NearestCentroid(metric='l2', mean='mean')

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.
- **centroid** (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 distances. 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 containing the centroid of each class

**Type** FDataGrid

**Examples**

Firstly, we will create a toy dataset with 2 classes

```python
>>> 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

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

We can predict the class of new samples

```python
>>> 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, 1])
```

See also:

KNeighborsClassifier RadiuNeighborsClassifier KNeighborsRegressor
RadiusNeighborsRegressor NearestNeighbors

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code> ([metric, mean])</td>
<td>Initialize the classifier.</td>
</tr>
<tr>
<td><code>fit(X, y)</code></td>
<td>Fit the model using X as training data and y as target values.</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>predict(X)</code></td>
<td>Predict the class labels for the provided data.</td>
</tr>
<tr>
<td><code>score(X, y[, sample_weight])</code></td>
<td>Return the mean accuracy on the given test data and labels.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

```

```
- **y (array-like or sparse matrix)** – Target values of shape = [n_samples] or [n_samples, n_outputs].

**get_params** *(deep=True)*
Get parameters for this estimator.

**Parameters**
- **deep** *(bool, default=True)* – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**
- **params** – Parameter names mapped to their values.

**predict** *(X)*
Predict the class labels for the provided data.

**Parameters**
- **X** *(FDataGrid)* – FDataGrid with the test samples.

**Returns**
- **y** : array of shape [n_samples] or [n_samples, n_outputs] with class labels for each data sample.

**score** *(X, y, sample_weight=None)*
Return the mean accuracy on the given test data and labels.
In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**
- **X** *(array-like of shape (n_samples, n_features))* – Test samples.
- **y** *(array-like of shape (n_samples,) or (n_samples, n_outputs))* – True labels for X.
- **sample_weight** *(array-like of shape (n_samples,), default=None)* – Sample weights.

**Returns**
- **score** – Mean accuracy of self.predict(X) wrt. y.

**set_params** *(**params)*
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters**
- ****params** *(dict)* – Estimator parameters.

**Returns**
- **self** – Estimator instance.

**Return type**
- **object**

### 1.6.2 Clustering

Module with classes to perform clustering of functional data.
K means algorithms

The following classes implement both, the K-Means and the Fuzzy K-Means algorithms respectively. In order to show the results in a visual way, the module `skfda.exploratory.visualization.clustering_plots` can be used. See the example `Clustering` for a detailed explanation.

**skfda.ml.clustering.KMeans([n_clusters, ...])** Representation and implementation of the K-Means algorithm for the FdataGrid object.

**skfda.ml.clustering.FuzzyCMeans([...])** Representation and implementation of the Fuzzy c-Means clustering algorithm for the FDataGrid object.

### KMeans

class skfda.ml.clustering.KMeans(n_clusters=2, init=None, metric=<function lp_distance>, n_init=1, max_iter=100, tol=0.0001, random_state=0)

Representation and implementation of the K-Means algorithm for the FdataGrid object.

Let $X = \{x_1, x_2, ..., x_n\}$ be a given dataset to be analyzed, and $V = \{v_1, v_2, ..., v_c\}$ be the set of centers of clusters in $X$ dataset in $m$ dimensional space ($\mathbb{R}^m$). Where $n$ is the number of objects, $m$ is the number of features, and $c$ is the number of partitions or clusters.

KM iteratively computes cluster centroids in order to minimize the sum with respect to the specified measure. KM algorithm aims at minimizing an objective function known as the squared error function given as follows:

$$ J_{KM}(X; V) = \sum_{i=1}^{c} \sum_{j=1}^{n} D_{ij}^2 $$

Where, $D_{ij}^2$ is the squared chosen distance measure which can be any p-norm: $D_{ij} = \|x_{ij} - v_i\| = \left( \int_I |x_{ij} - v_i|^p dx \right)^{\frac{1}{p}}$, being $I$ the domain where $X$ is defined, $1 \leq i \leq c$, $1 \leq j \leq n_i$. Where $n_i$ represents the number of data points in i-th cluster.

For $c$ clusters, KM is based on an iterative algorithm minimizing the sum of distances from each observation to its cluster centroid. The observations are moved between clusters until the sum cannot be decreased any more. KM algorithm involves the following steps:

1. **Centroids of $c$ clusters are chosen from $X$** randomly or are passed to the function as a parameter.
2. Distances between data points and cluster centroids are calculated.
3. **Each data point is assigned to the cluster whose centroid is closest to it.**
4. Cluster centroids are updated by using the following formula: $v_i = \sum_{i=1}^{n_i} x_{ij}/n_i$, $1 \leq i \leq c$.
5. Distances from the updated cluster centroids are recalculated.
6. **If no data point is assigned to a new cluster the run of algorithm is stopped**, otherwise the steps from 3 to 5 are repeated for probable movements of data points between the clusters.

This algorithm is applied for each dimension on the image of the FDataGrid object.

**Parameters**

1.6. Machine Learning 171
• `n_clusters (int, optional) – Number of groups into which the samples are classified. Defaults to 2.

• `init (FDataGrid, optional) – Contains the initial centers of the different clusters the algorithm starts with. Its data_matrix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.dim_codomain). Defaults to None, and the centers are initialized randomly.

• `metric (optional) – functional data metric. Defaults to `lp_distance`.

• `n_init (int, optional) – 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.

• `max_iter (int, optional) – Maximum number of iterations of the clustering algorithm for a single run. Defaults to 100.

• `tol (float, optional) – tolerance used to compare the centroids calculated with the previous ones in every single run of the algorithm.

• `random_state (int, RandomState instance or None, optional) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. Defaults to 0. See Glossary.

labels_ (numpy.ndarray)

labels is a vector in which each entry contains the cluster each observation belongs to.

cluster_centers_

data_matrix of shape (n_clusters, ncol, dim_codomain) and contains the centroids for each cluster.

Type FDataGrid object

inertia_

Sum of squared distances of samples to their closest cluster center for each dimension.

Type numpy.ndarray, (fdatagrid.dim_codomain)

n_iter_

number of iterations the algorithm was run for each dimension.

Type numpy.ndarray, (fdatagrid.dim_codomain)

Example

```python
>>> import skfda
>>> data_matrix = [[1, 1, 2, 3, 2.5, 2],
...                 [0.5, 0.5, 1, 2, 1.5, 1],
...                 [-1, -1, -0.5, 1, 1, 0.5],
...                 [-0.5, -0.5, -0.5, -1, -1, -1]]
>>> sample_points = [0, 2, 4, 6, 8, 10]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> kmeans = skfda.ml.clustering.KMeans(random_state=0)
>>> kmeans.fit(fd)
KMeans(...)
>>> kmeans.cluster_centers_.data_matrix
array(_gradients, 166666667],
      [ 0.16666667],
      [ 0.83333333])
```
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([n_clusters, init, metric, n_init,...])</td>
<td>Initialization of the KMeans class.</td>
</tr>
<tr>
<td>fit(X[, y, sample_weight])</td>
<td>Computes Fuzzy K-Means clustering calculating the attributes labels_, cluster_centers_, inertia_ and n_iter_.</td>
</tr>
<tr>
<td>fit_predict(X[, y])</td>
<td>Perform clustering on X and returns cluster labels.</td>
</tr>
<tr>
<td>fit_transform(X[, y, sample_weight])</td>
<td>Compute clustering and transform X to cluster-distance space.</td>
</tr>
<tr>
<td>get_params([deep])</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td>predict(X[, sample_weight])</td>
<td>Predict the closest cluster each sample in X belongs to.</td>
</tr>
<tr>
<td>score(X[, y, sample_weight])</td>
<td>Opposite of the value of X on the K-means objective.</td>
</tr>
<tr>
<td>set_params(**params)</td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td>transform(X)</td>
<td>Transform X to a cluster-distance space.</td>
</tr>
</tbody>
</table>

__init__(n_clusters=2, init=None, metric=<function lp_distance>, n_init=1, max_iter=100, tol=0.0001, random_state=0)

Initialization of the KMeans class.

Parameters

- **n_clusters** (*int, optional*) – Number of groups into which the samples are classified. Defaults to 2.
- **init** (*FDataGrid, optional*) – Contains the initial centers of the different clusters the algorithm starts with. Its data_matrix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.dim_codomain). Defaults to None, and the centers are initialized randomly.
- **metric** (*optional*) – functional data metric. Defaults to *lp_distance*.
- **n_init** (*int, optional*) – 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.
- **max_iter** (*int, optional*) – Maximum number of iterations of the clustering algorithm for a single run. Defaults to 100.
- **tol** (*float, optional*) – tolerance used to compare the centroids calculated with the previous ones in every single run of the algorithm.
- **random_state** *(int, RandomState instance or None, optional)* – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. Defaults to 0.

```python
fit(X, y=None, sample_weight=None)
```
Computes Fuzzy K-Means clustering calculating the attributes `labels_`, `cluster_centers_`, `inertia_` and `n_iter_`.

**Parameters**
- **X** *(FDataGrid object)* – Object whose samples are clustered, classified into different groups.
- **y** *(Ignored)* – present here for API consistency by convention.
- **sample_weight** *(Ignored)* – present here for API consistency by convention.

```python
fit_predict(X, y=None)
```
Perform clustering on X and returns cluster labels.

**Parameters**
- **X** *(ndarray, shape (n_samples, n_features))* – Input data.
- **y** *(Ignored)* – Not used, present for API consistency by convention.

**Returns**
- **labels** – Cluster labels.

**Return type** ndarray, shape (n_samples,)

```python
fit_transform(X, y=None, sample_weight=None)
```
Compute clustering and transform X to cluster-distance space.

**Parameters**
- **X** *(FDataGrid object)* – Object whose samples are classified into different groups.
- **y** *(Ignored)* – present here for API consistency by convention.
- **sample_weight** *(Ignored)* – present here for API consistency by convention.

**Returns**
- `(n_samples, n_clusters))`: distances of each sample to each cluster.

**Return type** distances_to_centers (numpy.ndarray)

```python
get_params(deep=True)
```
Get parameters for this estimator.

**Parameters**
- **deep** *(bool, default=True)* – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**
- **params** – Parameter names mapped to their values.

**Return type** mapping of string to any

```python
predict(X, sample_weight=None)
```
Predict the closest cluster each sample in X belongs to.

**Parameters**
- **X** *(FDataGrid object)* – Object whose samples are classified into different groups.
- **y** *(Ignored)* – present here for API consistency by convention.
- **sample_weight** *(Ignored)* – present here for API consistency by convention.
Returns Label of each sample.

score(X, y=None, sample_weight=None)
Opposite of the value of X on the K-means objective.

Parameters
- X (FDataGrid object) – Object whose samples are classified into different groups.
- y (Ignored) – present here for API consistency by convention.
- sample_weight (Ignored) – present here for API consistency by convention.

Returns
(fdatagrid.dim_codomain)): negative inertia_ attribute.

Returns (n_samples, n_clusters)): distances of each sample to each cluster.

Examples using skfda.ml.clustering.KMeans

- Clustering

FuzzyCMeans

class skfda.ml.clustering.FuzzyCMeans(n_clusters=2, init=None, metric=<function lp_distance>, n_init=1, max_iter=100, tol=0.0001, random_state=0, fuzzifier=2)

Representation and implementation of the Fuzzy c-Means clustering algorithm for the FDataGrid object.
Let $X = \{x_1, x_2, ..., x_n\}$ be a given dataset to be analyzed, and $V = \{v_1, v_2, ..., v_c\}$ be the set of centers of clusters in $X$ dataset in $m$ dimensional space ($\mathbb{R}^m$). Where $n$ is the number of objects, $m$ is the number of features, and $c$ is the number of partitions or clusters.

FCM minimizes the following objective function:

$$J_{FCM}(X; U, V) = \sum_{i=1}^{c} \sum_{j=1}^{n} u_{ij}^f D_{ij}^2.$$ 

This function differs from classical KM with the use of weighted squared errors instead of using squared errors only. In the objective function, $U$ is a fuzzy partition matrix that is computed from dataset $X$: $U = [u_{ij}] \in M_{FCM}$. The fuzzy clustering of $X$ is represented with $U$ membership matrix. The element $u_{ij}$ is the membership value of $j$-th object to $i$-th cluster. In this case, the $i$-th row of $U$ matrix is formed with membership values of $n$ objects to $i$-th cluster. $V$ is a prototype vector of cluster prototypes (centroids): $V = \{v_1, v_2, ..., v_c\}$, $\mathbf{v_i} \in \mathbb{R}^m$.

$D_{ij}^2$ is the squared chosen distance measure which can be any p-norm: $D_{ij} = \|x_{ij} - v_i\| = (\int_I |x_{ij} - v_i|^p dx)^{\frac{1}{p}}$, being $I$ the domain where $X$ is defined, $1 \leq i \leq c$, $1 \leq j \leq n_i$. Where $n_i$ represents the number of data points in $i$-th cluster.

FCM is an iterative process and stops when the number of iterations is reached to maximum, or when the centroids of the clusters do not change. The steps involved in FCM are:

1. **Centroids of $c$ clusters are chosen from $X$** randomly or are passed to the function as a parameter.
2. **Membership values of data points to each cluster are calculated** with: $u_{ij} = \left[\sum_{k=1}^{c} (D_{ij}/D_{kj})^{\frac{2}{p-1}}\right]^{-1}$.
3. **Cluster centroids are updated** by using the following formula: $v_i = \frac{\sum_{j=1}^{n} u_{ij}^f x_{ij}}{\sum_{j=1}^{n} u_{ij}^f}$, $1 \leq i \leq c$.
4. **If no cluster centroid changes the run of algorithm is stopped**, otherwise return to step 2.

This algorithm is applied for each dimension on the image of the FDataGrid object.

**Parameters**

- $n\_clusters$ (*int, optional*) – Number of groups into which the samples are classified. Defaults to 2.
- $init$ (FDataGrid, *optional*) – Contains the initial centers of the different clusters the algorithm starts with. Its data matrix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.dim_codomain). Defaults to None, and the centers are initialized randomly.
- $n\_init$ (*int, optional*) – 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.
- $max\_iter$ (*int, optional*) – Maximum number of iterations of the clustering algorithm for a single run. Defaults to 100.
- $tol$ (*float, optional*) – tolerance used to compare the centroids calculated with the previous ones in every single run of the algorithm.
• random_state (int, RandomState instance or None, optional) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. Defaults to 0. See Glossary.

• fuzzifier (int, optional) – Scalar parameter used to specify the degree of fuzziness in the fuzzy algorithm. Defaults to 2.

labels_ (numpy.ndarray)
  (n_samples, n_clusters)): 2-dimensional matrix in which each row contains the cluster that observation belongs to.

cluster_centers_
  data_matrix of shape (n_clusters, ncol, dim_codomain) and contains the centroids for each cluster.

  Type FDataGrid object

inertia_
  Sum of squared distances of samples to their closest cluster center for each dimension.

  Type numpy.ndarray, (fdatagrid.dim_codomain)

n_iter_
  number of iterations the algorithm was run for each dimension.

  Type numpy.ndarray, (fdatagrid.dim_codomain)

Example

```python
>>> import skfda
>>> data_matrix = [[[1, 0.3], [2, 0.4], [3, 0.5], [4, 0.6]],
                 ...
                 [2, 0.5], [3, 0.6], [4, 0.7], [5, 0.7]],
                 ...
                 [3, 0.2], [4, 0.3], [5, 0.4], [6, 0.5]]
>>> sample_points = [2, 4, 6, 8]
>>> fd = skfda.FDataGrid(data_matrix, sample_points)
>>> fuzzy_kmeans = skfda.ml.clustering.FuzzyCMeans(random_state=0)
>>> fuzzy_kmeans.fit(fd)
FuzzyCMeans(...)
>>> fuzzy_kmeans.cluster_centers_.data_matrix
array([[[2.83994301, 0.24786354],
        [3.83994301, 0.34786354],
        [4.83994301, 0.44786354],
        [5.83994301, 0.53191927]],
       [[1.25134384, 0.35023779],
        [2.25134384, 0.45023779],
        [3.25134384, 0.55023779],
        [4.25134384, 0.6251158 ]]]
```
### Table 87 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fit_predict(X[, y])</code></td>
<td>Perform clustering on X and returns cluster labels.</td>
</tr>
<tr>
<td><code>fit_transform(X[, y, sample_weight])</code></td>
<td>Compute clustering and transform X to cluster-distance space.</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>predict(X[, sample_weight])</code></td>
<td>Predict the closest cluster each sample in X belongs to.</td>
</tr>
<tr>
<td><code>score(X[, y, sample_weight])</code></td>
<td>Opposite of the value of X on the K-means objective.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>transform(X)</code></td>
<td>Transform X to a cluster-distance space.</td>
</tr>
</tbody>
</table>

#### `__init__`

Initialization of the FuzzyKMeans class.

**Parameters**

- **n_clusters** (int, optional) – Number of groups into which the samples are classified. Defaults to 2.
- **init** (FDataGrid, optional) – Contains the initial centers of the different clusters the algorithm starts with. Its data matrix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.dim_codomain). Defaults to None, and the centers are initialized randomly.
- **metric** (optional) – functional data metric. Defaults to `lp_distance`.
- **n_init** (int, optional) – 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.
- **max_iter** (int, optional) – Maximum number of iterations of the clustering algorithm for a single run. Defaults to 100.
- **tol** (float, optional) – tolerance used to compare the centroids calculated with the previous ones in every single run of the algorithm.
- **random_state** (int, RandomState instance or None, optional) – Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. Defaults to 0.
- **fuzzifier** (int, optional) – Scalar parameter used to specify the degree of fuzziness in the fuzzy algorithm. Defaults to 2.

#### `fit(X, y=None, sample_weight=None)`

Computes Fuzzy K-Means clustering calculating the attributes `labels_`, `cluster_centers_`, `inertia_` and `n_iter_`.

**Parameters**

- **X** (FDataGrid object) – Object whose samples are clustered, classified into different groups.
- **y** (Ignored) – present here for API consistency by convention.
- **sample_weight** (Ignored) – present here for API consistency by convention.

#### `fit_predict(X, y=None)`

Perform clustering on X and returns cluster labels.
Parameters

- \(X\) (\texttt{ndarray}, shape \((n\_samples, n\_features)\)) – Input data.
- \(y\) (\texttt{Ignored}) – Not used, present for API consistency by convention.

Returns \texttt{labels} – Cluster labels.

Return type \texttt{ndarray}, shape \((n\_samples,)\)

\texttt{fit\_transform}(X, y=None, sample\_weight=None)
Compute clustering and transform \(X\) to cluster-distance space.

Parameters

- \(X\) (\texttt{FDataGrid object}) – Object whose samples are classified into different groups.
- \(y\) (\texttt{Ignored}) – present here for API consistency by convention.
- \texttt{sample\_weight} (\texttt{Ignored}) – present here for API consistency by convention.

Returns

\((n\_samples, n\_clusters))): distances of each sample to each cluster.

Return type \texttt{distances\_to\_centers (numpy.ndarray}

\texttt{get\_params}(deep=True)
Get parameters for this estimator.

Parameters \texttt{deep} (\texttt{bool}, \texttt{default=True}) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns \texttt{params} – Parameter names mapped to their values.

Return type \texttt{mapping of string to any}

\texttt{predict}(X, sample\_weight=None)
Predict the closest cluster each sample in \(X\) belongs to.

Parameters

- \(X\) (\texttt{FDataGrid object}) – Object whose samples are classified into different groups.
- \(y\) (\texttt{Ignored}) – present here for API consistency by convention.
- \texttt{sample\_weight} (\texttt{Ignored}) – present here for API consistency by convention.

Returns \texttt{Label of each sample.}

\texttt{score}(X, y=None, sample\_weight=None)
Opposite of the value of \(X\) on the K-means objective.

Parameters

- \(X\) (\texttt{FDataGrid object}) – Object whose samples are classified into different groups.
- \(y\) (\texttt{Ignored}) – present here for API consistency by convention.
- \texttt{sample\_weight} (\texttt{Ignored}) – present here for API consistency by convention.

Returns

\((\text{fdatagrid.dim\_codomain})): negative \texttt{inertia} attribute.

Return type \texttt{score (numpy.array}
set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The
latter have parameters of the form <component>__<parameter> so that it’s possible to update
each component of a nested object.

Parameters **params (dict) – Estimator parameters.

Returns self – Estimator instance.

Return type object

transform(X)

Transform X to a cluster-distance space.

Parameters

• X (FDataGrid object) – Object whose samples are classified into different groups.
• y (Ignored) – present here for API consistency by convention.
• sample_weight (Ignored) – present here for API consistency by convention.

Returns

(n_samples, n_clusters)): distances of each sample to each cluster.

Return type distances_to_centers (numpy.ndarray

Examples using skfda.ml.clustering.FuzzyCMeans

• Clustering

Nearest Neighbors

The class NearestNeighbors implements the nearest neighbors algorithm to perform unsupervised neighbor
searches.

skfda.ml.clustering.NearestNeighbors([…]) Unsupervised learner for implementing neighbor
searches.

NearestNeighbors

class skfda.ml.clustering.NearestNeighbors(n_neighbors=5, radius=1.0, algorithm='auto',
leaf_size=30, metric='l2', metric_params=None,
n_jobs=1, multivariate_metric=False)

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 L2 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.

- **multivariate_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

```python
>>> 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

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
```

Now we can query the k-nearest neighbors.

```python
>>> distances, index = neigh.kneighbors(fd[:2])
```

```python
Index of k-neighbors of samples 0 and 1
array([[ 0,  7,  6, 11,  2],
       [ 0,  27,  28,  29,  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  NearestCentroids  KNeighborsRegressor  RadiusNeighborsRegressor

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.


Methods

__init__(n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric='l2', metric_params=None, n_jobs=1, multivariate_metric=False)

Initialize the nearest neighbors searcher.

fit(X, y=None)

Fit the model using X as training data and y as target values.

Parameters

- `X` (FDataGrid, array_matrix) – Training data. FDataGrid with the training data or array matrix with shape \([n\_samples, n\_samples]\) if `metric='precomputed'`.
- `y` (array-like or sparse matrix) – Target values of shape \([n\_samples]\) or \([n\_samples, n\_outputs]\). In the case of unsupervised search, this parameter is ignored.
Note: This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

```python
get_params(deep=True)
```

Get parameters for this estimator.

**Parameters**

- **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**

- **params** – Parameter names mapped to their values.

```python
kneighbors(X=None, n_neighbors=None, return_distance=True)
```

Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

**Parameters**

- **X** (*FDataGrid or matrix*) – FDatagrid with the query functions or matrix (n_query, n_indexed) if metric == ‘precomputed’. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

- **n_neighbors** (*int*) – Number of neighbors to get (default is the value passed to the constructor).

**return_distance** (*boolean, optional*): Defaults to True. If False, distances will not be returned.

**Returns**

- **array** Array representing the lengths to points, only present if return_distance=True

- **ind** [array] Indices of the nearest points in the population matrix.

**Return type**

- **dist**

**Examples**

Firstly, we will create a toy dataset.

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

We will fit a Nearest Neighbors estimator

```python
>>> from skfda.ml.clustering import NearestNeighbors
click.add_line('>>> neigh = NearestNeighbors()')
click.add_line('>>> neigh.fit(fd)')
click.add_line('NearestNeighbors(algorithm='auto', leaf_size=30,...)
```

Now we can query the k-nearest neighbors.

```python
>>> distances, index = neigh.kneighbors(fd[2])
click.add_line('>>> index # Index of k-neighbors of samples 0 and 1')
click.add_line('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 ]])

Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

`kneighbors_graph(X=None, n_neighbors=None, mode='connectivity')`  
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

- `X` (FDataGrid or matrix) – FDataGrid with the query functions or matrix (n_query, n_indexed) if metric == ‘precomputed’. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- `n_neighbors` (int) – Number of neighbors to get (default is the value passed to the constructor).
- `mode` (‘connectivity’ or ‘distance’, optional) – Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are distance between points.

Returns Sparse matrix in CSR format, shape = [n_samples, n_samples_fit] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

Examples

Firstly, we will create a toy dataset.

```python
>>> 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.

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

Now we can obtain the graph of k-neighbors of a sample.

```python
>>> graph = neigh.kneighbors_graph(fd[0])
>>> print(graph)
(0, 0) 1.0
(0, 7) 1.0
(0, 6) 1.0
(0, 11) 1.0
(0, 2) 1.0
```
Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**radius_neighbors** (*X=None, radius=None, return_distance=True*)

Finds the neighbors within a given radius of a fdatagrid or fdatagrids. Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results. The result points are not necessarily sorted by distance to their query point.

**Parameters**

- **X** (FDataGrid, optional) – fdatagrid with the sample or samples whose neighbors will be returned. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- **radius** (float, optional) – Limiting distance of neighbors to return. (default is the value passed to the constructor).
- **return_distance** (boolean, optional) – distances will not be returned

**Returns**

(array, shape (n_samples): dist [array of arrays representing the distances to each point, only present if return_distance=True. The distance values are computed according to the metric constructor parameter.]

(array, shape (n_samples,): An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

**Examples**

Firstly, we will create a toy dataset.

```python
>>> 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.

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
```

Now we can query the neighbors in the radius.

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

```python
>>> distances[0].round(2) # Distances to neighbors of the sample 0
array([ 0.0 ,  0.30,  0.29,  0.28,  0.29])
```
See also:

cneighbors

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple
query points cannot be fit in a standard data array. For efficiency, radius_neighbors returns arrays
of objects, where each object is a 1D array of indices or distances.

This method wraps the corresponding sklearn routine in the module sklearn.neighbors.

radius_neighbors_graph (X=None, radius=None, mode='connectivity')

Computes the (weighted) graph of Neighbors for points in X Neighborhoods are restricted the
points at a distance lower than radius.

Parameters

- X (FDataGrid) – The query sample or samples. If not provided, neighbors of each
  indexed point are returned. In this case, the query point is not considered its own
  neighbor.
- radius (float) – Radius of neighborhoods. (default is the value passed to the
  constructor).
- mode ('connectivity' or 'distance', optional) – Type of returned matrix:
  ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’
  the edges are distance between points.

Returns

sparse matrix in CSR format, shape = [n_samples, n_samples] A[i, j] is as-
signed the weight of edge that connects i to j.

Notes

This method wraps the corresponding sklearn routine in the module sklearn.neighbors.

set_params(**params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The
latter have parameters of the form <component>__<parameter> so that it’s possible to update
each component of a nested object.

Parameters  **params (dict) – Estimator parameters.

Returns  self – Estimator instance.

Return type  object

1.6.3 Regression

Module with classes to perform regression of functional data.

Linear regression

Todo: Add documentation of linear regression models.
**LinearScalarRegression**

```python
class skfda.ml.regression.LinearScalarRegression(beta_basis)
```

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialize self. See help(type(self)) for accurate signature.</td>
</tr>
<tr>
<td>fit(X, y, sample_weight)</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td>predict(X)</td>
<td>Return the coefficient of determination $R^2$ of the prediction.</td>
</tr>
<tr>
<td>score(X, y, sample_weight=None)</td>
<td>Return the coefficient of determination $R^2$ of the prediction.</td>
</tr>
<tr>
<td>set_params(**params)</td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

#### __init__ (beta_basis)

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

#### get_params (deep=True)

Get parameters for this estimator.

- **Parameters**
  - deep (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

- **Returns**
  - params – Parameter names mapped to their values.

#### score(X, y, sample_weight=None)

Return the coefficient of determination $R^2$ of the prediction.

The coefficient $R^2$ is defined as $(1 - u/v)$, where $u$ is the residual sum of squares $\sum ((y_{true} - y_{pred})^2)$ and $v$ is the total sum of squares $\sum ((y_{true} - y_{true.mean})^2)$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of $y$, disregarding the input features, would get a $R^2$ score of 0.0.

- **Parameters**
  - X (array-like of shape (n_samples, n_features)) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.
  - y (array-like of shape (n_samples,) or (n_samples, n_outputs)) – True values for X.
  - sample_weight (array-like of shape (n_samples,), default=None) – Sample weights.

- **Returns**
  - score – $R^2$ of self.predict(X) wrt. y.

- **Return type**
  - float
Notes

The R2 score used when calling score on a regressor will use multioutput='uniform_average' from version 0.23 to keep consistent with r2_score(). This will influence the score method of all the multioutput regressors (except for MultiOutputRegressor). To specify the default value manually and avoid the warning, please either call r2_score() directly or make a custom scorer with make_scorer() (the built-in scorer ‘r2’ uses multioutput='uniform_average').

set_params(**params)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it’s possible to update each component of a nested object.

Parameters
**params (dict) – Estimator parameters.

Returns
self – Estimator instance.

Return type
object

Nearest Neighbors

This module contains nearest neighbors estimators to perform regression. In the examples Neighbors Scalar Regression and Neighbors Functional Regression it is explained the basic usage of these estimators.

| `skfda.ml.regression.KNeighborsRegressor(__)` | Regression based on k-nearest neighbors. |
| `skfda.ml.regression.RadiusNeighborsRegressor(__)` | Regression based on neighbors within a fixed radius. |

KNeighborsRegressor

class `skfda.ml.regression.KNeighborsRegressor`(n_neighbors=5, weights='uniform', regressor='mean', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, n_jobs=1, multivariate_metric=False)

Regression based on k-nearest neighbors.

Regression with scalar, multivariate or functional 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.

• **repressor (callable, optional (default = mean))** Function to perform the local regression in the functional response case. By default used the mean. Can 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 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 L2 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.

• **multivariate_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.

```python
>>> from skfda.ml.regression import KNeighborsRegressor
>>> 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_train = y.flatten()

>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)

>>> X_test = make_multimodal_samples(n_samples=5, std=.05, random_state=0)
```

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

```python
>>> neigh = KNeighborsRegressor()
>>> neigh.fit(X_train, y_train)
KNeighborsRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples
```python
>>> neigh.predict(X_test).round(2)  # Predict test data
array([ 0.38, 0.14, 0.27, 0.52, 0.38])
```

Now we will create a functional response to train the model

```python
>>> y_train = 5 * X_train + 1
>>> y_train
FDataGrid(...)
```

We train the estimator with the functional response

```python
>>> neigh.fit(X_train, y_train)
KNeighborsRegressor(algorithm='auto', leaf_size=30,...)
```

And predict the responses as in the first case.

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

See also:

- `KNeighborsClassifier`
- `RadiusNeighborsClassifier`
- `NearestCentroids`
- `RadiusNeighborsRegressor`
- `NearestNeighbors`

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.


Methods

- `__init__([n_neighbors, weights, regressor, ...])` Initialize the regressor.
- `fit(X, y)` Fit the model using `X` as training data and `y` as responses.
- `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-Neighbors for points in `X`.
- `predict(X)` Predict the target for the provided data.
- `score(X, y[, sample_weight])` Return the coefficient of determination \( R^2 \) of the prediction.
- `set_params(**params)` Set the parameters of this estimator.
__init__(n_neighbors=5, weights='uniform', regressor='mean', algorithm='auto', leaf_size=30,
metric='l2', metric_params=None, n_jobs=1, multivariate_metric=False)
Initialize the regressor.

fit(X, y)
Fit the model using X as training data and y as responses.

Parameters

- X (FDataGrid, array_matrix) – Training data. FDataGrid with the training data or array matrix with shape \([n_{samples}, n_{samples}]\) if metric='precomputed'.
- Y (FData or array_like) – Training data. FData with the training responses (functional response case) or array matrix with length \(n_{samples}\) in the multivariate response case.

Returns self.

Return type Estimator

get_params(deep=True)
Get parameters for this estimator.

Parameters deep (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type mapping of string to any

kneighbors(X=None, n_neighbors=None, return_distance=True)
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

- X (FDataGrid or matrix) – FDatagrid with the query functions or matrix \((n_{query}, n_{indexed})\) if metric == 'precomputed'. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- n_neighbors (int) – Number of neighbors to get (default is the value passed to the constructor).

return_distance (boolean, optional): Defaults to True. If False, distances will not be returned.

Returns

array Array representing the lengths to points, only present if return_distance=True
ind [array] Indices of the nearest points in the population matrix.

Return type dist

Examples

Firstly, we will create a toy dataset.

```python
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=0.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
... phase_std=0.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
```
We will fit a Nearest Neighbors estimator

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

Now we can query the k-nearest neighbors.

```python
>>> 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 ]])
```

**Notes**

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

`kneighbors_graph(X=None, n_neighbors=None, mode='connectivity')`

Computes the (weighted) graph of k-Neighbors for points in X

**Parameters**

- `X` (FDataGrid or matrix) – FDatagrid with the query functions or matrix (n_query, n_indexed) if metric == ‘precomputed’. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

- `n_neighbors` (int) – Number of neighbors to get (default is the value passed to the constructor).

- `mode` (`'connectivity' or 'distance', optional`) – Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are distance between points.

**Returns** Sparse matrix in CSR format, shape = [n_samples, n_samples_fit] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

**Examples**

Firstly, we will create a toy dataset.

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

Now we can obtain the graph of k-neighbors of a sample.

```graph = neigh.kneighbors_graph(fd[0])
```
```print(graph)
(0, 0) 1.0
(0, 7) 1.0
(0, 6) 1.0
(0, 11) 1.0
(0, 2) 1.0
```

### Notes

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**predict(X)**

Predict the target for the provided data.

**Parameters**

- **X** (*FDataGrid* or array-like) – *FDataGrid* with the test samples or array (n_query, n_indexed) if metric == ‘precomputed’.

**Returns**

- **array of shape = [n_samples] or [n_samples, n_outputs]** or *FData* containing as many samples as X.

**Return type**

**y**

**score(X, y, sample_weight=None)**

Return the coefficient of determination $R^2$ of the prediction.

In the multivariate response case, the coefficient $R^2$ is defined as

$$1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$$

where $\hat{y}_i$ is the prediction associated to the test sample $X_i$, and $y_i$ is the true response. See `sklearn.metrics.r2_score` for more information.

In the functional case it is returned an extension of the coefficient of determination $R^2$, defined as

$$1 - \frac{\int \sum_{i=1}^{n} (y_i(t) - \hat{y}_i(t))^2 dt}{\int \sum_{i=1}^{n} (y_i(t) - \bar{y}(t))^2 dt}$$

The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a $R^2$ score of 0.0.

**Parameters**

- **X** (*FDataGrid*) – Test samples to be predicted.
- **y** (*FData or array-like*) – True responses of the test samples.
• **sample_weight** *(array_like, shape = [n_samples], optional) – Sample weights.*

**Returns** Coefficient of determination.

**Return type** *(float)*

**set_params(**params)**

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters** **params** *(dict) – Estimator parameters.*

**Returns** self – Estimator instance.

**Return type** *object*

**Examples using skfda.ml.regression.KNeighborsRegressor**

• *Neighbors Functional Regression*

• *Neighbors Scalar Regression*

**RadiusNeighborsRegressor**

*class* skfda.ml.regression.RadiusNeighborsRegressor*(radius=1.0, weights='uniform', regressor='mean', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, outlier_response=None, n_jobs=1, multivariate_metric=False)*

Regression based on neighbors within a fixed radius.

Regression with scalar, multivariate or functional response.

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 in the functional response case. By default used the mean. Can 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 L2 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** (FData, optional (default = None)) – Default response in the functional response case 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.

• **multivariate_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.

```python
>>> from skfda.ml.regression import RadiusNeighborsRegressor
>>> from skfda.datasets import make_multimodal_samples
>>> y = make_multimodal_landmarks(n_samples=30, std=.5, random_state=0)
>>> y_train = y.flatten()
>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
>>> X_test = make_multimodal_samples(n_samples=5, std=.05, random_state=0)
```

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

```python
>>> neigh = RadiusNeighborsRegressor(radius=0.2)
>>> neigh.fit(X_train, y_train)
RadiusNeighborsRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples
Now we will create a functional response to train the model

```python
>>> y_train = 5 * X_train + 1
>>> y_train
FDataGrid(...)
```

We train the estimator with the functional response

```python
>>> neigh.fit(X_train, y_train)
RadiusNeighborsRegressor(algorithm='auto', leaf_size=30, ...)
```

And predict the responses as in the first case.

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

See also:

- `KNeighborsClassifier`
- `RadiusNeighborsClassifier`
- `NearestCentroids`
- `KNeighborsRegressor`
- `NearestNeighbors`

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`.


Methods

- `__init__([radius, weights, regressor, ...])`: Initialize the classifier.
- `fit(X, y)`: Fit the model using X as training data and y as responses.
- `get_params([deep])`: Get parameters for this estimator.
- `predict(X)`: Predict the target for the provided data
- `radius_neighbors([X, radius, return_distance])`: Finds the neighbors within a given radius of a fdatagrid or fdatagrids.
- `radius_neighbors_graph([X, radius, mode])`: Computes the (weighted) graph of Neighbors for points in X Neighborhoods are restricted the points at a distance lower than radius.
- `score(X, y[, sample_weight])`: Return the coefficient of determination $R^2$ of the prediction.
- `set_params(**params)`: Set the parameters of this estimator.

```
__init__(radius=1.0, weights='uniform', regressor='mean', algorithm='auto', leaf_size=30, metric='l2', metric_params=None, outlier_response=None, n_jobs=1, multivariate_metric=False)
```

Initialize the classifier.
**fit**(*X, y*)
Fit the model using *X* as training data and *y* as responses.

**Parameters**
- *X* ([FDataGrid, array_matrix]) – Training data. FDataGrid with the training data or array matrix with shape [n_samples, n_samples] if metric='precomputed'.
- *Y* ([FData or array_like]) – Training data. FData with the training responses (functional response case) or array matrix with length n_samples in the multivariate response case.

**Returns**
- self. 

**Return type**
- Estimator

**get_params**(deep=True)
Get parameters for this estimator.

**Parameters**
- **deep** (bool, default=True) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**
- params – Parameter names mapped to their values.

**Return type**
- mapping of string to any

**predict**(*X*)
Predict the target for the provided data

**Parameters**
- *X* ([FDataGrid or array-like]) – FDataGrid with the test samples or array (n_query, n_indexed) if metric == 'precomputed'.

**Returns**
- array of shape = [n_samples] or [n_samples, n_outputs] or FData containing as many samples as *X*.

**Return type**
- *y*

**radius_neighbors**(*X=None, radius=None, return_distance=True*)
Finds the neighbors within a given radius of a fdatagrid or fdataarrays. Return the indices and distances of each point from the dataset lying in a ball with size radius around the points of the query array. Points lying on the boundary are included in the results. The result points are not necessarily sorted by distance to their query point.

**Parameters**
- *X* ([FDataGrid, optional]) – fdatagrid with the sample or samples whose neighbors will be returned. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- **radius** (float, optional) – Limiting distance of neighbors to return. (default is the value passed to the constructor).
- **return_distance** (boolean, optional) – distances will not be returned

**Returns**
- (array, shape (n_samples): dist [array of arrays representing the] distances to each point, only present if return_distance=True. The distance values are computed according to the metric constructor parameter.)
(array, shape (n_samples,): An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size radius around the query points.

Examples

Firstly, we will create a toy dataset.

```python
>>> 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.

```python
>>> from skfda.ml.clustering import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
```

Now we can query the neighbors in the radius.

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

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

See also:
kneighbors

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, `radius_neighbors` returns arrays of objects, where each object is a 1D array of indices or distances.

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

`radius_neighbors_graph(X=None, radius=None, mode='connectivity')`

Computes the (weighted) graph of Neighbors for points in X Neighborhoods are restricted the points at a distance lower than radius.

Parameters

- `X` (FDataGrid) – The query sample or samples. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.
- `radius` (float) – Radius of neighborhoods. (default is the value passed to the constructor).
- **mode** ("connectivity" or "distance", optional) – Type of returned matrix: ‘connectivity’ will return the connectivity matrix with ones and zeros, in ‘distance’ the edges are distance between points.

**Returns** sparse matrix in CSR format, shape = [n_samples, n_samples] \( A[i, j] \) is assigned the weight of edge that connects \( i \) to \( j \).

**Notes**

This method wraps the corresponding sklearn routine in the module `sklearn.neighbors`.

**score**(X, y, sample_weight=None)

Return the coefficient of determination \( R^2 \) of the prediction.

In the multivariate response case, the coefficient \( R^2 \) is defined as

\[
1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \frac{1}{n} \sum_{i=1}^{n} y_i)^2}
\]

where \( \hat{y}_i \) is the prediction associated to the test sample \( X_i \), and \( y_i \) is the true response. See `sklearn.metrics.r2_score` for more information.

In the functional case it is returned an extension of the coefficient of determination \( R^2 \), defined as

\[
1 - \frac{\sum_{i=1}^{n} \int (y_i(t) - \hat{y}_i(t))^2 dt}{\sum_{i=1}^{n} \int (y_i(t) - \frac{1}{n} \sum_{i=1}^{n} y_i(t))^2 dt}
\]

The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of \( y \), disregarding the input features, would get a \( R^2 \) score of 0.0.

**Parameters**

- **X** (*FDataGrid*) – Test samples to be predicted.
- **y** (*FData* or array-like) – True responses of the test samples.
- **sample_weight** (array_like, shape = [n_samples], optional) – Sample weights.

**Returns** Coefficient of determination.

**Return type** (float)

**set_params**(**params**)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>_<parameter>` so that it’s possible to update each component of a nested object.

**Parameters** **params** (dict) – Estimator parameters.

**Returns** self – Estimator instance.

**Return type** object
Examples of several functionalities of the package.

**Note:** Click [here](#) to download the full example code

### 2.1 Discretized function representation

Shows how to make a discretized representation of a function.

```python
# Author: Carlos Ramos Carreño <unmabus@gmail.com>
# License: MIT

# sphinx_gallery_thumbnail_number = 2

import numpy as np
from skfda import FDataGrid

We will construct a dataset containing several sinusoidal functions with random displacements.

```python
random_state = np.random.RandomState(0)

sample_points = np.linspace(0, 1)
data = np.array([np.sin((sample_points + random_state.randn())
                 * 2 * np.pi) for _ in range(5)])
```

The FDataGrid class is used for datasets containing discretized functions that are measured at the same points.

```python
fd = FDataGrid(data, sample_points,
               dataset_label='Sinusoidal curves',
               )
```
axes_labels=['t', 'x(t)'])

fd = fd[:5]

We can plot the measured values of each function in a scatter plot.

fd.scatter(s=0.5)

Out:

<Figure size 640x480 with 1 Axes>

We can also plot the interpolated functions.

fd.plot()
2.2 Function composition

This example shows the composition of multidimensional FDataGrids.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

from mpl_toolkits.mplot3d import axes3d
import numpy as np
import skfda
```
Function composition can be applied to our data once it is in functional form using the method `compose()`. Let \( f : X \rightarrow Y \) and \( g : Y \rightarrow Z \), the composition will produce a third function \( g \circ f : X \rightarrow Z \) which maps \( x \in X \) to \( g(f(x)) \) \([1]\).

In *Landmark registration* it is shown the simplest case, where it is used to apply a transformation of the time scale of unidimensional data to register its features.

The following example shows the basic usage applied to a surface and a curve, although the method will work for data with arbitrary dimensions to.

Firstly we will create a data object containing a surface \( g : \mathbb{R}^2 \rightarrow \mathbb{R} \).

Constructs example surface

```python
X, Y, Z = axes3d.get_test_data(1.2)
data_matrix = [Z.T]
sample_points = [X[:, 0], Y[:, 0]]

g = skfda.FDataGrid(data_matrix, sample_points)

# Sets cubic interpolation
# Sets cubic interpolation
g.interpolator = skfda.representation.interpolation.SplineInterpolator(
    interpolation_order=3)

# Plots the surface
# Plots the surface
```

```
```

```

![Image of surface plot]

```

```
We will create a parametric curve $f(t) = (10 \cos(t), 10 \sin(t))$. The result of the composition, $g \circ f : \mathbb{R} \to \mathbb{R}$ will be another functional object with the values of $g$ along the path given by $f$.

```python
# Creation of circumference in parametric form

t = np.linspace(0, 2 * np.pi, 100)

data_matrix = [10 * np.array([np.cos(t), np.sin(t)]).T]

f = skfda.FDataGrid(data_matrix, t)

# Composition of function

gof = g.compose(f)

gof.plot()
```

In the following chart it is plotted the curve $(10 \cos(t), 10 \sin(t), g \circ f(t))$ and the surface.
# Plots surface
fig = g.plot(alpha=.8)

# Plots path along the surface
path = f(t)[0]
fig.axes[0].plot(path[:, 0], path[:, 1], gof(t)[0], color="orange")
fig

Out:

<Figure size 640x480 with 1 Axes>


Total running time of the script: ( 0 minutes 0.492 seconds)

Note:  Click here to download the full example code
2.3 Surface Boxplot

Shows the use of the surface boxplot, which is a generalization of the functional boxplot for FDataGrid whose domain dimension is 2.

```python
# Author: Amanda Hernando Bernabé
# License: MIT
# sphinx_gallery_thumbnail_number = 3

import matplotlib.pyplot as plt
import numpy as np
from skfda import FDataGrid
from skfda.datasets import make_gaussian_process
from skfda.exploratory.visualization import SurfaceBoxplot, Boxplot

In order to instantiate a SurfaceBoxplot, a functional data object with bidimensional domain must be generated. In this example, a FDataGrid representing a function \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \) is constructed, using as an example a Brownian process extruded into another dimension.

The values of the Brownian process are generated using `make_gaussian_process()`. Those functions return FDataGrid objects whose `data_matrix` store the values needed.

```python
n_samples = 10
n_features = 10

fd = make_gaussian_process(n_samples=n_samples, n_features=n_features,
                            random_state=1)
fd.dataset_label = "Brownian process"
```

After, those values generated for one dimension on the domain are extruded along another dimension, obtaining a three-dimensional matrix or cube (two-dimensional domain and one-dimensional image).

```python
cube = np.repeat(fd.data_matrix, n_features).reshape((n_samples, n_features, n_features))
```

We can plot now the extruded trajectories.

```python
fd_2 = FDataGrid(data_matrix=cube,
                 sample_points=np.tile(fd.sample_points, (2, 1)),
                 dataset_label="Extruded Brownian process")
fd_2.plot()
```
Since matplotlib was initially designed with only two-dimensional plotting in mind, the three-dimensional plotting utilities were built on top of matplotlib’s two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization as we can observe.

For this reason, the profiles of the surfaces, which are contained in the first two generated functional data objects, are plotted below, to help to visualize the data.

```python
fd.plot()
```
To terminate the example, the instantiation of the `SurfaceBoxplot` object is made, showing the surface boxplot which corresponds to our FDataGrid.

```python
surfaceBoxplot = SurfaceBoxplot(fd_2)
surfaceBoxplot.plot()
```
Out:

```
<Figure size 640x480 with 1 Axes>
```

The surface boxplot contains the median, the central envelope and the outlying envelope plotted from darker to lighter colors, although they can be customized.

Analogous to the procedure followed before of plotting the three-dimensional data and their corresponding profiles, we can obtain also the functional boxplot for one-dimensional data with the `Boxplot` passing as arguments the first `FdataGrid` object. The profile of the surface boxplot is obtained.

```python
boxplot1 = Boxplot(fd)
boxplot1.plot()
```
2.4 Shift Registration

Shows the use of shift registration applied to a sinusoidal process represented in a Fourier basis.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

import matplotlib.pyplot as plt

from skfda.datasets import make_sinusoidal_process

(continues on next page)
In this example we will use a sinusoidal process synthetically generated. This dataset consists in a sinusoidal wave with fixed period which contains phase and amplitude variation with gaussian noise.

In this example we want to register the curves using a translation and remove the phase variation to perform further analysis.

```python
fd = make_sinusoidal_process(random_state=1)
fplot()
```

We will smooth the curves using a basis representation, which will help us to remove the gaussian noise. Smoothing before registration is essential due to the use of derivatives in the optimization process. Because of their sinusoidal nature we will use a Fourier basis.

```python
fd_basis = fd.to_basis(Fourier(n_basis=11))
f_basis.plot()
```
We will use the `ShiftRegistration()` transformer, which is suitable due to the periodicity of the dataset and the small amount of amplitude variation.

We can observe how the sinusoidal pattern is easily distinguishable once the alignment has been made.

```python
shift_registration = ShiftRegistration()
fd_registered = shift_registration.fit_transform(fd_basis)
fd_registered.plot()
```
We will plot the mean of the original smoothed curves and the registered ones, and we will compare with the original sinusoidal process without noise.

We can see how the phase variation affects to the mean of the original curves varying their amplitude with respect to the original process, however, this effect is mitigated after the registration.

```python
# sinusoidal process without variation and noise
sine = make_sinusoidal_process(n_samples=1, phase_std=0,
                               amplitude_std=0, error_std=0)

fig = fd_basis.mean().plot()
fd_registered.mean().plot(fig)
sine.plot(fig, linestyle='dashed')

fig.axes[0].legend(['original mean', 'registered mean', 'sine'])
```

Out:

```python
<Figure size 640x480 with 1 Axes>
```
The values of the shifts $\delta_i$, stored in the attribute `deltas_` may be relevant for further analysis, as they may be considered as nuisance or random effects.

```python
print(shift_registration.deltas_)
plt.show()
```

Out:
```
[ 0.09004943  0.01808744  0.08732826 -0.00013559 -0.04950421  0.11984576
 -0.09723283 -0.09330286 -0.04398832 -0.08389279  0.0583045  0.00503724
 0.08788296  0.0214795  -0.042531]
```

**Total running time of the script:** ( 0 minutes 0.638 seconds)

**Note:** Click [here](#) to download the full example code
2.5 Neighbors Functional Regression

Shows the usage of the nearest neighbors regressor with functional response.

```python
# Author: Pablo Marcos Manchón
# License: MIT
# sphinx_gallery_thumbnail_number = 4

from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import skfda
from skfda.ml.regression import KNeighborsRegressor
from skfda.representation.basis import Fourier

In this example we are going to show the usage of the nearest neighbors regressors with functional response. There is available a K-nn version, `KNeighborsRegressor`, and other one based in the radius, `RadiusNeighborsRegressor`.

As in the scalar response example, we will fetch the Canadian weather dataset, which contains the daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994. The following figure shows the different temperature and precipitation curves.

```python
data = skfda.datasets.fetch_weather()
fd = data['data']

# Split dataset, temperatures and curves of precipitation
X, y = fd.coordinates

Temperatures
X.plot()
```
Canadian Weather

Out:

```python
<Figure size 640x480 with 1 Axes>
```

Precipitation

```python
y.plot()
```
We will try to predict the precipitation curves. First of all we are going to make a smoothing of the precipitation curves using a basis representation, employing for it a fourier basis with 5 elements.

```python
y = y.to_basis(Fourier(n_basis=5))
y.plot()
```
We will split the dataset in two partitions, for training and test, using the sklearn function `train_test_split()`.

\[
X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}} = \text{train\_test\_split}(X, y, \text{test\_size}=0.1, \text{random\_state}=28)
\]

We will try make a prediction using 5 neighbors and the $L_2$ distance. In this case, to calculate the response we will use a mean of the response, weighted by their distance to the test sample.

\[
knn = \text{KNeighborsRegressor}(\text{n\_neighbors}=5, \text{weights}=\text{'distance'})
\]

\[
knn\text{.fit}(X_{\text{train}}, y_{\text{train}})
\]

Out:

\[
\text{KNeighborsRegressor}(\text{algorithm}=\text{'auto'}, \text{leaf\_size}=30, \text{metric}=\text{'l2'}, \text{metric\_params}=\text{None}, \text{multivariate\_metric}=\text{False}, \text{n\_jobs}=1, \text{n\_neighbors}=5, \text{regressor}=\text{'mean'}, \text{weights}=\text{'distance'})
\]

We can predict values for the test partition using `predict()`. The following figure shows the real precipitation curves, in dashed line, and the predicted ones.
y_pred = knn.predict(X_test)

# Plot prediction
fig = y_pred.plot()
fig.axes[0].set_prop_cycle(None)  # Reset colors
y_test.plot(fig=fig, linestyle='--')

Out:

<Figure size 640x480 with 1 Axes>

We can quantify how much variability it is explained by the model using the score() method, which computes the value

\[
1 - \frac{\sum_{i=1}^{n} \int (y_i(t) - \hat{y}_i(t))^2 dt}{\sum_{i=1}^{n} \int (y_i(t) - \frac{1}{n} \sum_{i=1}^{n} y_i(t))^2 dt}
\]

where \(y_i\) are the real responses and \(\hat{y}_i\) the predicted ones.

score = knn.score(X_test, y_test)
print(score)

Out:
More detailed information about the Canadian weather dataset can be obtained in the following references.


**Total running time of the script:** (0 minutes 2.735 seconds)

**Note:** Click [here](#) to download the full example code

## 2.6 Elastic registration

Shows the usage of the elastic registration to perform a groupwise alignment.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 5

import numpy as np
import skfda
from skfda.datasets import make_multimodal_samples, fetch_growth
from skfda.preprocessing.registration import ElasticRegistration
from skfda.preprocessing.registration.elastic import elastic_mean

In the example of pairwise alignment was shown the usage of ElasticRegistration to align a set of functional observations to a given template or a set of templates.

In the groupwise alignment all the samples are aligned to the same template, constructed to minimise some distance, generally a mean or a median. In the case of the elastic registration, due to the use of the elastic distance in the alignment, one of the most suitable templates is the karcher mean under this metric.

We will create a synthetic dataset to show the basic usage of the registration.

```python
fd = make_multimodal_samples(n_modes=2, stop=4, random_state=1)
fds.plot()
```
The following figure shows the `elastic_mean()` of the dataset and the cross-sectional mean, which correspond to the karcher-mean under the $L^2$ distance.

It can be seen how the elastic mean better captures the geometry of the curves compared to the standard mean, since it is not affected by the deformations of the curves.

```python
fig = fd.mean().plot(label="L2 mean")
elastic_mean(fd).plot(fig=fig, label="Elastic mean")
fig.legend()
```
In this case, the alignment completely reduces the amplitude variability between the samples, aligning the maximum points correctly.

```
elastic_registration = ElasticRegistration()
fd_align = elastic_registration.fit_transform(fd)
fd_align.plot()
```
In general these type of alignments are not possible, in the following figure it is shown how it works with a real dataset. The Berkeley growth dataset contains the growth curves of a set children, in this case will be used only the males. The growth curves will be resampled using cubic interpolation and derived to obtain the velocity curves.

First we show the original curves:

```python
growth = fetch_growth()

# Select only one sex
fd = growth['data'][growth['target'] == 0]

# Obtain velocity curves
fd.interpolator = skfda.representation.interpolation.SplineInterpolator(3)
fd = fd.to_grid(np.linspace(*fd.domain_range[0], 200)).derivative()
fd = fd.to_grid(np.linspace(*fd.domain_range[0], 50))
fd.plot()
```
We now show the aligned curves:

```python
def_align = elastic_registration.fit_transform(fd)
def_align.dataset_label += " - aligned"
fd_align.plot()
```
2.7 Landmark registration

This example shows the basic usage of the landmark registration.

```python
# Author: Pablo Marcos Manchón
# License: MIT

import matplotlib.pyplot as plt
```

Note: Click [here](#) to download the full example code.
The simplest curve alignment procedure is landmark registration. This method only takes into account a discrete amount of features of the curves which will be registered.

A landmark or a feature of a curve is some characteristic that one can associate with a specific argument value \( t \). These are typically maxima, minima, or zero crossings of curves, and may be identified at the level of some derivatives as well as at the level of the curves themselves. We align the curves by transforming \( t \) for each curve so that landmark locations are the same for all curves (RaSi2005, RaHoGr2009).

We will use a dataset synthetically generated by \texttt{make_multimodal_samples()}\footnote{fd = skfda.datasets.make_multimodal_samples(n_samples=4, n_modes=2, std=.002, mode_std=.005, random_state=1)\texttt{fd.plot()}}\footnote{Out: <Figure size 640x480 with 1 Axes>}, which in this case will be used to generate bimodal curves.

For this type of alignment we need to know in advance the location of the landmarks of each of the samples, in our case it will correspond to the two maximum points of each sample. Because our dataset has been generated synthetically we can obtain the value of the landmarks using the function.
make_multimodal_landmarks(), which is used by make_multimodal_samples() to set the location of the modes.

In general it will be necessary to use numerical or other methods to determine the location of the landmarks.

```python
landmarks = skfda.datasets.make_multimodal_landmarks(n_samples=4, n_modes=2,
std=.002, random_state=1).squeeze()
print(landmarks)
```

```
Out:

[[-0.2606904  0.30597475]
 [-0.35695389  0.28534872]
 [-0.29463113  0.23040539]
 [-0.25530298  0.29929113]]
```

The transformation will not be linear, and will be the result of applying a warping function to the time of our curves.

After the identification of the landmarks associated with the features of each of our curves we can construct the warping function with the function landmark_registration_warping().

Let \( h_i \) be the warping function corresponding with the curve \( i \), \( t_{ij} \) the time where the curve \( i \) has their feature \( j \) and \( t^*_{ij} \) the new location of the feature \( j \). The warping functions will transform the new time in the original time of the curve, i.e., \( h_i(t^*_{ij}) = t_{ij} \). These functions will be defined between landmarks using monotone cubic interpolation (see the example of interpolation for more details).

In this case we will place the landmarks at -0.5 and 0.5.

```python
warping = skfda.preprocessing.registration.landmark_registration_warping(
    fd, landmarks, location=[-0.5, 0.5])

# Plots warping
fig = warping.plot()

# Plot landmarks
for i in range(fd.n_samples):
    fig.axes[0].scatter([-0.5, 0.5], landmarks[i])
```
Once we have the warping functions, the registered curves can be obtained using function composition. Let $x_i$ a curve, we can obtain the corresponding registered curve as $x^*_i(t) = x_i(h_i(t))$.

```python
fd_registered = fd.compose(warping)
fig = fd_registered.plot()
fig.axes[0].scatter([-0.5, 0.5], [1, 1])
```
If we do not need the warping function we can obtain the registered curves directly using the function `landmark_registration()`.

If the position of the new location of the landmarks is not specified the mean position is taken.

```python
fd_registered = skfda.preprocessing.registration.landmark_registration(fd, landmarks)
fd_registered.plot()
plt.scatter(np.mean(landmarks, axis=0), [1, 1])
plt.show()
```
2.8 Exploring data

Explores the Tecator data set by plotting the functional data and calculating means and derivatives.

```python
# Author: Miguel Carbajo Berrocal
# License: MIT

import numpy as np
import skfda
```

In this example we are going to explore the functional properties of the Tecator dataset. This dataset measures the infrared absorbance spectrum of meat samples. The objective is to predict the fat, water, and protein content of the samples.

In this example we only want to discriminate between meat with less than 20% of fat, and meat with a higher fat content.
dataset = skfda.datasets.fetch_tecator()
fat = dataset['data']
y = dataset['target']
target_feature_names = dataset['target_feature_names']
fat = y[:, np.asarray(target_feature_names) == 'Fat'].ravel()

We will now plot in red samples containing less than 20% of fat and in blue the rest.

low_fat = fat < 20
labels = np.full(fd.n_samples, 'high fat')
labels[low_fat] = 'low fat'
colors = {'high fat': 'red', 'low fat': 'blue'}

fig = fd.plot(group=labels, group_colors=colors, linewidth=0.5, alpha=0.7, legend=True)

The means of each group are the following ones.

mean_low = skfda.exploratory.stats.mean(fd[low_fat])
mean_high = skfda.exploratory.stats.mean(fd[~low_fat])
means = mean_high.concatenate(mean_low)

(continues on next page)
means.dataset_label = fd.dataset_label + ' - means'
means.plot(group=['high fat', 'low fat'], group_colors=colors,
    linewidth=0.5, legend=True)

Out:

<Figure size 640x480 with 1 Axes>

In this dataset, the vertical shift in the original trajectories is not very significative for predicting the fat content. However, the shape of the curve is very relevant. We can observe that looking at the first and second derivatives.

The first derivative is shown below:

fdd = fd.derivative(1)
fig = fdd.plot(group=labels, group_colors=colors,
    linewidth=0.5, alpha=0.7, legend=True)
We now show the second derivative:

```python
fdd = fd.derivative(2)
fig = fdd.plot(group=labels, group_colors=colors,
                linewidth=0.5, alpha=0.7, legend=True)
```

![Spectrometric curves - 1 derivative](image)
2.9 Representation of functional data

Explores the different representations of functional data.

```python
# Author: Carlos Ramos Carreño
# License: MIT

import skfda
import skfda.representation.basis as basis
from skfda.representation.interpolation import SplineInterpolator
```

In this example we are going to show the different representations of functional data available in scikit-fda. First we are going to fetch a functional data dataset, such as the Berkeley Growth Study. This dataset correspond to the height of several boys and girls measured until the 18 years of age. The number and times of the measurements are the same for each individual.
```python
dataset = skfda.datasets.fetch_growth()
fd = dataset['data']
y = dataset['target']

print(repr(fd))

fd.plot(group=y, group_colors=['red', 'blue'])
```

Out:

```
FDataGrid(
    array([[ 81.3],
            [ 84.2],
            [ 86.4],
            ...
            [193.8],
            [194.3],
            [195.1]],

            [[ 76.2],
            [ 80.4],
            [ 83.2],
            ...

```
This kind of representation is a discretized representation, in which the measurement points are shared between samples.

2.9. Representation of functional data 237
```python
print(fd.sample_points)
```
Out:
```python
[array([1. , 1.25, 1.5 , 1.75, 2. , 3. , 4. , 5. , 6. , 7. , 8. , 8.5 ,
       9. , 9.5 , 10. , 10.5 , 11. , 11.5 , 12. , 12.5 , 13. , 13.5 ,
```
In this representation, the data can be arranged as a matrix.
```python
print(fd.data_matrix)
```
Out:
```python
[[ [ 81.3]
  [ 84.2]
  [ 86.4]
  ...
  [193.8]
  [194.3]
  [195.1]]

[[ [ 76.2]
  [ 80.4]
  [ 83.2]
  ...
  [176.1]
  [177.4]
  [178.7]]

[[ [ 76.8]
  [ 79.8]
  [ 82.6]
  ...
  [170.9]
  [171.2]
  [171.5]]

...

[[ [ 68.6]
  [ 73.6]
  [ 78.6]
  ...
  [166. ]
  [166.3]
  [166.8]]

[[ [ 79.9]
  [ 82.6]
  [ 84.8]
  ...
  [166. ]
  [166.3]
  [166.8]]
```
(continues on next page)
By default, the data points are interpolated using a linear interpolation, but this is configurable.

```python
dataset = skfda.datasets.fetch_medflies()
fd = dataset['data']
first_curve = fd[0]
first_curve.plot()
```

```
[[ 76.1]
 [ 78.4]
 [ 82.3]
 ...]
[168.6]
[168.9]
[169.2]]
```

Out:
The interpolation used can however be changed. Here, we will use an interpolation with degree 3 splines.

```python
first_curve.interpolator = SplineInterpolator(3)
first_curve.plot()
```

This representation allows also functions with arbitrary dimensions of the domain and codomain.

```python
fd = skfda.datasets.make_multimodal_samples(n_samples=1, dim_domain=2,
                                          dim_codomain=2)
print(fd.dim_domain)
print(fd.dim_codomain)
fd.plot()
```
Another possible representation is a decomposition in a basis of functions. $f(t) = \sum_{i=1}^{N} a_i \phi_i(t)$ It is possible to transform between both representations. Let us use again the Berkeley Growth dataset.

```python
dataset = skfda.datasets.fetch_growth()
fd = dataset['data']
y = dataset['target']

fd.plot()
```
We will represent it using a basis of B-splines.

```python
fd_basis = fd.to_basis(basis.BSpline(n_basis=4))
fd_basis.plot()
```
We can increase the number of elements in the basis to try to reproduce the original data with more fidelity.

```
fd_basis_big = fd.to_basis(basis.BSpline(n_basis=7))
fd_basis_big.plot()
```
Lets compare the different representations in the same plot, for the same curve

```python
fig = fd[0].plot()
fd_basis[0].plot(fig=fig)
fd_basis_big[0].plot(fig=fig)

fig.axes[0].legend(['Original', '4 elements', '7 elements'])
```
We can also see the effect of changing the basis. For example, in the Fourier basis the functions start and end at the same points if the period is equal to the domain range, so this basis is clearly non suitable for the Growth dataset.

```python
fd_basis = fd.to_basis(basis.Fourier(n_basis=7))
fd_basis.plot()
```
The data is now represented as the coefficients in the basis expansion.

```python
print(fd_basis)
```

Out:

```
FDataBasis(
    _basis=Fourier(domain_range=[[ 1. 18.], n_basis=7, period=17.0],
    coefficients=[[ 5.99308923e+02 -1.14873764e+02 2.75173660e+01 -5.99461262e+01
                   3.00407655e+01 -2.42099297e+01 3.06116238e+01]
                  [ 5.48897873e+02 -8.54081358e+01 1.29613544e+01 -4.69373290e+01
                    3.05139003e+01 -2.11534333e+01 3.43876128e+01]
                  [ 5.34606133e+02 -8.92928005e+01 2.19564051e+01 -4.88308206e+01
                    2.67601831e+01 -2.05310410e+01 2.75094852e+01]
                  [ 5.60189583e+02 -9.81740138e+01 2.19564051e+01 -4.88308206e+01
                    2.67601831e+01 -2.05310410e+01 2.75094852e+01]
                  [ 5.60189583e+02 -9.14228813e+01 1.79086286e+01 -5.05486070e+01
                    2.74899404e+01 -2.13354242e+01 2.92148972e+01]
                  [ 5.31961222e+02 -9.84166135e+01 2.29692209e+01 -5.08047387e+01
                    2.83861058e+01 -2.25286667e+01 3.14426375e+01]
                  [ 5.44375259e+02 -9.14228813e+01 1.79086286e+01 -5.05486070e+01
                    2.74899404e+01 -2.13354242e+01 2.92148972e+01]
```

(continues on next page)
(continued from previous page)

| 5.38760220e+02 | -8.78443853e+01 | 1.11447104e+01 | -4.92415698e+01 |
| 2.93566074e+01 | -2.19018332e+01 | 3.18418770e+01 |
| 5.74075270e+02 | -1.06735394e+02 | 2.93566074e+01 |
| 2.64575878e+01 | -2.38079035e+01 | 3.18418770e+01 |
| 5.36487114e+02 | -8.32523968e+01 | 1.11447104e+01 |
| 2.81948114e+01 | -2.11428475e+01 | 3.18418770e+01 |
| 5.95983096e+02 | -1.09237516e+02 | 2.93566074e+01 |
| 2.49614186e+01 | -2.26934866e+01 | 3.18418770e+01 |
| 5.57080765e+02 | -9.14783686e+01 | 1.11447104e+01 |
| 3.08468578e+01 | -2.58741704e+01 | 3.18418770e+01 |
| 5.79295854e+02 | -1.05128952e+02 | 2.93566074e+01 |
| 3.16530111e+01 | -2.32304890e+01 | 3.18418770e+01 |
| 6.16989062e+02 | -9.87617375e+01 | 1.11447104e+01 |
| 3.09546822e+01 | -2.34860283e+01 | 3.18418770e+01 |
| 5.80002600e+02 | -1.06359425e+02 | 2.93566074e+01 |
| 2.68841745e+01 | -2.46243600e+01 | 3.18418770e+01 |
| 5.76586832e+02 | -9.77828548e+01 | 1.11447104e+01 |
| 2.13361411e+01 | -2.14705636e+01 | 3.18418770e+01 |
| 5.76900386e+02 | -1.04096143e+02 | 2.93566074e+01 |
| 2.53397993e+01 | -2.32538346e+01 | 3.18418770e+01 |
| 5.69262456e+02 | -1.07160575e+02 | 2.93566074e+01 |
| 2.96883343e+01 | -2.27857514e+01 | 3.18418770e+01 |
| 5.72827912e+02 | -1.02730159e+02 | 2.93566074e+01 |
| 2.18407799e+01 | -1.79388600e+01 | 3.18418770e+01 |
| 5.42814074e+02 | -9.64583360e+01 | 1.11447104e+01 |
| 2.74183538e+01 | -2.39784066e+01 | 3.18418770e+01 |
| 5.54718900e+02 | -1.02314586e+02 | 2.93566074e+01 |
| 2.43879308e+01 | -2.17599864e+01 | 3.18418770e+01 |
| 5.40664948e+02 | -8.55292538e+01 | 1.11447104e+01 |
| 2.93851436e+01 | -2.09164947e+01 | 3.18418770e+01 |
| 5.43190236e+02 | -9.31881751e+01 | 1.11447104e+01 |
| 2.67885059e+01 | -2.27877886e+01 | 3.18418770e+01 |
| 5.48101974e+02 | -9.42605483e+01 | 1.11447104e+01 |
| 3.01148080e+01 | -2.51875112e+01 | 3.18418770e+01 |
| 5.59743797e+02 | -9.31891666e+01 | 1.11447104e+01 |
| 2.41687758e+01 | -1.97785861e+01 | 3.18418770e+01 |
| 5.45080584e+02 | -9.07019215e+01 | 1.11447104e+01 |
| 2.69158000e+01 | -2.15649788e+01 | 3.18418770e+01 |
| 5.56071285e+02 | -1.09701191e+02 | 1.11447104e+01 |
| 3.20481885e+01 | -2.30340669e+01 | 3.18418770e+01 |
| 5.65317909e+02 | -8.85619621e+01 | 1.11447104e+01 |
| 2.98196483e+01 | -2.13220966e+01 | 3.18418770e+01 |
| 5.32437188e+02 | -8.40135179e+01 | 1.11447104e+01 |
| 2.67772466e+01 | -2.89090212e+01 | 3.18418770e+01 |
| 5.06168309e+02 | -1.13998299e+02 | 2.14362628e+01 |
| 2.02123250e+01 | -2.45679365e+01 | 3.18418770e+01 |
| 5.46526265e+02 | -9.59141822e+01 | 1.11447104e+01 |
| 3.26084310e+01 | -2.42618221e+01 | 3.18418770e+01 |
| 5.70570928e+02 | -1.07306210e+02 | 2.14362628e+01 |
| 2.38801932e+01 | -2.14537891e+01 | 3.18418770e+01 |
| 5.98736539e+02 | -1.09836266e+02 | 2.14362628e+01 |
| 2.81480280e+01 | -2.32859370e+01 | 3.18418770e+01 |

(continues on next page)
```
[ 5.52219352e+02 -8.68132817e+01 1.73874116e+01 -4.62856832e+01
  2.80210530e+01 -2.22090130e+01 3.02535201e+01]
[ 5.45150662e+02 -9.57236717e+01 2.57514324e+01 -5.02866832e+01
  2.50527956e+01 -2.12796444e+01 3.02535201e+01]
[ 5.97211431e+02 -1.06850021e+02 1.32554219e+01 -5.17107091e+01
  2.55641465e+01 -2.51713173e+01 3.02535201e+01]
[ 5.61518039e+02 -1.04507220e+02 1.73437109e+01 -5.74511173e+01
  2.73371579e+01 -2.51743884e+01 3.02535201e+01]
[ 5.89181128e+02 -1.01765056e+02 1.96490191e+01 -5.13798171e+01
  2.62424568e+01 -1.98186766e+01 2.74849815e+01]
[ 5.7982454e+02 -1.05791220e+02 1.39358199e+01 -5.26716093e+01
  2.3694842e+01 -1.96474940e+01 2.87840205e+01]
[ 5.51407740e+02 -9.57356066e+01 1.92393603e+01 -5.04749182e+01
  2.63879989e+01 -2.28241092e+01 3.07542560e+01]
[ 5.39135250e+02 -8.88513179e+01 2.55144104e+00 -3.97960747e+01
  1.40580870e+01 -1.54917823e+01 2.59400829e+01]
[ 5.46244322e+02 -9.56132391e+01 9.23417894e+00 -4.42620585e+01
  1.4501201e+01 -1.6843571e+01 2.56146984e+01]
[ 5.60313637e+02 -9.15648753e+01 -5.82003275e+00 -3.83271136e+01
  1.7691238e+01 -1.82840495e+01 2.64319644e+01]
[ 5.52572545e+02 -9.71360646e+01 1.50556655e+01 -4.70741375e+01
  2.16358629e+01 -1.79074999e+01 2.58851254e+01]
[ 5.36934255e+02 -9.76897659e+01 1.28613949e+01 -4.54412544e+01
  1.75414041e+01 -1.65134308e+01 2.65471784e+01]
[ 5.19979422e+02 -8.57299537e+01 1.08433283e+00 -4.67556614e+01
  2.39816474e+01 -2.16429969e+01 2.87503736e+01]
[ 6.12204934e+02 -1.09601555e+02 1.31527081e+00 -4.55721145e+01
  1.83217318e+01 -2.1128813e+01 2.37639995e+01]
[ 5.0480200e+02 -9.51255290e+01 5.75751181e+00 -4.36021818e+01
  1.56118505e+01 -1.61905251e+01 2.67642244e+01]
[ 5.75273565e+02 -1.07385710e+02 1.28613949e+01 -4.54412544e+01
  1.77107011e+01 -1.89021366e+01 3.12706087e+01]
[ 5.39181810e+02 -8.74897261e+01 9.63381503e+00 -4.73582040e+01
  2.34098312e+01 -2.2053642e+01 2.72025837e+01]
[ 5.38321252e+02 -9.25717805e+01 6.44366835e+00 -4.74686455e+01
  1.75843207e+01 -1.83972642e+01 2.62746131e+01]
[ 5.00140286e+02 -7.9292834e+01 9.20713731e+00 -4.2206682e+01
  2.13577811e+01 -1.96794009e+01 2.51025814e+01]
[ 5.52970660e+02 -9.28711825e+01 1.12721168e+01 -4.76036376e+01
  2.26887239e+01 -1.95380353e+01 2.69499961e+01]
[ 5.61345841e+02 -1.07035732e+02 4.54667968e+00 -4.58593784e+01
  2.11905849e+01 -1.47219200e+01 2.90074639e+01]
[ 5.10558942e+02 -8.11616379e+01 1.06513156e+01 -4.2197638e+01
  1.70349388e+01 -1.58033350e+01 2.46077602e+01]
[ 5.43627855e+02 -9.68349026e+01 -1.22877454e+00 -3.90608220e+01
  1.43854996e+01 -1.79044988e+01 2.70921791e+01]
[ 5.77161078e+02 -9.8533632e+01 1.20014603e-01 -4.39834039e+01
  1.71144994e+01 -1.66659377e+01 2.73868569e+01]
[ 5.49485081e+02 -1.00664731e+02 4.94702320e+00 -4.37712750e+01
  1.49573258e+01 -1.70420508e+01 3.0106654e+01]
```

The table below contains the data:

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
<th>X5</th>
<th>X6</th>
<th>X7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5910804e+02</td>
<td>-9.8770628e+01</td>
<td>8.26596278e+00</td>
<td>-4.62067956e+01</td>
<td>1.77503711e+01</td>
<td>-1.87062333e+01</td>
<td>2.84358540e+01</td>
</tr>
<tr>
<td>5.65003297e+02</td>
<td>-1.06435308e+02</td>
<td>5.49420625e+00</td>
<td>-4.85466595e+01</td>
<td>1.88716609e+01</td>
<td>-1.73550262e+01</td>
<td>2.98855083e+01</td>
</tr>
<tr>
<td>5.38255284e+02</td>
<td>-9.59331252e+01</td>
<td>6.52369178e+00</td>
<td>-4.21452697e+01</td>
<td>1.7639944e+01</td>
<td>-1.53534156e+01</td>
<td>2.81154045e+01</td>
</tr>
<tr>
<td>5.2370065e+02</td>
<td>-8.95406540e+01</td>
<td>6.30223624e+00</td>
<td>-4.30799760e+01</td>
<td>1.62142161e+01</td>
<td>-1.56457097e+01</td>
<td>2.69396619e+01</td>
</tr>
<tr>
<td>5.66950202e+01</td>
<td>-1.42799947e+01</td>
<td>1.84983994e+00</td>
<td>-5.2639178e+01</td>
<td>1.59712192e+01</td>
<td>-1.07954730e+01</td>
<td>2.92125040e+01</td>
</tr>
<tr>
<td>5.13426363e+01</td>
<td>-1.80018899e+00</td>
<td>1.98254292e+00</td>
<td>-4.38539357e+01</td>
<td>1.85658856e+01</td>
<td>-1.65421271e+01</td>
<td>2.41213457e+01</td>
</tr>
<tr>
<td>5.52621888e+01</td>
<td>-9.21345904e+01</td>
<td>1.89079196e+01</td>
<td>-4.38539357e+01</td>
<td>1.69308410e+01</td>
<td>-1.71481129e+01</td>
<td>2.73311887e+01</td>
</tr>
<tr>
<td>5.4964443e+01</td>
<td>-9.27854148e+01</td>
<td>1.48112465e+00</td>
<td>-4.75482728e+01</td>
<td>1.43935455e+01</td>
<td>-1.76356094e+01</td>
<td>2.67026063e+01</td>
</tr>
<tr>
<td>5.00705442e+01</td>
<td>-8.53092980e+00</td>
<td>1.43470009e+00</td>
<td>-4.40308172e+01</td>
<td>2.08833472e+01</td>
<td>-1.78026303e+00</td>
<td>2.47683865e+01</td>
</tr>
<tr>
<td>5.46098900e+01</td>
<td>-9.39415718e+00</td>
<td>1.30284787e+00</td>
<td>-4.79309436e+01</td>
<td>1.93794660e+01</td>
<td>-1.78586939e+00</td>
<td>2.61801955e+01</td>
</tr>
<tr>
<td>5.46913499e+01</td>
<td>-8.93705300e+00</td>
<td>8.52549272e+00</td>
<td>-4.52328860e+01</td>
<td>1.8680044e+01</td>
<td>-1.62722943e+01</td>
<td>2.63048681e+01</td>
</tr>
<tr>
<td>5.53660929e+01</td>
<td>-9.94032360e+00</td>
<td>8.52549272e+00</td>
<td>-4.52328860e+01</td>
<td>1.51694656e+01</td>
<td>-1.85739527e+01</td>
<td>2.97632580e+01</td>
</tr>
<tr>
<td>5.59813837e+01</td>
<td>-9.49280033e+00</td>
<td>1.15486892e+00</td>
<td>-4.95937257e+01</td>
<td>2.21884722e+01</td>
<td>-2.07278596e+00</td>
<td>2.73240069e+01</td>
</tr>
<tr>
<td>5.42647879e+01</td>
<td>-9.36782615e+00</td>
<td>7.39169942e+00</td>
<td>-3.82180712e+01</td>
<td>1.56916538e+01</td>
<td>-1.69621653e+00</td>
<td>2.65022153e+01</td>
</tr>
<tr>
<td>5.57556323e+01</td>
<td>-1.00060254e+00</td>
<td>5.46702364e+00</td>
<td>-4.72428033e+01</td>
<td>1.97932991e+01</td>
<td>-1.72037367e+00</td>
<td>2.79946719e+01</td>
</tr>
<tr>
<td>5.40474747e+01</td>
<td>-9.45361518e+00</td>
<td>1.06497154e+00</td>
<td>-4.67932237e+01</td>
<td>1.93909626e+01</td>
<td>-1.80518300e+00</td>
<td>2.58176113e+01</td>
</tr>
<tr>
<td>5.2676001e+01</td>
<td>-9.26407816e+00</td>
<td>1.14811489e+00</td>
<td>-4.81163405e+01</td>
<td>2.17720423e+01</td>
<td>-2.04315426e+00</td>
<td>2.69207214e+01</td>
</tr>
<tr>
<td>5.80189078e+02</td>
<td>-1.07654945e+00</td>
<td>3.55013740e+00</td>
<td>-4.93143073e+01</td>
<td>2.04402466e+01</td>
<td>-1.98081935e+00</td>
<td>3.06725691e+01</td>
</tr>
<tr>
<td>5.62261092e+02</td>
<td>-1.02538949e+00</td>
<td>7.24425492e+00</td>
<td>-4.73770964e+01</td>
<td>1.71637520e+01</td>
<td>-1.93611070e+00</td>
<td>2.82762388e+01</td>
</tr>
<tr>
<td>5.64264106e+02</td>
<td>-1.00658789e+00</td>
<td>5.01547652e+00</td>
<td>-3.45126326e+01</td>
<td>1.84211522e+01</td>
<td>-1.66923320e+00</td>
<td>2.81219876e+01</td>
</tr>
<tr>
<td>5.39635405e+02</td>
<td>-9.57912620e+00</td>
<td>5.91820048e+00</td>
<td>-4.29163832e+01</td>
<td>1.63854379e+01</td>
<td>-1.79586073e+00</td>
<td>2.70622725e+01</td>
</tr>
<tr>
<td>5.04454697e+02</td>
<td>-8.86624547e+00</td>
<td>1.07196974e+00</td>
<td>-4.24950760e+01</td>
<td>1.72544787e+01</td>
<td>-1.60229907e+00</td>
<td>2.39382266e+01</td>
</tr>
<tr>
<td>5.73010226e+02</td>
<td>-1.01034303e+00</td>
<td>6.30609704e+00</td>
<td>-4.65098937e+01</td>
<td>1.46097690e+01</td>
<td>-1.88507258e+00</td>
<td>2.98536969e+01</td>
</tr>
<tr>
<td>5.52457923e+02</td>
<td>-9.36222927e+00</td>
<td>8.91452791e+00</td>
<td>-4.64958302e+01</td>
<td>1.89624335e+01</td>
<td>-1.66408567e+00</td>
<td>2.59420577e+01</td>
</tr>
<tr>
<td>5.41903921e+02</td>
<td>-9.44261431e+00</td>
<td>8.51705002e+00</td>
<td>-4.58494268e+01</td>
<td>1.64118507e+01</td>
<td>-1.58280167e+00</td>
<td>2.79485257e+01</td>
</tr>
</tbody>
</table>

(continues on next page)
2.10 Boxplot

Shows the use of the functional Boxplot applied to the Canadian Weather dataset.

```python
# Author: Amanda Hernando Bernabé
# License: MIT

# sphinx_gallery_thumbnail_number = 2

import matplotlib.pyplot as plt
import numpy as np
from skfda.datasets import fetch_weather
from skfda import BandDepth, FraimanMunizDepth
from skfda.exploratory.visualization import Boxplot

First, the Canadian Weather dataset is downloaded from the package ‘fda’ in CRAN. It contains a FDataGrid with daily temperatures and precipitations, that is, it has a 2-dimensional image. We are interested only in the daily average temperatures, so we will use the first coordinate.

dataset = datasets.fetch_weather()
fd = dataset['data']["data"]
fd_temperatures = fd.coordinates[0]
```

The data is plotted to show the curves we are working with. They are divided according to the target. In this case, it includes the different climates to which the weather stations belong to.
Each climate is assigned a color. Defaults to grey.

colormap = plt.cm.get_cmap('seismic')
label_names = dataset["target_names"]
nlabels = len(label_names)
label_colors = colormap(np.arange(nlabels) / (nlabels - 1))

fd_temperatures.plot(group=dataset["target"],
                       group_colors=label_colors,
                       group_names=label_names)

Out:

<Figure size 640x480 with 1 Axes>

We instantiate a Boxplot object with the data, and we call its plot() function to show the graph.

By default, only the part of the outlier curves which falls out of the central regions is plotted. We want the entire curve to be shown, that is why the show_full_outliers parameter is set to True.

fdBoxplot = Boxplot(fd_temperatures)
fdBoxplot.show_full_outliers = True

fdBoxplot.plot()
We can observe in the boxplot the median in black, the central region (where the 50% of the most centered samples reside) in pink and the envelopes and vertical lines in blue. The outliers detected, those samples with at least a point outside the outlying envelope, are represented with a red dashed line. The colors can be customized.

The outliers are shown below with respect to the other samples.

color = 0.3
outliercol = 0.7

fd_temperatures.plot(group=fdBoxplot.outliers.astype(int),
group_colors=colormap([color, outliercol]),
group_names=['nonoutliers', 'outliers'])
The curves pointed as outliers are those curves with significantly lower values than the rest. This is the expected result due to the depth measure used, `fraiman_muniz_depth()`, which ranks the samples according to their magnitude.

The `Boxplot` object admits any depth measure defined or customized by the user. Now the call is done with the `band_depth()` and the factor is reduced in order to designate some samples as outliers (otherwise, with this measure and the default factor, none of the curves are pointed out as outliers). We can see that the outliers detected belong to the Pacific and Arctic climates which are less common to find in Canada. As a consequence, this measure detects better shape outliers compared to the previous one.

```python
fdBoxplot = Boxplot(fd_temperatures, depth_method=band_depth, factor=0.4)
fdBoxplot.show_full_outliers = True
fdBoxplot.plot()
```
Another functionality implemented in this object is the enhanced functional boxplot, which can include other central regions, apart from the central or 50\% one.

In the following instantiation, the \texttt{fraiman\_muniz\_depth()} is used and the 25\% and 75\% central regions are specified.

```python
fdBoxplot = Boxplot(fd_temperatures, depth_method=fraiman_muniz_depth,
                     prob=[0.75, 0.5, 0.25])
fdBoxplot.plot()
```
Out:

```
<Figure size 640x480 with 1 Axes>
```

The above two lines could be replaced just by `fdBoxplot` inside a notebook since the default representation of the `Boxplot` is the image of the plot.

**Total running time of the script:** ( 0 minutes 1.192 seconds)

---

**Note:** Click [here](#) to download the full example code

---

### 2.11 Neighbors Scalar Regression

Shows the usage of the nearest neighbors regressor with scalar response.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

from sklearn.model_selection import train_test_split, GridSearchCV, KFold
```

(continues on next page)
In this example, we are going to show the usage of the nearest neighbors regressors with scalar response. There is available a K-nn version, \texttt{KNeighborsRegressor}, and other one based in the radius, \texttt{RadiusNeighborsRegressor}.

Firstly we will fetch a dataset to show the basic usage.

The Canadian weather dataset contains the daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994.

The following figure shows the different temperature and precipitation curves.

```python
import matplotlib.pyplot as plt
import numpy as np

import skfda
from skfda.datasets import fetch_weather

data = skfda.datasets.fetch_weather()
fd = data['data']

# Split dataset, temperatures and curves of precipitation
X, y_func = fd.coordinates

Temperatures

X.plot()
```
Out:

```python
y_func.plot()
```

Precipitation
We will try to predict the total log precipitation, i.e, $\log PrecTot_i = \log \sum_{t=0}^{365} prec_i(t)$ using the temperature curves.

```python
# Sum directly from the data matrix
prec = y_func.data_matrix.sum(axis=1)[:, 0]
log_prec = np.log(prec)
print(log_prec)
```

As in the nearest neighbors classifier examples, we will split the dataset in two partitions, for training and test, using the sklearn function `train_test_split()`.
Firstly we will try make a prediction with the default values of the estimator, using 5 neighbors and the $L^2$ distance.

We can fit the `KNeighborsRegressor` in the same way than the sklearn estimators. This estimator is an extension of the sklearn `KNeighborsRegressor`, but accepting a `FDataGrid` as input instead of an array with multivariate data.

```python
knn = KNeighborsRegressor(weights='distance')
knn.fit(X_train, y_train)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='l2',
                    metric_params=None, multivariate_metric=False, n_jobs=1,
                    n_neighbors=5, regressor='mean', weights='distance')
```

We can predict values for the test partition using `predict()`.

```python
pred = knn.predict(X_test)
pred
```

```
[7.11225785 5.99768933 7.05559273 6.88718564 6.78535172 5.97132028
  6.56125279 6.47991884 6.92965595]
```

The following figure compares the real precipitations with the predicted values.

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(y_test, pred)
ax.plot(y_test, y_test)
ax.set_xlabel("Total log precipitation")
ax.set_ylabel("Prediction")
```
We can quantify how much variability it is explained by the model with the coefficient of determination $R^2$ of the prediction, using `score()` for that.

The coefficient $R^2$ is defined as $(1 - u/v)$, where $u$ is the residual sum of squares $\sum_i (y_i - y_{pred})^2$ and $v$ is the total sum of squares $\sum_i (y_i - \bar{y})^2$.

```python
score = knn.score(X_test, y_test)
print(score)
```

Out:

```
0.9244558571515656
```

In this case, we obtain a really good approximation with this naive approach, although, due to the small number of samples, the results will depend on how the partition was done. In the above case, the explained variation is inflated for this reason.

We will perform cross-validation to test more robustly our model.

As in the neighbors classifiers examples, we can use a sklearn metric to approximate the $L^2$ metric between function, but with a much lower computational cost.
Also, we can make a grid search, using GridSearchCV, to determine the optimal number of neighbors and the best way to weight their votes.

```python
param_grid = {'n_neighbors': np.arange(1, 12, 2),
              'weights': ['uniform', 'distance']}

knn = KNeighborsRegressor(metric='euclidean', multivariate_metric=True)
gscv = GridSearchCV(knn, param_grid, cv=KFold(n_splits=3,
                                               shuffle=True, random_state=0))
gscv.fit(X, log_prec)
```

Out:

```python
GridSearchCV(cv=KFold(n_splits=3, random_state=0, shuffle=True),
            error_score=nan,
            estimator=KNeighborsRegressor(algorithm='auto', leaf_size=30,
                                           metric='euclidean',
                                           metric_params=None,
                                           multivariate_metric=True, n_jobs=1,
                                           n_neighbors=5, regressor='mean',
                                           weights='uniform'),
            iid='deprecated', n_jobs=None,
            param_grid={'n_neighbors': array([ 1, 3, 5, 7, 9, 11]),
                        'weights': ['uniform', 'distance']},
            pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
            scoring=None, verbose=0)
```

We obtain that 7 is the optimal number of neighbors.

```python
print("Best params", gscv.best_params_)
print("Best score", gscv.best_score_)
```

Out:

```text
Best params {'n_neighbors': 7, 'weights': 'distance'}
Best score 0.5345582611225209
```

More detailed information about the Canadian weather dataset can be obtained in the following references.


**Total running time of the script:** (0 minutes 1.154 seconds)

**Note:** Click [here](#) to download the full example code
2.12 Landmark shift

This example shows how to shift functional data objects to align its samples with a particular reference point.

```python
# Author: Pablo Marcos Manchón
# License: MIT
# sphinx_gallery_thumbnail_number = 2

import matplotlib.pyplot as plt
import numpy as np
import skfda

We will use an example dataset synthetically generated by `make_multimodal_samples()`, which in this case will be used to generate gaussian-like samples with a mode near to 0. Each sample will be shifted to align their modes to a reference point using the function `landmark_shift()`.

```python
fd = skfda.datasets.make_multimodal_samples(random_state=1)
fd.extrapolation = 'bounds'  # See extrapolation for a detailed explanation.

fd.plot()
```
A landmark or a feature of a curve is some characteristic that one can associate with a specific argument value $t$. These are typically maxima, minima, or zero crossings of curves, and may be identified at the level of some derivatives as well as at the level of the curves themselves [RaSi2005-2].

For alignment we need to know in advance the location of the landmark of each of the samples, in our case it will correspond to the maxima of each sample. Because our dataset has been generated synthetically we can obtain the value of the landmarks using the function `make_multimodal_landmarks()`, which is used by `make_multimodal_samples()` to set the location of the modes.

In general it will be necessary to use numerical or other methods to determine the location of the landmarks.

```python
landmarks = skfda.datasets.make_multimodal_landmarks(random_state=1).squeeze()

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(landmarks, np.repeat(1, fd.n_samples))
fd.plot(fig=fig)
```

Out:
Location of the landmarks:

```python
print(landmarks)
```

Out:

```
[ 0.36321467 -0.13679289 -0.11810279 -0.23992308 0.19351103 -0.5146397
 0.39015177 -0.17021104 0.07133931 -0.05576091 0.32693727 -0.46066147
-0.07209468 -0.08587716 0.25351855]
```

The following figure shows the result of shifting the curves to align their landmarks at 0.

```python
fd_registered = skfda.preprocessing.registration.landmark_shift(
    fd, landmarks, location=0)
fig = fd_registered.plot()
fig.axes[0].scatter(0, 1)
```

Out:

```
<matplotlib.collections.PathCollection object at 0x7f8a3122f748>
```

In many circumstances it is possible that we could not apply extrapolation, in these cases it is possible to restrict the domain to avoid evaluating points outside where our curves are defined.
If the location of the new reference point is not specified it is chosen the point that minimizes the maximum amount of shift.

```python
# Curves aligned restricting the domain
fd_restricted = skfda.preprocessing.registration.landmark_shift(
    fd, landmarks, restrict_domain=True)

# Curves aligned to default point without restrict domain
fd_extrapolated = skfda.preprocessing.registration.landmark_shift(
    fd, landmarks)

fig = fd_extrapolated.plot(linestyle='dashed', label='Extrapolated samples')
fd_restricted.plot(fig=fig, label="Restricted samples")
```

Out:

<Figure size 640x480 with 1 Axes>

The previous method is also applicable for multidimensional objects, without limitation of the domain or image dimension. As an example we are going to create a dataset with surfaces, in a similar way to the previous case.

```python
fd = skfda.datasets.make_multimodal_samples(n_samples=3, points_per_dim=30,
(continues on next page)
In this case the landmarks will be defined by tuples with 2 coordinates.

```python
landmarks = skfda.datasets.make_multimodal_landmarks(n_samples=3, dim_domain=2, random_state=1).squeeze()
print(landmarks)
```

```
[[ 0.36321467 -0.13679289]
 [-0.11810279 -0.23992308]
 [ 0.19351103 -0.5146397 ]]
```

As in the previous case, we can align the curves to a specific point, or by default will be chosen the point that minimizes the maximum amount of displacement.
```python
fd_registered = skfda.preprocessing.registration.landmark_shift(fd, landmarks)
fd_registered.plot()
plt.show()
```

**Total running time of the script:** 0 minutes 1.266 seconds

**Note:** Click [here](#) to download the full example code

### 2.13 K-nearest neighbors classification

Shows the usage of the k-nearest neighbors classifier.

```python
# Author: Pablo Marcos Manchón
# License: MIT

from sklearn.model_selection import (train_test_split, GridSearchCV,
                                     StratifiedShuffleSplit)
```

(continues on next page)
```python
import matplotlib.pyplot as plt
import numpy as np
import skfda
from skfda.ml.classification import KNeighborsClassifier
```

In this example we are going to show the usage of the K-nearest neighbors classifier in their functional version, which is a extension of the multivariate one, but using functional metrics between the observations.

Firstly, we are going to fetch a functional dataset, such as the Berkeley Growth Study. This dataset contains the height of several boys and girls measured until the 18 years of age. We will try to predict the sex by using its growth curves.

The following figure shows the growth curves grouped by sex.

Loads dataset

```python
data = skfda.datasets.fetch_growth()
X = data['data']
y = data['target']
class_names = data['target_names']

# Plot samples grouped by sex
X.plot(group=y, group_names=class_names)
```

![Berkeley Growth Study](image)

Out:
In this case, the class labels are stored in an array with 0’s in the male samples and 1’s in the positions with female ones.

```python
print(y)
```

Out:

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

We can split the dataset using the sklearn function `train_test_split()`.

The function will return two `FDataGrid`s, `X_train` and `X_test` with the corresponding partitions, and arrays with their class labels.

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, random_state=0)
```

We will fit the classifier `KNeighborsClassifier` with the training partition. This classifier works exactly like the sklearn multivariate classifier `KNeighborsClassifier`, but will accept as input a `FDataGrid` with functional observations instead of an array with multivariate data.

```python
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
```

Out:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='l2', metric_params=None, multivariate_metric=False, n_jobs=1, n_neighbors=5, weights='uniform')
```

Once it is fitted, we can predict labels for the test samples.

To predict the label of a test sample, the classifier will calculate the k-nearest neighbors and will assign the majority class. By default, it is used the $L^2$ distance between functions, to determine the neighbourhood of a sample, with 5 neighbors. Can be used any of the functional metrics described in `Metrics`.

```python
pred = knn.predict(X_test)
print(pred)
```

Out:

```
[0 0 1 0 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1]
```

The `score()` method allows us to calculate the mean accuracy for the test data. In this case we obtained around 96% of accuracy.

```python
score = knn.score(X_test, y_test)
print(score)
```

Out:
We can also estimate the probability of membership to the predicted class using `predict_proba()`, which will return an array with the probabilities of the classes, in lexicographic order, for each test sample.

```python
probs = knn.predict_proba(X_test[:5])  # Predict first 5 samples
print(probs)
```

```
[[1.  0. ]
 [0.6 0.4]
 [0.  1. ]
 [1.  0. ]
 [0.  1. ]]
```

We can use the sklearn `GridSearchCV` to perform a grid search to select the best hyperparams, using cross-validation.

In this case, we will vary the number of neighbors between 1 and 11.

```python
# Only odd numbers, to prevent ties
param_grid = {'n_neighbors': np.arange(1, 12, 2)}

knn = KNeighborsClassifier()

# Perform grid search with cross-validation
ss = StratifiedShuffleSplit(n_splits=5, test_size=.25, random_state=0)
gscv = GridSearchCV(knn, param_grid, cv=ss)
gscv.fit(X, y)

print("Best params:", gscv.best_params_)
print("Best score:", gscv.best_score_)
```

```
Best params: {'n_neighbors': 11}
Best score: 0.975
```

We have obtained the greatest mean accuracy using 11 neighbors. The following figure shows the score depending on the number of neighbors.

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.bar(param_grid['n_neighbors'], gscv.cv_results_['mean_test_score'])
ax.set_xticks(param_grid['n_neighbors'])
ax.set_ylabel("Number of Neighbors")
ax.set_xlabel("Test score")
ax.set_ylim((0.9, 1))
```
When the functional data have been sampled in an equispaced way, or approximately equispaced, it is possible to use the scikit-learn vector metrics with similar results.

For example, in the case of the $L^2$ distance, if the integral of the distance it is approximated as a Riemann sum, we obtain that it is proportional to the euclidean distance between vectors.

$$\|f - g\|_{L^2} = \left( \int_a^b |f(x) - g(x)|^2 \, dx \right)^{\frac{1}{2}} \approx \left( \sum_{n=0}^{N} \Delta h |f(x_n) - g(x_n)|^2 \right)^{\frac{1}{2}} = \sqrt{\Delta h} d_{\text{euclidean}}(\vec{f}, \vec{g})$$

So, in this case, it is roughly equivalent to use this metric instead of the functional one, due to the constant multiplication not affecting the order of the neighbors.

Setting the parameter sklearn_metric of the classifier to True, a vectorial metric of sklearn can be passed. In DistanceMetric there are listed all the metrics supported.

We will fit the model with the sklearn distance and search for the best parameter. The results can vary slightly, due to the approximation during the integration, but the result should be similar.

```python
knn = KNeighborsClassifier(metric='euclidean', multivariate_metric=True)
gscv2 = GridSearchCV(knn, param_grid, cv=ss)
```
gscv2.fit(X, y)

print("Best params: ", gscv2.best_params_)
print("Best score: ", gscv2.best_score_)

Out:

Best params: 
{'n_neighbors': 5}
Best score: 0.975

The advantage of using the scikit-learn metrics is the computational speed, three orders of magnitude faster. But it is not always possible to have equispaced samples nor do all functional metrics have a vector equivalent in this way.

The mean score time depending on the metric is shown below.

print("Mean score time (milliseconds)")
print("L2 distance: ", 1000 * np.mean(gscv.cv_results_['mean_score_time']), "(ms)")
print("Euclidean distance: ", 1000 * np.mean(gscv2.cv_results_['mean_score_time']), "(ms)")

Out:

Mean score time (milliseconds)
L2 distance: 430.81672191619873 (ms)
Euclidean distance: 1.7486174901326499 (ms)

This classifier can be used with multivariate functional data, as surfaces or curves in \( \mathbb{R}^N \), if the metric supports it too.

**Total running time of the script:** (0 minutes 15.735 seconds)

**Note:** Click [here](#) to download the full example code

### 2.14 Magnitude-Shape Plot

Shows the use of the MS-Plot applied to the Canadian Weather dataset.

```python
# Author: Amanda Hernando Bernabé
# License: MIT
# sphinx_gallery_thumbnail_number = 2

import matplotlib.pyplot as plt
import numpy as np
from skfda import datasets
from skfda.exploratory.depth import fraiman_muniz_depth, modified_band_depth
from skfda.exploratory.visualization import MagnitudeShapePlot
```
First, the Canadian Weather dataset is downloaded from the package 'fda' in CRAN. It contains a FDataGrid with daily temperatures and precipitations, that is, it has a 2-dimensional image. We are interested only in the daily average temperatures, so we extract the first coordinate.

```python
dataset = datasets.fetch_weather()
f = dataset['data']
fd_temperatures = fd.coordinates[0]
```

The data is plotted to show the curves we are working with. They are divided according to the target. In this case, it includes the different climates to which the weather stations belong.

```python
# Each climate is assigned a color. Defaults to grey.
colormap = plt.cm.get_cmap('seismic')
label_names = dataset['target_names']
nlabels = len(label_names)
label_colors = colormap(np.arange(nlabels) / (nlabels - 1))

fd_temperatures.plot(group=dataset['target'],
                     group_colors=label_colors,
                     group_names=label_names)
```

Out:

<Figure size 640x480 with 1 Axes>

2.14. Magnitude-Shape Plot
The MS-Plot is generated. In order to show the results, the `plot()` method is used. Note that the colors have been specified before to distinguish between outliers or not. In particular the tones of the default colormap, (which is ‘seismic’ and can be customized), are assigned.

```python
msplot = MagnitudeShapePlot(fdatagrid=fd_temperatures,
                           depth_method=modified_band_depth)

color = 0.3
outliercol = 0.7

msplot.color = color
msplot.outliercol = outliercol
msplot.plot()
```

Out:

![Figure size 640x480 with 1 Axes]

To show the utility of the plot, the curves are plotted according to the distinction made by the MS-Plot (outliers or not) with the same colors.

```python
fd_temperatures.plot(group=msplot.outliers.astype(int),
                     group_colors=msplot.colormap([color, outliercol]),
                     group_names=['nonoutliers', 'outliers'])
```
We can observe that most of the curves pointed as outliers belong either to the Pacific or Arctic climates which are not the common ones found in Canada. The Pacific temperatures are much smoother and the Arctic ones much lower, differing from the rest in shape and magnitude respectively.

There are two curves from the Arctic climate which are not pointed as outliers but in the MS-Plot, they appear further left from the central points. This behaviour can be modified specifying the parameter alpha.

Now we use the pointwise `fraiman_muniz_depth()` in the MS-Plot.

```python
msplot = MagnitudeShapePlot(fdatagrid=fd_temperatures,
                            depth_method=fraiman_muniz_depth)
msplot.color = color
msplot.outliercol = outliercol
msplot.plot()
```
We can observe that almost none of the samples are pointed as outliers. Nevertheless, if we group them in three groups according to their position in the MS-Plot, the result is the expected one. Those samples at the left (larger deviation in the mean directional outlyingness) correspond to the Arctic climate, which has lower temperatures, and those on top (larger deviation in the directional outlyingness) to the Pacific one, which has smoother curves.

```python
group1 = np.where(msplot.points[:, 0] < -0.6)
group2 = np.where(msplot.points[:, 1] > 0.12)

colors = np.copy(msplot.outliers).astype(float)
colors[:] = color
colors[group1] = outliercol
colors[group2] = 0.9

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(msplot.points[:, 0], msplot.points[:, 1], c=colormap(colors))
ax.set_title("MS-Plot")
ax.set_xlabel("magnitude outlyingness")
ax.set_ylabel("shape outlyingness")
```

(continues on next page)
labels = np.copy(msplot.outliers.astype(int))
labels[group1] = 1
labels[group2] = 2

We now plot the curves with their corresponding color:

```python
fd_temperatures.plot(group=labels,
                      group_colors=colormap([color, outliercol, 0.9]))
```
2.15 Kernel Smoothing

This example uses different kernel smoothing methods over the phoneme data set and shows how cross validations scores vary over a range of different parameters used in the smoothing methods. It also show examples of undersmoothing and oversmoothing.

```python
# Author: Miguel Carbajo Berrocal
# License: MIT

import matplotlib.pylab as plt
import numpy as np
import skfda
```

(continues on next page)
import skfda.preprocessing.smoothing.kernel_smoothers as ks
import skfda.preprocessing.smoothing.validation as val

For this example, we will use the phoneme dataset. This dataset contains the log-periodograms of several phoneme pronunciations. The phoneme curves are very irregular and noisy, so we usually will want to smooth them as a preprocessing step.

As an example, we will smooth the first 300 curves only. In the following plot, the first five curves are shown.

dataset = skfda.datasets.fetch_phoneme()
fds = dataset['data'][:300]
fds[0:5].plot()

Out:

<Figure size 640x480 with 1 Axes>

Here we show the general cross validation scores for different values of the parameters given to the different smoothing methods.

param_values = np.linspace(start=2, stop=25, num=24)
# Local linear regression kernel smoothing.
llr = val.SmoothingParameterSearch(
    ks.LocalLinearRegressionSmoother(), param_values)
llr.fit(fd)
llr_fd = llr.transform(fd)

# Nadaraya-Watson kernel smoothing.
nw = val.SmoothingParameterSearch(
    ks.NadarayaWatsonSmoother(), param_values)
nw.fit(fd)
nw_fd = nw.transform(fd)

# K-nearest neighbours kernel smoothing.
knn = val.SmoothingParameterSearch(
    ks.KNeighborsSmoother(), param_values)
knn.fit(fd)
knn_fd = knn.transform(fd)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(param_values, knn.cv_results_['mean_test_score'],
    label='k-nearest neighbors')
ax.plot(param_values, llr.cv_results_['mean_test_score'],
    label='local linear regression')
ax.plot(param_values, nw.cv_results_['mean_test_score'],
    label='Nadaraya-Watson')
ax.legend()
fig
We can plot the smoothed curves corresponding to the 11th element of the data set (this is a random choice) for the three different smoothing methods.

```python
fig = plt.figure()
asx = fig.add_subplot(1, 1, 1)
asx.set_xlabel('Smoothing method parameter')
asx.set_ylabel('GCV score')
asx.set_title('Scores through GCV for different smoothing methods')
fd[10].plot(fig=fig)
knn_fd[10].plot(fig=fig)
llr_fd[10].plot(fig=fig)
nw_fd[10].plot(fig=fig)
asx.legend(['original data', 'k-nearest neighbors', 'local linear regression', 'Nadaraya-Watson'], title='Smoothing method')
fig
```

2.15. Kernel Smoothing
We can compare the curve before and after the smoothing.

Not smoothed

```python
fd[10].plot()
```
Smoothed

```
fig = fd[10].scatter(s=0.5)
nw_fd[10].plot(fig=fig, color='green')
fig
```
Now, we can see the effects of a proper smoothing. We can plot the same 5 samples from the beginning using the Nadaraya-Watson kernel smoother with the best choice of parameter.

```python
nw_fd[0:5].plot()
```
We can also appreciate the effects of undersmoothing and oversmoothing in the following plots.

```python
fd_us = ks.NadarayaWatsonSmoother(smoothing_parameter=2).fit_transform(fd[10])
fd_os = ks.NadarayaWatsonSmoother(smoothing_parameter=15).fit_transform(fd[10])
```

Under-smoothed

```python
fig = fd[10].scatter(s=0.5)
fd_us.plot(fig=fig)
```
Out:

```python
fig = fd[10].scatter(s=0.5)
fd_os.plot(fig=fig)
```

Over-smoothed
2.16 Interpolation

This example shows the types of interpolation used in the evaluation of FDataGrids.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)
import skfda
from skfda.representation.interpolation import SplineInterpolator

The `FDataGrid` class is used for datasets containing discretized functions. For the evaluation between the points of discretization, or sample points, it is necessary to interpolate.

We will construct an example dataset with two curves with 6 points of discretization.

```python
fd = skfda.datasets.make_sinusoidal_process(n_samples=2, n_features=6, random_state=1)
fig = fd.scatter()
fig.legend(["Sample 1", "Sample 2"])
```

Out:

```
<matplotlib.legend.Legend object at 0x7f8a313736a0>
```

By default it is used linear interpolation, which is one of the simplest methods of interpolation and therefore one of the least computationally expensive, but has the disadvantage that the interpolant is not differentiable at the points of discretization.

```python
fig = fd.plot()
fd.scatter(fig=fig)
```
The interpolation method of the FDataGrid could be changed setting the attribute `interpolator`. Once we have set an interpolator it is used for the evaluation of the object.

Polynomial spline interpolation could be performed using the interpolator `SplineInterpolator`. In the following example a cubic interpolator is set.

```python
fd.interpolator = SplineInterpolator(interpolation_order=3)
fig = fd.plot()
fd.scatter(fig=fig)
```
Smooth interpolation could be performed with the attribute `smoothness_parameter` of the spline interpolator.

```python
# Sample with noise
fd_smooth = skfda.datasets.make_sinusoidal_process(n_samples=1, n_features=30,
                                                 random_state=1, error_std=.3)

# Cubic interpolator
fd_smooth.interpolator = SplineInterpolator(interpolation_order=3)
fig = fd_smooth.plot(label="Cubic")

# Smooth interpolation
fd_smooth.interpolator = SplineInterpolator(interpolation_order=3,
                                             smoothness_parameter=1.5)
fd_smooth.plot(fig=fig, label="Cubic smoothed")
fd_smooth.scatter(fig=fig)
fig.legend()
```
It is possible to evaluate derivatives of the FDatagrid, but due to the fact that interpolation is performed first, the interpolation loses one degree for each order of derivation. In the next example, it is shown the first derivative of a sample using interpolation with different degrees.

```python
fd = fd[1]
fig = plt.figure()
fig.add_subplot(1, 1, 1)
for i in range(1, 4):
    fd.interpolator = SplineInterpolator(interpolation_order=i)
    fd.plot(fig=fig, derivative=1, label=f"Degree {i}")
fig.legend()
```

Out:

```
<matplotlib.legend.Legend object at 0x7f8a3110cda0>
```
FDataGrids can be differentiated using lagged differences with the method `derivative()`, creating another FDataGrid which could be interpolated in order to avoid interpolating before differentiating.

```python
fd_derivative = fd.derivative()

fig = fd_derivative.plot(label="Differentiation first")
fd_derivative.scatter(fig=fig)

fd.plot(fig=fig, derivative=1, label="Interpolation first")

fig.legend()
```
Sometimes our samples are required to be monotone, in these cases it is possible to use monotone cubic interpolation with the attribute `monotone`. A piecewise cubic hermite interpolating polynomial (PCHIP) will be used.

```python
fd_monotone = fd.copy(data_matrix=np.sort(fd.data_matrix, axis=1))

fig = fd_monotone.plot(linestyle='--', label="cubic")

fd_monotone.interpolator = SplineInterpolator(interpolation_order=3, monotone=True)

fd_monotone.plot(fig=fig, label="PCHIP")

fd_monotone.scatter(fig=fig, c='C1')
fig.legend()
```

2.16. Interpolation
All the interpolators will work regardless of the dimension of the image, but depending on the domain dimension some methods will not be available.

For the next examples it is constructed a surface, \( x_i : \mathbb{R}^2 \rightarrow \mathbb{R} \). By default, as in unidimensional samples, it is used linear interpolation.

```python
X, Y, Z = axes3d.get_test_data(1.2)
data_matrix = [Z.T]
sample_points = [X[:, 0], Y[:, 0]]

fd = skfda.FDataGrid(data_matrix, sample_points)

fig = fd.plot()
fd.scatter(fig=fig)
```
In the following figure it is shown the result of the cubic interpolation applied to the surface.

The degree of the interpolator polynomial does not have to coincide in both directions, for example, cubic interpolation in the first component and quadratic in the second one could be defined using a tuple with the values (3,2).

```python
fd.interpolator = SplineInterpolator(interpolation_order=3)

fig = fd.plot()
fd.scatter(fig=fig)
```
In case of surface derivatives could be taked in two directions, for this reason a tuple with the order of derivates in each direction could be passed. Let \( x(t, s) \) be the surface, in the following example it is shown the derivative with respect to the second coordinate, \( \frac{\partial}{\partial s} x(t, s) \).

```python
fd.plot(derivative=(0, 1))
```
The following table shows the interpolation methods available by the class `SplineInterpolator` depending on the domain dimension.

<table>
<thead>
<tr>
<th>Domain dimension</th>
<th>Linear</th>
<th>Up to degree 5</th>
<th>Monotone</th>
<th>Derivatives</th>
<th>Smoothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3 or more</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total running time of the script: (0 minutes 1.327 seconds)

Note: Click here to download the full example code

---

### 2.17 Magnitude-Shape Plot synthetic example

Shows the use of the MS-Plot applied to a synthetic dataset.
First, we generate a synthetic dataset following [DaWe18]

```python
random_state = np.random.RandomState(0)
n_samples = 200

fd = skfda.datasets.make_gaussian_process(
    n_samples=n_samples, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t,
    random_state=random_state)
```

We now add the outliers

```python
magnitude_outlier = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t + 20,
    random_state=random_state)

shape_outlier_shift = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t + 10 * (t > 0.4),
    random_state=random_state)

shape_outlier_peak = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t - 10 * ((0.25 < t) & (t < 0.3)),
    random_state=random_state)

shape_outlier_sin = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t + 2 * np.sin(18 * t),
    random_state=random_state)

shape_outlier_slope = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 10 * t,
    random_state=random_state)
```

(continues on next page)
magnitude_shape_outlier = skfda.datasets.make_gaussian_process(
    n_samples=1, n_features=100,
    cov=skfda.misc.covariances.Exponential(),
    mean=lambda t: 4 * t + 2 * np.sin(18 * t) - 20,
    random_state=random_state)

fd = fd.concatenate(magnitude_outlier, shape_outlier_shift,
                    shape_outlier_peak, shape_outlier_sin,
                    shape_outlier_slope, magnitude_shape_outlier)

The data is plotted to show the curves we are working with.

labels = [0] * n_samples + [1] * 6
fd.plot(group=labels,
        group_colors=['lightgrey', 'black'])

Out:

<Figure size 640x480 with 1 Axes>

The MS-Plot is generated. In order to show the results, the plot() method is used.
msplot = MagnitudeShapePlot(fdatagrid=fd)
msplot.plot()

Out:

<Figure size 640x480 with 1 Axes>

To show the utility of the plot, the curves are plotted showing each outlier in a different color

labels = [0] * n_samples + [1, 2, 3, 4, 5, 6]
colors = ['lightgrey', 'orange', 'blue', 'black',
         'green', 'brown', 'lightblue']

fd.plot(group=labels,
        group_colors=colors)
We now show the points in the MS-plot using the same colors

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(msplot.points[:, 0].ravel(), msplot.points[:, 1].ravel(),
           c=colors[0:1] * n_samples + colors[1:])
ax.set_title("MS-Plot")
ax.set_xlabel("magnitude outlyingness")
ax.set_ylabel("shape outlyingness")
```
Out:

```
Text(0, 0.5, 'shape outlyingness')
```

References

Total running time of the script: ( 0 minutes 1.402 seconds)

Note:  Click here to download the full example code

2.18 Radius neighbors classification

Shows the usage of the radius nearest neighbors classifier.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 2
```

(continues on next page)
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import numpy as np
import skfda
from skfda.misc.metrics import pairwise_distance, lp_distance
from skfda.ml.classification import RadiusNeighborsClassifier

In this example, we are going to show the usage of the radius nearest neighbors classifier in their functional version, a variation of the K-nearest neighbors classifier, where it is used a vote among neighbors within a given radius, instead of use the k nearest neighbors.

Firstly, we will construct a toy dataset to show the basic usage of the API. We will create two classes of sinusoidal samples, with different phases.

Make toy dataset

fd1 = skfda.datasets.make_sinusoidal_process(error_std=.0, phase_std=.35, random_state=0)
fd2 = skfda.datasets.make_sinusoidal_process(phase_mean=1.9, error_std=.0, random_state=1)

X = fd1.concatenate(fd2)
y = np.array([15 * [0] + 15 * [1]])

# Plot toy dataset
X.plot(group=y, group_colors=['C0', 'C1'])
As in the K-nearest neighbor example, we will split the dataset in two partitions, for training and test, using the sklearn function `train_test_split()`.

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, shuffle=True, stratify=y, random_state=0)
```

The label assigned to a test sample will be the majority class of its neighbors, in this case all the samples in the ball center in the sample.

If we use the $L^\infty$ metric, we can visualize a ball as a bandwidth with a fixed radius around a function. The following figure shows the ball centered in the first sample of the test partition.

```python
radius = 0.3
sample = X_test[0]  # Center of the ball
fig = X_train.plot(group=y_train, group_colors=['C0', 'C1'])
```

(continues on next page)
In this case, all the neighbors in the ball belong to the first class, so this will be the class predicted.

```python
# Creation of pairwise distance
l_inf = pairwise_distance(lp_distance, p=np.inf)
distances = l_inf(sample, X_train)[0]  # L_inf distances to 'sample'

# Plot samples in the ball
fig = X_train[distances <= radius].plot(color='C0')
sample.plot(fig=fig, color='red', linewidth=3)
fig.axes[0].fill_between(
    sample.sample_points[0], lower.data_matrix.flatten(),
    upper.data_matrix[0].flatten(), alpha=.25, color='C1')
```

Out:

<matplotlib.collections.PolyCollection object at 0x7f8a3110cda0>

2.18. Radius neighbors classification
We will fit the classifier `RadiusNeighborsClassifier`, which has a similar API than the sklearn estimator `RadiusNeighborsClassifier` but accepting `FDataGrid` instead of arrays with multivariate data.

The vote of the neighbors can be weighted using the parameter `weights`. In this case we will weight the vote inversely proportional to the distance.

```python
radius_nn = RadiusNeighborsClassifier(radius=radius, weights='distance')
radius_nn.fit(X_train, y_train)
```

Out:

```
RadiusNeighborsClassifier(algorithm='auto', leaf_size=30, metric='l2',
metric_params=None, multivariate_metric=False,
n_jobs=1, outlier_label=None, radius=0.3,
weights='distance')
```

We can predict labels for the test partition with `predict()`.

```python
pred = radius_nn.predict(X_test)
print(pred)
```

Out:
In this case, we get 100% accuracy, although it is a toy dataset.

```python
test_score = radius_nn.score(X_test, y_test)
print(test_score)
```

Out:

```
1.0
```

As in the K-nearest neighbor example, we can use the euclidean sklearn metric approximately equivalent to the functional $L^2$ one, but computationally faster.

We saw that $\|f - g\|_2 \approx \sqrt{\Delta h} \text{euclidean}(\vec{f}, \vec{g})$ if the samples are equiespaced (or almost).

In the KNN case, the constant $\sqrt{\Delta h}$ does not matter, but in this case will affect the value of the radius, dividing by $\sqrt{\Delta h}$.

In this dataset $\Delta h = 0.001$, so, we have to multiply the radius by $\frac{1}{\Delta h} = 10$ to achieve the same result.

The computation using this metric it is 1000 times faster. See the K-neighbors classifier example and the API documentation to get detailed information.

We obtain 100% accuracy with this metric too.

```python
radius_nn = RadiusNeighborsClassifier(radius=3, metric='euclidean',
                                       weights='distance', multivariate_metric=True)

radius_nn.fit(X_train, y_train)
test_score = radius_nn.score(X_test, y_test)
print(test_score)
```

Out:

```
1.0
```

If the radius is too small, it is possible to get samples with no neighbors. The classifier will raise an exception in this case.

```python
radius_nn.set_params(radius=.5)  # Radius 0.05 in the L2 distance
radius_nn.fit(X_train, y_train)

try:
    radius_nn.predict(X_test)
except ValueError as e:
    print(e)
```

Out:

```
No neighbors found for test samples array([[1, 2, 4, 7]]), you can try using larger radius, giving a label for outliers, or considering removing them from your dataset.
```

A label to these outlier samples can be provided to avoid this problem.
radius_nn.set_params(outlier_label=2)
radius_nn.fit(X_train, y_train)
pred = radius_nn.predict(X_test)

print(pred)

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/fda/envs/latest/lib/python3.6/site-packages/sklearn/neighbors/_classification.py:566: UserWarning: Outlier label 2 is not in training classes. All class probabilities of outliers will be assigned with 0.
  
[0 2 2 1 2 1 1 2]
```

This classifier can be used with multivariate functional data, as surfaces or curves in $\mathbb{R}^N$, if the metric support it too.

Total running time of the script: ( 0 minutes 0.584 seconds)

Note: Click here to download the full example code

## 2.19 Clustering

In this example, the use of the clustering plot methods is shown applied to the Canadian Weather dataset. K-Means and Fuzzy K-Means algorithms are employed to calculate the results plotted.

```python
# Author: Amanda Hernando Bernabé
# License: MIT
# sphinx_gallery_thumbnail_number = 6

import matplotlib.pyplot as plt
import numpy as np
from skfda import datasets
from skfda.exploratory.visualization.clustering import (plot_clusters, plot_cluster_lines, plot_cluster_bars)
from skfda.ml.clustering import KMeans, FuzzyCMeans

First, the Canadian Weather dataset is downloaded from the package ‘fda’ in CRAN. It contains a FDataGrid with daily temperatures and precipitations, that is, it has a 2-dimensional image. We are interested only in the daily average temperatures, so we select the first coordinate function.

dataset = datasets.fetch_weather()
fd = dataset['data'][0]
fd_temperatures = fd.coordinates[0]

# The desired FDataGrid only contains 10 random samples, so that the example
# provides clearer plots.
indices_samples = np.array([1, 3, 5, 10, 14, 17, 21, 25, 27, 30])
fd = fd_temperatures[indices_samples]
```
The data is plotted to show the curves we are working with. They are divided according to the target. In this case, it includes the different climates to which the weather stations belong to.

```python
climate_by_sample = [dataset["target"]][i] for i in indices_samples

# Note that the samples chosen belong to three of the four possible target groups. By coincidence, these three groups correspond to indices 1, 2, 3, that is why the indices (climate_by_sample’) are decremented in 1. In case # of reproducing the example with other ‘indices_samples’ and the four groups # are not present in the sample, changes should be made in order ‘indexer’ # contains numbers in the interval [0, n_target_groups) and at least, an # occurrence of each one.
indexer = np.asarray(climate_by_sample) - 1

indices_target_groups = np.unique(climate_by_sample)
climates = dataset["target_names"][:indices_target_groups]

# Assigning the color to each of the groups.
colormap = plt.cm.get_cmap(‘tab20b’)
n_climates = len(climates)
climate_colors = colormap(np.arange(n_climates) / (n_climates - 1))

fd.plot(group=indexer, group_colors=climate_colors,
        group_names=climates)
```

![Canadian Weather](image)

2.19. Clustering
The number of clusters is set with the number of climates, in order to see the performance of the clustering methods, and the seed is set to one in order to obtain always the same result for the example.

```python
def kmeans:
    kmeans = KMeans(n_clusters=n_clusters, random_state=seed)
    kmeans.fit(fd)
    print(kmeans.predict(fd))
```

To see the information in a graphic way, the method `plot_clusters()` can be used.

```python
# Customization of cluster colors and labels in order to match the first image of raw data.
cluster_colors = climate_colors[np.array([0, 2, 1])]
cluster_labels = climates[np.array([0, 2, 1])]
plot_clusters(kmeans, fd, cluster_colors=cluster_colors,
             cluster_labels=cluster_labels)
```
Other clustering algorithm implemented is the Fuzzy K-Means found in the class `FuzzyCMeans`. Following the above procedure, an object of this type is instantiated with the desired data and then, the `fit()` method is called. Internally, the attribute `labels_` is calculated, which contains `n_clusters` elements for each sample and dimension, denoting the degree of membership of each sample to each cluster. They are obtained calling the method `predict()`. Also, the centroids of each cluster are obtained.

```python
fuzzy_kmeans = FuzzyCMeans(n_clusters=n_clusters, random_state=seed)
fuzzy_kmeans.fit(fd)
print(fuzzy_kmeans.predict(fd))
```

Out:

```
[[0.8721254  0.11189295  0.01598165]
 [0.4615364  0.51285956  0.02560405]
 [0.97428363 0.01882257  0.0068938]
 [0.91184323 0.05369029  0.03446648]
 [0.79072268 0.18411219  0.02516513]
 [0.178624   0.05881132  0.76256468]
 [0.01099498  0.00492593  0.98407909]
 [0.03166897  0.96349997  0.00493106]]
```
To see the information in a graphic way, the method `plot_clusters()` can be used. It assigns each sample to the cluster whose membership value is the greatest.

```python
plot_clusters(fuzzy_kmeans, fd, cluster_colors=cluster_colors,
             cluster_labels=cluster_labels)
```

Out:

![Canadian Weather](image)

Another plot implemented to show the results in the class `FuzzyCMeans` is `plot_cluster_lines()` which is similar to parallel coordinates. It is recommended to assign colors to each of the samples in order to identify them. In this example, the colors are the ones of the first plot, dividing the samples by climate.

```python
colors_by_climate = colormap(indexer / (n_climates - 1))

plot_cluster_lines(fuzzy_kmeans, fd, cluster_labels=cluster_labels,
                   sample_colors=colors_by_climate)
```
Finally, the function `plot_cluster_bars()` returns a barplot. Each sample is designated with a bar which is filled proportionally to the membership values with the color of each cluster.

```python
plot_cluster_bars(fuzzy_kmeans, fd, cluster_colors=cluster_colors,
                 cluster_labels=cluster_labels)
```
The possibility of sorting the bars according to a cluster is given specifying the number of cluster, which belongs to the interval \([0, n\_clusters)\).

We can order the data using the first cluster:

```python
plot_cluster_bars(fuzzy_kmeans, fd, sort=0, cluster_colors=cluster_colors,
                 cluster_labels=cluster_labels)
```
Out:

```python
<Figure size 640x480 with 1 Axes>
```

Using the second cluster:

```python
plot_cluster_bars(fuzzy_kmeans, fd, sort=1, cluster_colors=cluster_colors,
                 cluster_labels=cluster_labels)
```
And using the third cluster:

```
plot_cluster_bars(fuzzy_kmeans, fd, sort=2, cluster_colors=cluster_colors,
                 cluster_labels=cluster_labels)
```
2.20 Pairwise alignment

Shows the usage of the elastic registration to perform a pairwise alignment.

```
import matplotlib.colors as clr
import matplotlib.pyplot as plt
import numpy as np
```
Given any two functions $f$ and $g$, we define their pairwise alignment or registration to be the problem of finding a warping function $\gamma^*$ such that a certain energy term $E[f, g \circ \gamma]$ is minimized.

$$
\gamma^* = \arg\min_{\gamma} E[f \circ \gamma, g]
$$

In the case of elastic registration it is taken as energy function the Fisher-Rao distance with a penalisation term, due to the property of invariance to reparameterizations of warpings functions.

$$
E[f \circ \gamma, g] = d_{FR}(f \circ \gamma, g)
$$

Firstly, we will create two unimodal samples, $f$ and $g$, defined in $[0, 1]$ which will be used to show the elastic registration. Due to the similarity of these curves can be aligned almost perfectly between them.

```python
# Samples with modes in 1/3 and 2/3
fd = make_multimodal_samples(n_samples=2, modes_location=[1 / 3, 2 / 3],
                             random_state=1, start=0, mode_std=.01)

fig = fd.plot()
fig.axes[0].legend(['$f$', '$g$'])
```
In this example $g$ will be used as template and $f$ will be aligned to it. In the following figure it is shown the result of the registration process, which can be computed using `ElasticRegistration`.

```python
def fd = [f, g]
elastic_registration = ElasticRegistration(template=g)

# Aligns f to g
f_align = elastic_registration.fit_transform(f)

fig = fd.plot()
f_align.plot(fig=fig, color='C0', linestyle='--')

# Legend
fig.axes[0].legend(['$f$', '$g$', '$f \circ \gamma$'])
```

Out:

```
<matplotlib.legend.Legend object at 0x7f8a313cc5f8>
```
The non-linear transformation $\gamma$ applied to $f$ in the alignment is stored in the attribute `warping_`.

```python
# Warping used in the last transformation
warping = elastic_registration.warping_
fig = warping.plot()

# Plot identity
t = np.linspace(0, 1)
fig.axes[0].plot(t, t, linestyle='--')

# Legend
fig.axes[0].legend(['$\gamma$', '$\gamma_{id}$'])

fig
```

The transformation necessary to align $g$ to $f$ will be the inverse of the original warping function, $\gamma^{-1}$. This fact is a consequence of the use of the Fisher-Rao metric as energy function.
warping_inverse = invert_warping(warping)

fig = fd.plot(label='$f$')
g.compose(warping_inverse).plot(fig=fig, color='C1', linestyle='--')

# Legend
fig.axes[0].legend(['$f$', '$g$', '$g \circ \gamma^{-1}$'])

Out:
<matplotlib.legend.Legend object at 0x7f8a3108f438>

The amount of deformation used in the registration can be controlled by using a variation of the metric with a penalty term $\lambda R(\gamma)$ which will reduce the elasticity of the metric.

The following figure shows the original curves and the result to the alignment varying $\lambda$ from 0 to 0.2.

# Values of lambda
penalties = np.linspace(0, .2, 20)

# Creation of a color gradient
cmap = clr.LinearSegmentedColormap.from_list('custom cmap', ['C1', 'C0'])
color = cmap(.2 + 3 * penalties)

(continues on next page)
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

for penalty, c in zip(penalties, color):
    elastic_registration.set_params(penalty=penalty)
    elastic_registration.transform(f).plot(fig, color=c)

f.plot(fig=fig, color='C0', linewidth=2., label='$f$')
g.plot(fig=fig, color='C1', linewidth=2., label='$g$')

# Legend
fig.axes[0].legend()

Out:
<matplotlib.legend.Legend object at 0x7f8a317c1e48>

This phenomenon of loss of elasticity is clearly observed in the warpings used, since as the term of penalty increases, the functions are closer to $\gamma_{id}$.
```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

for penalty, c in zip(penalties, color):
    elastic_registration.set_params(penalty=penalty)
    elastic_registration.transform(f)
    elastic_registration.warping_.plot(fig, color=c)

# Plots identity
fig.axes[0].plot(t, t, color='C0', linestyle='--')
```

Out:
```
[<matplotlib.lines.Line2D object at 0x7f8a315e8a90>]
```

We can perform the pairwise of multiple curves at once. We can use a single curve as template to align a set of samples to it or a set of templates to make the alignment the two sets.

In the elastic registration example it is shown the alignment of multiple curves to the same template.

We will build two sets with 3 curves each, \{f_i\} and \{g_i\}.

```python
# Creation of the 2 sets of functions
state = np.random.RandomState(0)
```

(continues on next page)
location1 = state.normal(loc=-.3, scale=.1, size=3)
f = skfda.datasets.make_multimodal_samples(
    n_samples=3, modes_location=location1, noise=.001, random_state=1)

location2 = state.normal(loc=.3, scale=.1, size=3)
g = skfda.datasets.make_multimodal_samples(
    n_samples=3, modes_location=location2, random_state=2)

# Plot of the sets
fig = f.plot(color="C0", label="f_i")
g.plot(fig=fig, color="C1", label="g_i")

labels = fig.axes[0].get_lines()
fig.axes[0].legend(handles=[labels[0], labels[-1]])
# Registration of the sets
elastic_registration = ElasticRegistration(template=g)

fd_registered = elastic_registration.fit_transform(fd)

# Plot of the curves
fig = fd.plot(color="C0", label="f_i")
l1 = fig.axes[0].get_lines()[-1]
g.plot(fig=fig, color="C1", label="g_i")
l2 = fig.axes[0].get_lines()[-1]
fd_registered.plot(fig=fig, color="C0", linestyle="--", label="f_i circ gamma_i")
l3 = fig.axes[0].get_lines()[-1]

fig.axes[0].legend(handles=[l1, l2, l3])

Out:

<matplotlib.legend.Legend object at 0x7f8a3132aeb8>


2.20. Pairwise alignment 325
2.21 Extrapolation

Shows the usage of the different types of extrapolation.

```python
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 2
import mpl_toolkits.mplot3d
import matplotlib.pyplot as plt
import numpy as np
import skfda

The extrapolation defines how to evaluate points that are outside the domain range of a FDataBasis or a FDataGrid.

The FDataBasis objects have a predefined extrapolation which is applied in evaluate if the argument extrapolation is not supplied. This default value could be specified when the object is created or changing the attribute extrapolation.

The extrapolation could be specified by a string with the short name of an extrapolator or with an :class:`~skfda.representation.extrapolation.Extrapolator`.

To show how it works we will create a dataset with two unidimensional curves defined in (0,1), and we will represent it using a grid and different types of basis.

```python
dgrid = skfda.datasets.make_sinusoidal_process(
    n_samples=2, error_std=0, random_state=0)
dgrid.dataset_label = "Grid"

fd_fourier = fdgrid.to_basis(skfda.representation.basis.Fourier())
fd_fourier.dataset_label = "Fourier Basis"

fd_monomial = fdgrid.to_basis(skfda.representation.basis.Monomial(n_basis=5))
fd_monomial.dataset_label = "Monomial Basis"

fd_bspline = fdgrid.to_basis(skfda.representation.basis.BSpline(n_basis=5))
fd_bspline.dataset_label = "BSpline Basis"

# Plot of different representations
fig, ax = plt.subplots(2, 2)
dgrid.plot(ax[0][0])
fd_fourier.plot(ax[0][1])
fd_monomial.plot(ax[1][0])
fd_bspline.plot(ax[1][1])
```

(continues on next page)
If the extrapolation is not specified when a list of points is evaluated and the default extrapolation of the objects has not been specified it is used the type “none”, which will evaluate the points outside the domain without any kind of control.

For this reason the behavior outside the domain will change depending on the representation, obtaining a periodic behavior in the case of the Fourier basis and polynomial behaviors in the rest of the cases.
fd_monomial.plot(ax[1][0], domain_range=domain_extended, linestyle='--')
fd_bspline.plot(ax[1][1], domain_range=domain_extended, linestyle='--')

# Plot configuration
for axes in fig.axes:
    axes.set_prop_cycle(None)
    axes.set_ylim((-1.5, 1.5))
    axes.set_xlim((-0.25, 1.25))

# Disable xticks of first row
ax[0][0].set_xticks([])
ax[0][1].set_xticks([])

# Plot objects in the domain range
fdgrid.plot(ax[0][0])
fd_fourier.plot(ax[0][1])
fd_monomial.plot(ax[1][0])
fd_bspline.plot(ax[1][1])

BSpline Basis

Out:

<Figure size 640x480 with 4 Axes>
Periodic extrapolation will extend the domain range periodically. The following example shows the periodical extension of an FDataGrid.

It should be noted that the Fourier basis is periodic in itself, but the period does not have to coincide with the domain range, obtaining different results applying or not extrapolation in case of not coinciding.

```python
import numpy as np
import matplotlib.pyplot as plt

# Initial domain
domain = (-0.5, 1.5)
domain_extended = 10

# Create data
t = np.linspace(*domain)
fdgrid = FDataGrid()
fdgrid.dataset_label = 'Periodic extrapolation'

# Evaluation of the grid
# Extrapolation supplied in the evaluation
values = fdgrid(t, extrapolation='periodic')

# Plotting
plt.plot(t, values, linestyle='--
plt.gca().set_prop_cycle(]
fig = plt.figure()
fdgrid.plot(fig)

Out: [Figure size 640x480 with 1 Axes]
```

**Periodic extrapolation**

![](image)

Out:

<Figure size 640x480 with 1 Axes>
Another possible extrapolation, "bounds", will use the values of the interval bounds for points outside the domain range.

```python
fig = plt.figure()
fdgrid.dataset_label = "Boundary extrapolation"

# Other way to call the extrapolation, changing the default value
fdgrid.extrapolation = "bounds"

# Evaluation of the grid
values = fdgrid(t)
plt.plot(t, values.T, linestyle='--')
plt.gca().set_prop_cycle(None)  # Reset color cycle
fdgrid.plot(fig=fig)  # Plot dataset
```

The **FillExtrapolation** will fill the points extrapolated with the same value. The case of filling with zeros could be specified with the string "zeros", which is equivalent to `extrapolation=FillExtrapolation(0)`. 

Out:

```
<Figure size 640x480 with 1 Axes>
```
```
fdgrid.dataset_label = "Fill with zeros"

# Evaluation of the grid filling with zeros
fdgrid.extrapolation = "zeros"

# Plot in domain extended
fig = fdgrid.plot(domain_range=domain_extended, linestyle='--')
plt.gca().set_prop_cycle(None) # Reset color cycle
fdgrid.plot(fig=fig) # Plot dataset
```

```
Fill with zeros
```

Out:

```
<Figure size 640x480 with 1 Axes>
```

The string "nan" is equivalent to FillExtrapolation(np.nan).

```
values = fdgrid([-1, 0, 0.5, 1, 2], extrapolation="nan")
print(values)
```

Out:

```
```
It is possible to configure the extrapolation to raise an exception in case of evaluating a point outside the domain.

```python
try:
    res = fd_fourier(t, extrapolation="exception")
except ValueError as e:
    print(e)
```

Out:

```
Attempt to evaluate 15 points outside the domain range.
```

All the extrapolators shown will work with multidimensional objects. In the following example it is constructed a 2d-surface and it is extended using periodic extrapolation.

```python
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Make data.
t = np.arange(-2.5, 2.75, 0.25)
X, Y = np.meshgrid(t, t)
Z = np.exp(-0.5 * (X**2 + Y**2))

# Creation of FDataGrid
fd_surface = skfda.FDataGrid([Z], (t, t))

# Evaluation with periodic extrapolation
values = fd_surface((t, t), grid=True, extrapolation="periodic")
T, S = np.meshgrid(t, t)

ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```
The previous extension can be compared with the extrapolation using the values of the bounds.

```python
values = fd_surface((t, t), grid=True, extrapolation="bounds")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```
Or filling the surface with zeros outside the domain.

```python
code
values = fd_surface((t, t), grid=True, extrapolation="zeros")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```
Out:

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection object at 0x7f8a3124aba8>
```

**Total running time of the script:** ( 0 minutes 1.283 seconds)

An exhaustive list of all the contents of the package can be found in the genindex.
Currently, scikit-fda is available in Python 3.6 and 3.7, regardless of the platform. The stable version can be installed via PyPI:

```
$ pip install scikit-fda
```

It is possible to install the latest version of the package, available in the develop branch, by cloning this repository and doing a manual installation.

```
$ git clone https://github.com/GAA-UAM/scikit-fda.git
$ pip install ./scikit-fda
```

In this type of installation make sure that your default Python version is currently supported, or change the python and pip commands by specifying a version, such as python3.6.
All contributions are welcome. You can help this project grow in multiple ways, from creating an issue, reporting an improvement or a bug, to doing a repository fork and creating a pull request to the development branch. The people involved at some point in the development of the package can be found in the contributors file.
The package is licensed under the BSD 3-Clause License. A copy of the license can be found along with the code or in the project page.


Symbols

__init__() (skfda.exploratory.outliers.Direction- 
alOutlierDetector method), 138
__init__() (skfda.exploratory.outliers.IQROut- 
lierDetector method), 135
__init__() (skfda.exploratory.visualization.Boxplot 
method), 118
__init__() (skfda.exploratory.visualization.Magni-
tudeShapePlot method), 125
__init__() (skfda.ml.classification.KNeighborsClas-
sifier method), 160
__init__() (skfda.ml.classification.NearestCentroid 
method), 169
__init__() (skfda.ml.classification.RadiusNeigh-
borsClassifier method), 165
__init__() (skfda.ml.clustering.FuzzyCMeans 
method), 178
__init__() (skfda.ml.clustering.KMeans method), 173
__init__() (skfda.ml.clustering.NearestNeighbors 
method), 182
__init__() (skfda.ml.regression.KNeighborsRegress-
or method), 191
__init__() (skfda.ml.regression.LinearScalarRegres-
sion method), 187
__init__() (skfda.ml.regression.RadiusNeigh-
borsRegressor method), 196
__init__() (skfda.preprocessing.registration.Elasti-
cRegistration method), 98
__init__() (skfda.preprocessing.registration.ShiftRegister-
ation method), 90
__init__() (skfda.preprocessing.registration.elas-
tic.SRSF method), 104
__init__() (skfda.preprocessing.registration.valida-
tion.AmplitudePhaseDecomposition 
method), 108
__init__() (skfda.preprocessing.registration.validation. 
LeastSquares method), 110
__init__() (skfda.preprocessing.registration.validation.PairwiseCorrelation 
method), 113
__init__() (skfda.preprocessing.registration.validation.SobolevLeastSquares 
method), 111
__init__() (skfda.preprocessing.smoothing.BasisS-
moother method), 78
__init__() (skfda.preprocessing.smoothing.kernel_smothers.KNeighborsSmoothen 
method), 73
__init__() (skfda.preprocessing.smoothing.kernel_smothers.NadarayaWatsonSmoothen 
method), 66
__init__() (skfda.preprocessing.smoothing.validation.LinearSmootherGeneralizedCVScorer 
method), 80
__init__() (skfda.preprocessing.smoothing.validation.LinearSmootherLeaveOneOutScorer 
method), 79
__init__() (skfda.preprocessing.smoothing.validation.SmoothingParameterSearch 
method), 85
__init__() (skfda.representation.FData method), 48
__init__() (skfda.representation.basis.BSpline 
method), 35
__init__() (skfda.representation.basis.Constant 
method), 45
__init__() (skfda.representation.basis.FDataBasis 
method), 22
__init__() (skfda.representation.basis.Fourier 
method), 39
__init__() (skfda.representation.basis.Monomial 
method), 42
__init__() (skfda.representation.evaluator.Evalua-
tor method), 62
__init__() (skfda.representation.evaluator.Evalua-
torConstructor method), 62
Index
evaluate() (skfda.representation.basis.BSpline method), 36
evaluate() (skfda.representation.basis.Constant method), 45
evaluate() (skfda.representation.basis.FDataBasis method), 24
evaluate() (skfda.representation.basis.Fourier method), 39
evaluate() (skfda.representation.basis.Monomial method), 43
evaluate() (skfda.representation.evaluator.Evaluator method), 62
evaluate() (skfda.representation.FData method), 51
evaluate() (skfda.representation.grid.FDataGrid method), 51
evaluate_composed() (skfda.representation.evaluator.Evaluator method), 63
Evaluator (class in skfda.representation.evaluator), 62
evaluator() (skfda.representation.evaluator.EvaluatorConstructor method), 62
evaluator() (skfda.representation.extrapolation.extrapolation.BoundaryExtrapolation method), 58
evaluator() (skfda.representation.extrapolation.extrapolation.ExceptionExtrapolation method), 59
evaluator() (skfda.representation.extrapolation.extrapolation.FillExtrapolation method), 60
evaluator() (skfda.representation.extrapolation.extrapolation.PeriodicExtrapolation method), 61
EvaluatorConstructor (class in skfda.representation.evaluator), 61
ExceptionExtrapolation (class in skfda.representation.extrapolation), 58
extrapolation (skfda.representation.basis.FDataBasis attribute), 25
extrapolation (skfda.representation.FData attribute), 48, 51
extrapolation (skfda.representation.grid.FDataGrid attribute), 4, 11
extrapolator_evaluator (skfda.representation.basis.FDataBasis attribute), 25
extrapolator_evaluator (skfda.representation.FData attribute), 51
extrapolator_evaluator (skfda.representation.grid.FDataGrid attribute), 11
factorize() (skfda.representation.basis.FDataBasis method), 25
factorize() (skfda.representation.grid.FDataGrid method), 51
FData (class in skfda.representation), 47
FDataBasis (class in skfda.representation.basis), 20
FDataGrid (class in skfda.representation.grid), 3
datagrid (skfda.exploratory.visualization.Boxplot attribute), 116
datagrid (skfda.exploratory.visualization.MagnitudeShapePlot attribute), 123
datagrid (skfda.exploratory.visualization.SurfaceBoxplot attribute), 144
fetch_aemet() (in module skfda.datasets), 144
fetch_cran() (in module skfda.datasets), 145
fetch_growth() (in module skfda.datasets), 141
fetch_medflies() (in module skfda.datasets), 143
fetch_octane() (in module skfda.datasets), 145
fetch_phoneme() (in module skfda.datasets), 142
fetch_tecator() (in module skfda.datasets), 142
fetch_ucr() (in module skfda.datasets), 145
fetch_weather() (in module skfda.datasets), 144
fill_value (skfda.representation.extrapolation.FillExtrapolation method), 60
FillExtrapolation (class in skfda.representation.extrapolation), 59
fillna() (skfda.representation.basis.FDataBasis method), 25
fillna() (skfda.representation.FData method), 52
fillna() (skfda.representation.grid.FDataGrid method), 12
finite_prediction_error() (in module skfda.preprocessing.smoothing.validation), 81
fisher_rao_distance() (in module skfda.misc.metrics), 153
fit() (skfda.ml.classification.KNeighborsClassifier method), 160
fit() (skfda.ml.classification.KNeighborsRegressor method), 174
fit() (skfda.ml.cluster.FuzzyCMeans method), 178
fit() (skfda.ml.cluster.KMeans method), 174
fit() (skfda.ml.cluster.NearestCentroid method), 169
fit() (skfda.ml.cluster.NearestNeighborsClassifier method), 165
fit() (skfda.ml.cluster.NearestNeighborsRegressor method), 182
fit() (skfda.ml.regression.KNeighborsRegressor method), 191
fit() (skfda.ml.regression.RadiusNeighborsRegressor method), 196
fit() (skfda.preprocessing-registration.elastic.SRSF method), 104
fit() (skfda.preprocessing-registration.ElasticRegistration method), 98
Index 351
inertia_

inertia_

initial_value

inknots

inner_product()

interpolation_order

interpolator

inverse_transform()

inverse_transform()

inverse_transform()

inverse_transform()

inverse_transform()

inverse_transform()

inverse_warping()

IQROutlierDetector

isna()

isna()

isna()

K

keepdims

keepdims

KMeans

kneighbors()

kneighbors()

kneighbors()

kneighbors()

kneighbors()

kneighbors_graph()

kneighbors_graph()

kneighbors_graph()

KNeighborsClassifier

KNeighborsRegressor

KNeighborsSmother

landmark_registration()

landmark_registration_warping()

landmark_shift()

landmark_shift_deltas()

LeastSquares

LinearScalarRegression

LinearSmotherGeneralizedCVScorer

LinearSmotherLeaveOneOutScorer

LocalLinearRegressionSmother

lp_distance()

MagnitudeShapePlot

make_gaussian_process()

make_multimodal_landmarks()

make_multimodal_samples()

make_random_warping()

make_sinusoidal_process()

mean()

mean()

mean()

median
median (skfda.exploratory.visualization.SurfaceBoxplot attribute), 120
mediancol (skfda.exploratory.visualization.Boxplot attribute), 116
modified_band_depth() (in module skfda.exploratory.depth), 131
Monomial (class in skfda.representation.basis), 41
monotone (skfda.representation.interpolation.SplineInterpolator attribute), 19
n_basis (skfda.representation.basis.BSpline attribute), 34
n_samples (skfda.representation.FData attribute), 47, 52
NadarayaWatsonSmoother (class in skfda.preprocessing.smoothing.kernel_smoothers), 64
nbytes (skfda.representation.basis.FDataBasis attribute), 28
nbytes (skfda.representation.grid.FDataGrid attribute), 12
PairwiseCorrelation (class in skfda.preprocessing.registration.validation), 111
penalty() (skfda.representation.basis.BSpline method), 36
period (skfda.representation.basis.Fourier attribute), 38
PeriodicExtrapolation (class in skfda.representation.extrapolation), 60
phase_distance() (in module skfda.misc.metrics), 154
plot() (skfda.exploratory.visualization.Boxplot method), 119
plot() (skfda.exploratory.visualization.MagnitudeShapePlot method), 126
plot() (skfda.exploratory.visualization.SurfaceBoxplot method), 122
plot() (skfda.representation.basis.BSpline method), 37
plot() (skfda.representation.basis.Constant method), 47
plot() (skfda.representation.basis.FDataBasis method), 28
plot() (skfda.representation.basis.Fourier method), 40
plot() (skfda.representation.basis.Fourier method), 44
plot() (skfda.representation.basis.Monomial method), 44
plot() (skfda.representation.FData method), 53
plot() (skfda.representation.grid.FDataGrid method), 13

O
order (skfda.representation.basis.BSpline attribute), 34
outcol (skfda.exploratory.visualization.SurfaceBoxplot attribute), 120
outliercol (skfda.exploratory.visualization.Boxplot attribute), 116
outliercol (skfda.exploratory.visualization.MagnitudeShapePlot attribute), 124
outliers (skfda.exploratory.visualization.Boxplot attribute), 116
outliers (skfda.exploratory.visualization.MagnitudeShapePlot attribute), 123
outlyingness_to_depth() (in module skfda.exploratory.depth), 133

P
pairwise_distance() (in module skfda.misc.metrics), 157
pairwise_distance() (in module skfda.misc.metrics), 157
PairwiseCorrelation (class in skfda.preprocessing.registration.validation), 111
penalty() (skfda.representation.basis.BSpline method), 36
penalty() (skfda.representation.basis.Constant method), 46
penalty() (skfda.representation.basis.Fourier method), 40
penalty() (skfda.representation.basis.Monomial method), 43
period (skfda.representation.basis.Fourier attribute), 38
PeriodicExtrapolation (class in skfda.representation.extrapolation), 60
phase_distance() (in module skfda.misc.metrics), 154
plot() (skfda.exploratory.visualization.Boxplot method), 119
plot() (skfda.exploratory.visualization.MagnitudeShapePlot method), 126
plot() (skfda.exploratory.visualization.SurfaceBoxplot method), 122
plot() (skfda.representation.basis.BSpline method), 37
plot() (skfda.representation.basis.Constant method), 47
plot() (skfda.representation.basis.FDataBasis method), 28
plot() (skfda.representation.basis.Fourier method), 40
plot() (skfda.representation.basis.Monomial method), 44
plot() (skfda.representation.FData method), 53
plot() (skfda.representation.grid.FDataGrid method), 13

Index 353