UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Double Degree in Computer Science and Mathematics

DEGREE WORK

Development of a Python package for Functional Data Analysis

Depth Measures, Applications and Clustering

Author: Amanda Hernando Bernabé
Advisor: José Luis Torrecilla

junio 2019

Amanda Hernando Bernabé
Development of a Python package for Functional Data Analysis

Amanda Hernando Bernabé
C\ Francisco Tomás y Valiente N$^{\underline{o}}$ 11

PRINTED IN SPAIN

A mi familia y a mis amigos

Striving for success without hard work is like
trying to harvest where you haven't planted.

David Bly

Agradecimientos

Me gustaría dar las gracias a todo el equipo que forma parte de este proyecto. A Alberto Suárez por proponerlo y darme la oportunidad de participar en una iniciativa con aplicación en el mundo real a corto plazo. A José Luis Torrecilla por introducirme en la parte más teórica de los datos funcionales y a Carlos Ramos por el gran esfuerzo hecho enseñándome Python. Además agradezco a mis compañeros Pablo Pérez y Pablo Marcos la facilidad de trabajar con ellos y la motivación puesta en el proyecto. Por último, agradecer a Eloy Anguiano la plantilla de este artículo que ha hecho la escritura mucho menos tediosa.

Resumen

En este trabajo, se aborda el problema del Análisis de Datos Funcionales (FDA). Cada observación en datos funcionales es una función que varía sobre un continuo. Este tipo de datos complejos se está volviendo cada vez más común en muchos campos de la investigación. Sin embargo, el Análisis de Datos Funcionales es un campo relativamente reciente en el que las implementaciones de software se limitan básicamente a R. Además, aunque sigan un esquema *open-source*, la contibución a las mismas puede resultar dificultosa. El objetivo final de este proyecto es proporcionar un paquete completo, *scikit-fda*, para el Análisis de Datos Funcionales escrito en Python.

En este trabajo de fin de grado, la funcionalidad implementada en el paquete incluye las medidas de profundidad funcional junto con sus aplicaciones y nociones elementales de clustering. En los espacios funcionales, establecer un orden es complicado debido a su naturaleza. Las medidas de profundidad permiten definir estadísticos robustos para los datos funcionales. En el paquete se pueden encontrar unas de las más habituales, la medida de profundidad de Fraiman y Muñiz, la *band depth* o una modificación de esta última, la *modified band depth*. Las medidas de profundidad se utilizan en la construcción de herramientas gráficas, tanto el diagrama de caja funcional como el *magnitude-shape plot* se introducen en el paquete además de sus procedimientos de detección de valores atípicos. Asimismo, se realizan contribuciones en el área del aprendizaje automático en el cual se añaden algoritmos básicos de clustering al paquete: K-means y Fuzzy K-means. Finalmente, se muestran los resultados de la aplicación de estos métodos al conjunto de datos del clima canadiense.

El paquete Python está publicado en un repositorio de GitHub. Es *open-source* con el objetivo de crecer y mantenerse actualizado. A largo plazo, se espera que cubra las técnicas fundamentales del FDA y se convierta en una *toolbox* ampliamente utilizada para la investigación en el FDA.

Palabras clave

Análisis de Datos Funcionales, Medidas de Profundidad, Diagrama de Caja, Detección de datos atípicos, Clustering, Python, Software

Abstract

In this paper, the problem of analyzing functional data is addressed. Each observation in functional data is a function that varies over a continuum. This kind of complex data is increasingly becoming more common in many research fields. However, Functional Data Analysis (FDA) is a relatively recent field in which software implementations are basically limited to R. In addition, although they may follow an open-source scheme, the contribution to them may turn out to be complicated. The final goal of this project is to provide a comprehensive Python package for Functional Data Analysis, *scikit-fda*.

In this undergraduate thesis, the functionality implemented in the package includes functional depth measures together with their applications and elementary notions of clustering. In a functional space, establishing an order is complicated due to its nature. Depth measures allow to define robust statistics for functional data. In the package you can find some of the most common, Fraiman and Muniz depth measure, the band depth measure or a modification of the latter, the modified band depth. Depth measures are used in the construction of graphic tools, both the functional boxplot and the magnitude-shape plot are introduced in the package along with their outlier detection procedures. Furthermore, contributions in the area of machine learning are made in which basic clustering algorithms are added to the package: K-means and Fuzzy K-means. Finally, the results of applying these methods to the Canadian Weather dataset are shown.

The Python package is published in a GitHub repository. It is open-source wth the aim of growing and being kept up to date. In the long term it is expected to cover the fundamental techniques in FDA and become a widely-used toolbox for research in FDA.

Keywords

Functional Data Analysis, Depth Measures, Boxplot, Outlier detection, Clustering, Python, Software

Table of Contents

Lists

List of algorithms

List of equations

List of figures

1 Introduction

In recent years, Functional Data Analysis (FDA) has become one of the most active domains in Statistics. The objects under study are real functions which are assumed to be realizations of stochastic processes that can represent curves, surfaces or anything else varying over a continuum.

Due to the advances in technology, such functional data can be collected in many scientific areas including but not limited to biology, finance, engineering, medicine and meteorology. As a result, FDA has engaged an increasing number of researchers during the past decades. Many methods have been proposed to extract useful information from functional data. The main references in this field are Ramsay and Silverman (2005) [1], and Ferraty and Vieu (2006) [2].

Nevertheless, software implementations are restricted fundamentally to R programming language. The available packages include some general purpose ones, such as *fda* [3] or *fda.usc* [4] and others more specific, among which the *refund* [5], *roahd* [6] or *rainbow* [7] packages can be found. They implement functionality regarding regression, robust statistics and visualization techniques respectively. All of them can be found in the The Comprehensive R Archive Network (CRAN) repository. More rare to encounter, there are also implementations written in Matlab. They include also the *fda* [3] package or the PACE [8] package, the latter developed by the Department of Statistics at the University of California.

As a consequence, the implementation of a Python package for FDA was considered to be a valuable tool for the increasing number of researchers who are adopting this language. In addition, dealing with an open-source software in which continuous collaboration is possible promotes an up-to-date tool.

1.1. Goals and Scope

The main purpose of this project is to expand the functionality of *scikit-fda*, the Python FDA package started last year by the former student Miguel Carbajo [9]. The initial version of the package contained some basic tools to work with functional data. The functionality implemented was principally related with the representation of the objects studied: functions.

The functions are commonly assumed to belong to a Hilbert space and to be able to be represented with a convenient functional basis, such as B-Splines or Fourier. On the other hand, individual observations are generally recorded only in a finite number of moments, giving rise to a grid. As a consequence, we often work with discretized versions of the functional data. These two frameworks were addressed in scikit-fda by means of two classes: FDataGrid and FDataBasis respectively. They included methods to compute the basic statistics and to change from one representation to another. Furthermore, simple smoothing techniques were also covered.

From this base, the functionality implemented includes depth measures along with their applications and some fundamental methods for clustering functional data. Due to the complexity of functional spaces, they do not present a natural order such as the one found in the real line. An approach proposed to cover this lack of a definition of distance between functions resides in the idea of functional depth. Functional depth introduces an ordering within a sample and can provide a measure to analyze how similar observations are. Functional depth measures implemented include Fraiman and Muniz depth, the band depth and the modified band depth.

Having ranks of curves, the functional boxplot, an appealing visualization tool, is implemented as a natural extension of the classical boxplot. Another graphical tool for visualizing centrality and detecting outliers for functional data, the magnitude-shape plot, has been included. Moreover, once specific distances are defined, clustering algorithms can be applied straightforward to the data. Both K-means and Fuzzy K-means algorithms can be found in the package. The results can be plotted as an effective way to illustrate the characteristics that are not apparent from the mathematical models or summary statistics.

1.2. Document Structure

The paper is organized as follows. In Chapter 2, Functional Data Analysis is introduced. A brief overview is given followed by a deeper presentation of functional depth, its applications and clustering analysis in functional data. Each of the tools implemented is discussed in detail, both the practical context and theoretic calculations are explained. In Chapter 3, the solution implemented is described, which can be object oriented or consist in a functional approach. The possible customizations of the classes or methods are also exposed. The results obtained applying the functionality introduced to the package are shown in Chapter 4. In order to do this, a specific dataset is chosen and the different methods are applied to it: the boxplot, the magnitude-shape plot and the clustering algorithms. Chapter 5 contains the future work and conclusions. Finally, Appendix A contains the documentation found online for *scikit-fda*. First, the practical examples found in Jupyter Notebooks are appended and then, the documentation of the classes and functions implemented.

2 State of the Art - FDA: Depth Measures, Applications and Clustering

Nowadays, data are frequently obtained as trajectories or images in many research fields. Typically, a functional dataset consists of n curves measured at different time points, $T_k = \{t_1, t_2, ..., t_k\}$, which do not need to be equally spaced. In the example below 2.1, obtained from the Berkeley Growth Study [3], we can observe the heights of 10 children measured at a set of 31 ages, between 1 and 18 years old. The observations are recorded every 3 months during the first year, every year until the age of 8, and during the next ten years every half a year.

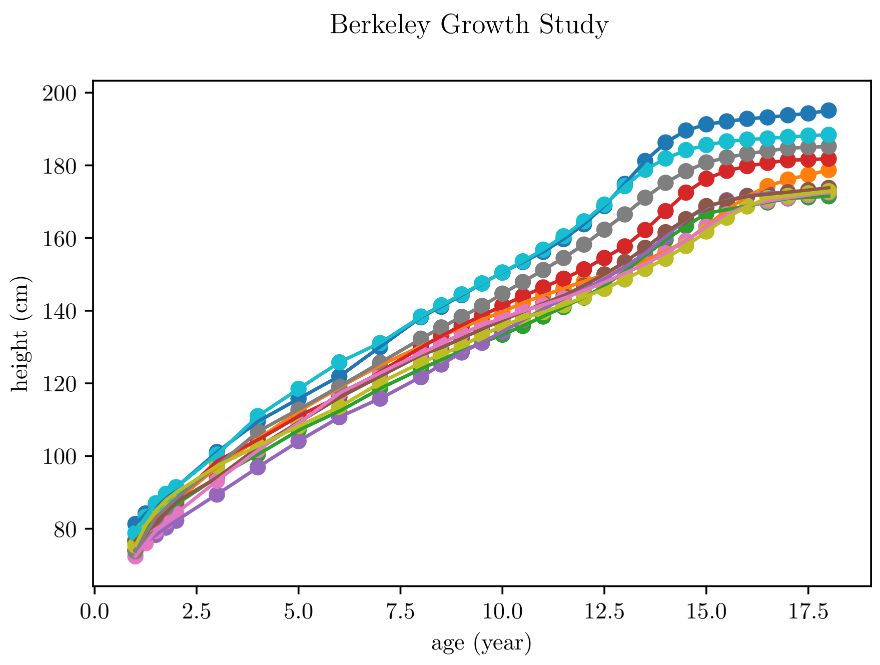


Figure 2.1: The heights of 10 children measured at 31 ages. The circles indicate the unequally spaced 31 measurements of each boy or girl.

Interesting questions that could be asked include, how much a child grow on average, at what age children have a more equally height, is this child abnormally tall/short (which can derive in a growing problem) or does this observation belong to a girl or a boy. This questions are related to the estimation of the central tendency of the curves, to the estimation of the variability among the curves, the detection of outlying curves, and the classification of such curves respectively.

Multivariate functional data are also considered along the paper. Each multivariate functional datum consists of a set of d curves. In this context, the Canadian Weather dataset [3] can be mentioned since it contains simultaneous observations of temperature and precipitation measurements ($d = 2$) recorded every day during a year in different Canadian weather stations. Only the measurements recorded in ten weather stations the first two weeks of the year are shown in Figure 2.2 in order to observe the equally spaced time points. Additionally, spatial surfaces can be considered, in which now the multiple dimensions are found on the domain. Examples include face recognition or neurological disorders assessment with brain images. These last ones can be found in [10].

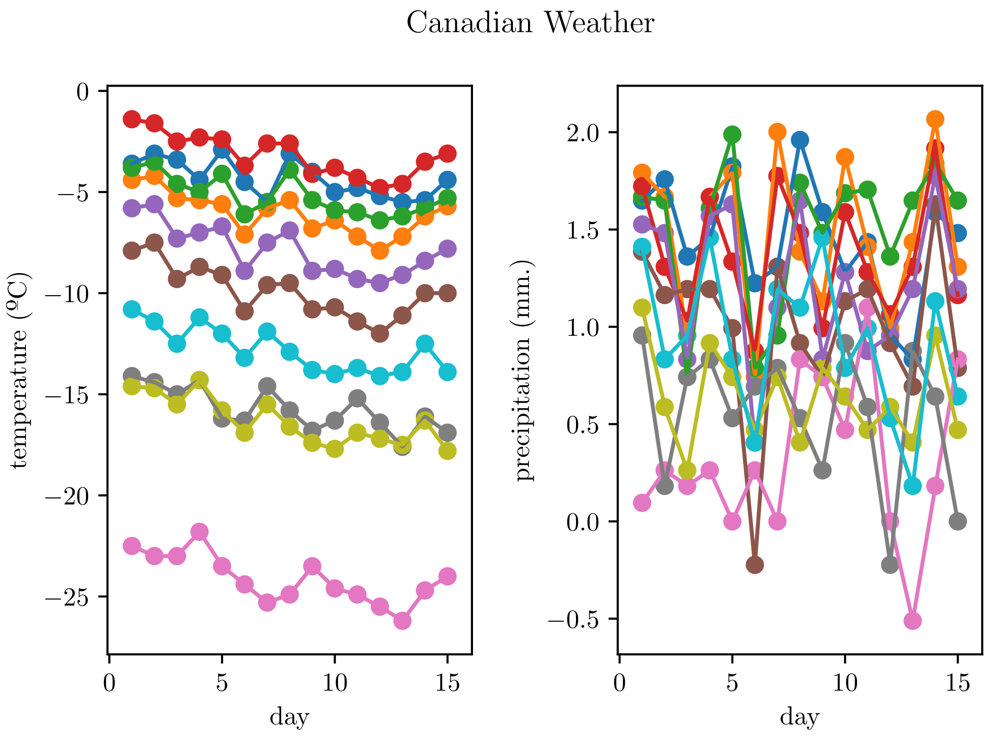


Figure 2.2: The temperatures and precipitations recorded in 10 different weather stations every day during a year. Only the fisrt 15 days of the year are plotted to show the equally spaced design points.

Notation

Before going into detail, let specify some notation. Consider a q-variate stochastic process $\mathbf{X} = (X_1, X_2, ..., X_q)^T : I \longrightarrow \mathbb{R}^q$ where the coordinates $X_i : I \longrightarrow \mathbb{R}$, for $1 \leq i \leq q$, are univariate stochastic processes. In most cases, I is a compact interval which belongs to $\mathbb{R}$. Nevertheless, this definition changes for multiple dimensions on the domain, I should be a compact set defined on the domain space of $\mathbf{X}$. For example, in a brain image the domain belongs to $\mathbb{R}^2$, so the compact set is the cross-section area of the brain. Besides, q, a positive integer, indicates the dimensionality of the functional data. If $q = 1$, univariate functional data are considered, as in the example of the Berkeley Growth Study, whereas if $q \geq 2$, multivariate functional data are found, as in the Canadian Weather dataset. $\mathbf{X}$ takes values in the space $C\left(I, \mathbb{R}^q\right)$ of real continuous functions with probability distribution

$F_{\mathbf{X}}$.

Furthermore, a stochastic process can be seen as a family of random variables. At each design point $t \in I$, $\mathbf{X}(t)$ is a q-variate random variable, or random vector, with probability distribution $F_{\mathbf{X}(t)}$. Along the document, the random variables are indexed by the set $T_k \subset I$. In the weather example, the 2-dimensional random vector of each day is composed of the temperature and precipitation measurements. If I is multidimensional, $\mathbf{X}(t)$ is called random field.

Additionally, for a sample of independent and identically distributed stochastic processes $\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_n$, the empirical distribution is denoted with $F_{\mathbf{X},n}$. Analogously, $F_{\mathbf{X}(t),n}$ is used for the random variables $\mathbf{X}_1(t), \mathbf{X}_2(t), ..., \mathbf{X}_n(t)$.

2.1. Functional Depth

Statistical depth provides a measure of centrality or outlyingness of an observation with respect to a given dataset or population distribution. The most central object is assigned the highest value while the least central, the lowest value. Those values are positive and bounded, without loss of generality, the explanation is given with the interval $[0,1] \subset \mathbb{R}$. Since in functional spaces there is no natural order, depths, which provide rankings of curves and a notion of centrality, are very useful. The uses of statistical depths include the construction of linear estimators, or functional boxplots, the detection of outlying observations or the classification of the data among others.

Although depth measures in $\mathbb{R}$ are trivial, this is not the case in the multivariate setting nor the functional. First, for each $t \in T_k$, consider the one dimensional random variable $X(t)$ and a depth measure denoted by $d\left(X(t), F_{X(t)}\right) : X(t) \longrightarrow [0,1]$. In this case, there are no doubts of the order independently of the metric considered. The properties are clear in $\mathbb{R}$ and the deepest observation is the median. One approach of extending this definition to the multivariate setting is to regard the depth value of a random vector as a weighted average of the marginal depths. As a consequence, the statistical depth measure $d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) : \mathbf{X}(t) \longrightarrow [0,1]$ for a multivariate random vector $\mathbf{X}(t)$ is calculated as:

$$d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) = \sum_{i=1}^{q} d\left(X_i(t), F_{X_i(t)}\right) \cdot p_i, \quad \sum_{i=1}^{q} p_i = 1, \tag{2.1}$$

where p_i, for $1 \leq i \leq q$ are the weights given to each of the dimensions. As a result, the multivariate depths are used to rank the marginal observations of a sample of multivariate functional data $\mathbf{X}_1(t), \mathbf{X}_2(t), ..., \mathbf{X}_n(t)$ found at each design point.

Noteworthy contributions proposed to rank multivariate data include the halfspace depth by Tukey (1975) [11] or the simplicial depth by Liu (1990) [12] and Zuo and Serfling (2000) [13] introduced the key properties that a depth function should verify: affine invariance, maximality at center, monotonicity and vanishing at infinity.

Finally, we still need to order functions over time. This is a much more difficult problem since, as a difference with respect to $\mathbb{R}^q$, in a functional space distinct metrics are no longer equivalent. This leads to very different rankings depending on the depth measure.

A first attempt to extend the previous definitions to the functional setting are the so-called integral depths, based on the integration of the marginal depths (univariate or multivariate) over time. Hence, for a stochastic process $\mathbf{X}$, an integral depth function is:

$$d\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I d\left(\mathbf{X}\left(t\right), F_{\mathbf{X}(t)}\right) \cdot w\left(t\right) dt, \quad \int_I w(t) = 1, \tag{2.2}$$

where $w(t)$ is a weight function defined on I. Usually, $w(t) = \{\lambda(I)\}^{-1}$ being $\lambda(\cdot)$ the Lebesgue measure.

Replacing the multivariate pointwise depth with Equation 2.1:

$$\begin{aligned} d\left(\mathbf{X}, F_{\mathbf{X}}\right) &= \int_I d\left(\mathbf{X}\left(t\right), F_{\mathbf{X}(t)}\right) \cdot w\left(t\right) dt \\ &= \int_I \left(\sum_{i=1}^{q} d\left(X_i\left(t\right), F_{X_i(t)}\right) \cdot p_i \right) \cdot w\left(t\right) dt \\ &= \sum_{i=1}^{q} \left(\int_I d\left(X_i\left(t\right), F_{X_i}\left(t\right)\right) \cdot w(t) dt \right) \cdot p_i \\ &= \sum_{i=1}^{q} d\left(X_i, F_{X_i}\right) \cdot p_i, \end{aligned} \tag{2.3}$$

another definition for a multivariate stochastic process depth function is obtained in terms of the univariate processes.

The numerous notions of depth encountered in the literature vary regarding robustness, sensitivity to reflect asymmetric shapes or computability. In any case, all of them allow to sort a sample of functional data $\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_n$ according to their depth obtaining the order statistics $\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, ..., \mathbf{X}^{(n)}$. If curves are sorted by their decreasing depth, the median (based on this depth) can be defined as the deepest point, $\mathbf{X}^{(1)}$. While the median is the observations that stays more *in the middle of the set* and has the highest depth value, the curves further away from the rest, with depth values proximate to zero, can be considered as the *outer skin* of the data and sometimes outliers.

The functional depths implemented in the package include Fraiman and Muniz, Band Depth and a modification of this last one, the Modified Band Depth. The first one is explained following the first approach 2.2 while the others, the second approach (inferred from 2.3) based in a weighted average of the univariate stochastic processes:

$$d(\mathbf{X}, F_{\mathbf{X}}) = \sum_{i=1}^{q} d(X_i, F_{X_i}) * p_i. \tag{2.4}$$

2.1.1. Fraiman and Muniz Depth

Fraiman and Muniz (FM) [14] proposed the first integral depth for functional data. The goal is to measure how much time every function is deep inside the dataset. Let start with the definition of the empirical, note the n subindex, cumulative distribution function used for a one dimensional random variable $X(t)$:

$$F_{X(t),n} = \frac{1}{n}\sum_{j=1}^{n} \mathbb{I}(X_j(t) \leq X(t)), \tag{2.5}$$

where $\mathbb{I}$ is the indicator function, $\mathbb{I}(A) = 1$ if A is true and $\mathbb{I}(A) = 0$ otherwise. The empirical version of this depth is:

$$d_n\left(X(t), F_{X(t),n}\right) = 1 - \left|\frac{1}{2} - F_{X(t),n}\right|. \tag{2.6}$$

In Figure 2.3, the relationship between the cumulative distribution function and the depth defined in 2.6 can be seen. Note the maximum depth value is obtained at the median for any distribution considered.

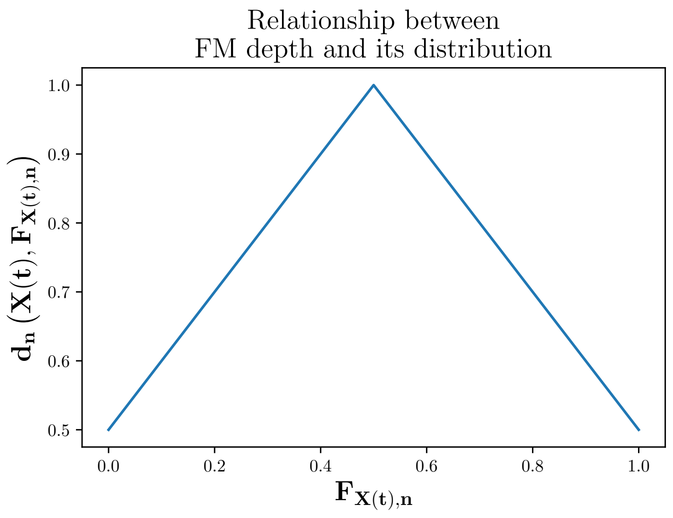


Figure 2.3: Relationship between Fraiman and Muniz depth and the cumulative distribution function considered.

Applying Equations 2.3 and 2.6, one possible implementation for multivariate stochastic processes using Fraiman and Muniz definition is the following:

$$\begin{aligned} d_n\left(\mathbf{X}, F_{\mathbf{X}}\right) &= \sum_{i=1}^{q}\left(\int_I d\left(X_i\left(t\right), F_{X_i}\left(t\right)\right) * w(t)dt\right) * p_i \\ &= \sum_{i=1}^{q}\left(\int_I \left(1 - \left|\frac{1}{2} - F_{X_i(t_j),n}\right|\right) * w(t)dt\right) * p_i \end{aligned} \tag{2.7}$$

It is a weighted average of the depth values of each of the dimensions of the image, in turn, this depth values are calculated as integrals of pointwise data depth values.

2.1.2. Band Depth and Modified Band Depth

Other implemented measure is the Band Depth (BD) introduced by López-Pintado and Romo (2009) [15] which is based on the graphic representation of functions. It makes use of the bands defined by their graphs on the plane. First, the original proposal for univariate processes is explained and the one for multivariate functional data later on.

Let remind the definition of graph of a function, a realization of a stochastic process X, $G(X) = \{(t, X(t)) : t \in I\}$. Then, the band in $\mathbb{R}^2$ delimited by h curves $X_{i_1}, X_{i_2}, ..., X_{i_h}$ is defined as:

$$B\left(X_{i_1}, X_{i_2}, ..., X_{i_h}\right) = \left\{(t, X(t)) : t \in I, \min_{r=1,...,h} X_{i_r}(t) \leq X(t) \leq \max_{r=1,...,h} X_{i_r}(t)\right\}. \tag{2.8}$$

The grey area in Figure 2.4 is the band delimited by the blue and green curves, but it is also the band delimited by three curves: the blue, the orange and the green. For any function X in the sample composed of curves $X_1, X_2, ..., X_n$, the quantity

$$S_n^{(h)}\left(X\right) = \binom{n}{h} \sum_{1 \leq i_1 \leq ... \leq i_h \leq n} \mathbb{I}\{G(X) \subset B\left(X_{i_1}, X_{i_2}, ..., X_{i_h}\right)\}, \quad 2 \leq h \leq n, \tag{2.9}$$

denotes the proportion of bands $B\left(X_{i_1}, X_{i_2}, ..., X_{i_h}\right)$ determined by h different curves containing the graph of X, where $\mathbb{I}$ is the indicator function. By computing the fraction of the bands containing the curve X, the bigger the value of band depth, the more central position the curve has.

With Equation 2.9, the band depth function of a trajectory is defined as:

$$d_{n,H}\left(X, F_{X,n}\right) = \sum_{h=2}^{H} S_n^{(h)}(X), \quad 2 \leq H \leq n. \tag{2.10}$$

As a consequence, using Equation 2.4, the multivariate functional band depth function [16] for q-dimensional data is given by:

$$d_{n,H}\left(\mathbf{X}, F_{\mathbf{X},n}\right) = \sum_{j=1}^{q} p_j \sum_{h=2}^{H} S_n^{(h)}(X_j), \quad 2 \leq H \leq n. \tag{2.11}$$

López-Pintado and Romo (2009) [15] also proposed a more flexible definition of the band depth, the Modified Band Depth (MBD) . Instead of using the indicator function in Equation 2.9, the proportion of time the curve is inside the band is measured. It becomes:

$$S_n^{(h)}\left(X\right) = \binom{n}{h} \sum_{1 \leq i_1 \leq ... \leq i_h \leq n} \lambda_k \left\{A\left(X; X_{i_1}, X_{i_2}, ..., X_{i_h}\right)\right\}, \quad 2 \leq h \leq n, \tag{2.12}$$

where $A\left(X; X_{i_1}, X_{i_2}, ..., X_{i_h}\right) = \{t \in I : \text{mín}_{r=1,...,h} X_{i_r}(t) \leq X(t) \leq \text{máx}_{r=1,...,h} X_{i_r}(t)\}$ and $\lambda_k = \lambda\left\{A\left(X; X_{i_1}, X_{i_2}, ..., X_{i_h}\right)\right\} / \lambda\left\{I\right\}$, where $\lambda\left\{\cdot\right\}$ is the Lebesgue measure.

The MBD is more convenient to obtain representative curves in terms of magnitude since less ties occur in terms of depth values among the observations. On the other hand, the band depth is preferred to detect shape differences. If curves do not intersect between them, the MBD turns out to give the same values as the band depth.

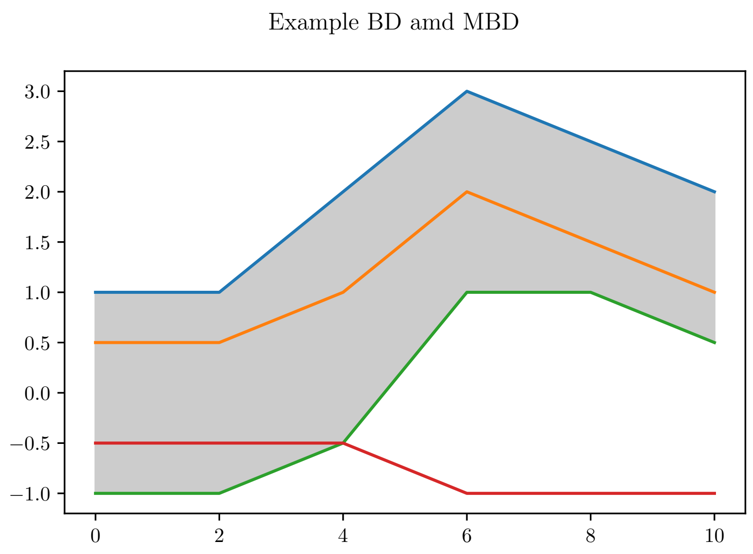


Figure 2.4: Basic example of BD and MBD applied to a dataset composed of four curves.

Differences between the BD and the MBD are illustrated with a simple example in Figure 2.4. It is composed of 4 observations ($n = 4$) and $H = 2$, so there are 6 bands, one for every pair of curves. First, note that each curve belongs to those bands that delimits. Furthermore, it can be observed that the orange curve is completely inside other two bands (blue-green and blue-red), consequently, its BD is $5/6$. On the other hand, the other three curves are not inside any other band resulting in 0.5 their depth value. The MBD values for the blue and orange observations stay the same since they do not intersect with other curves. However, the red observation belongs to the grey band 40 % of the time, so its MBD value consists of $(3 + 0{,}4 + 0{,}4)/6 = 0{,}63$, where the three comes from the three bands it is border of, and the two 0.4 of the proportion of time it spends in the green-blue and green-orange bands. Likewise the MBD of the green observation is $(3 + 0{,}6 + 0{,}6)/6 = 0{,}7$.

2.2. Functional Boxplot

Sun and Genton (2011) [17] introduced functional boxplots to visualize the result of ranking. Other informative exploratory tools include the rainbow plots and bagplots proposed by Hyndman and Shang (2010) [7] and the outliergram by Arribas-Gil and Romo (2013) [18]. The functional boxplot is an extension of the classical boxplot which displays five statistics: the median, the first and third quartiles and the non-outlying maximum and minimum observations; and indicates the outlying observations. Its construction is based on depth measures which define the order statistics and consequently, the functional quantiles.

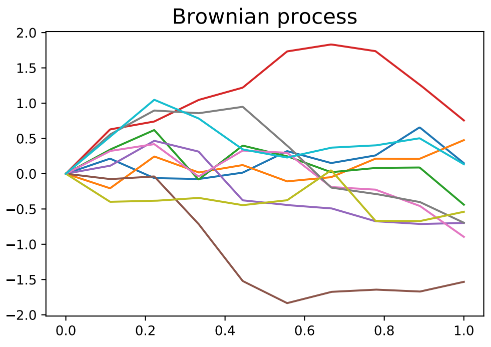
Analogously to the classical boxplot, the descriptive statistics shown in this plot include the 50 % central envelope, the median and the maximum non-outlying envelope. The 50 % central envelope, or 50 % central region, could be compared to the box of the classical boxplot which represents the interquartile range (IQR). More formally, the α-central region, C_α, $0 \leq \alpha \leq 1$, is delimited by the α proportion of deepest curves:

$$C_\alpha = \left\{ (t, y) : t \in T_k, \min_{r=1,\dots,\lceil \alpha \cdot n \rceil} \mathbf{X}^{(r)}(t) \leq y \leq \max_{r=1,\dots,\lceil \alpha \cdot n \rceil} \mathbf{X}^{(r)}(t) \right\}, \tag{2.13}$$

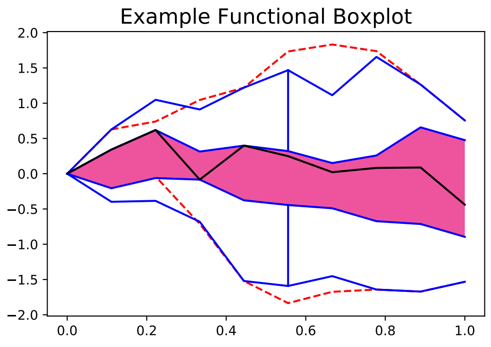
where $\lceil \alpha \cdot n \rceil$ represents the smallest integer not less than $\alpha \cdot n$. The median, as mentioned in the previous section, is $\mathbf{X}^{(1)}$, the most central curve with the largest depth value. It is always found inside the 50 % central region. It is a robust statistic to measure centrality.

The maximum non-outlying envelope is indicated by the whiskers (vertical lines extending from the box. The maximum non-outlying envelope is composed of the highest values (without taking into account outliers) found at each design point. So first, the outliers must be identified. The cutoff values are the fences obtained by inflating the the borders of the central region $C_{0,5}$ by 1.5 times the range of the $C_{0,5}$. The observations outside the fences are flagged as outliers.

In Figure 2.5(a), a dataset composed of ten random realizations of a Brownian process is shown. Alongside, in 2.5(b), the functional boxplot built from this data can be found. The median is plotted in black, the envelopes and the vertical lines in blue, the $C_{0,5}$ in pink and the outliers in red. Note that only the median and the outliers are real observations.



(a) Raw data



(b) Functional boxplot

Figure 2.5: On the left, a dataset composed of ten random realizations of a Brownian process is represented and on the right figure its boxplot.

The resulting functional boxplot reveals useful information when looking at their shape, length, position and size. The spacings between the different parts of the box, intuitively indicate the degree of dispersion and skewness in the data. Note that in the functional context, robust methods are possibly more useful than in multivariate problems since there are more ways in which outliers affect functional statistics. A curve could be an outlier without having any unusually large value; besides magnitude, shape is also important.

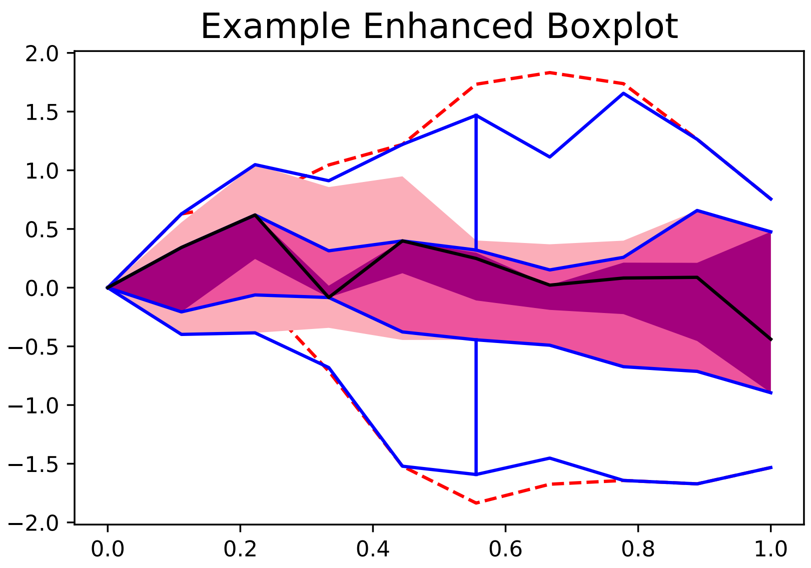
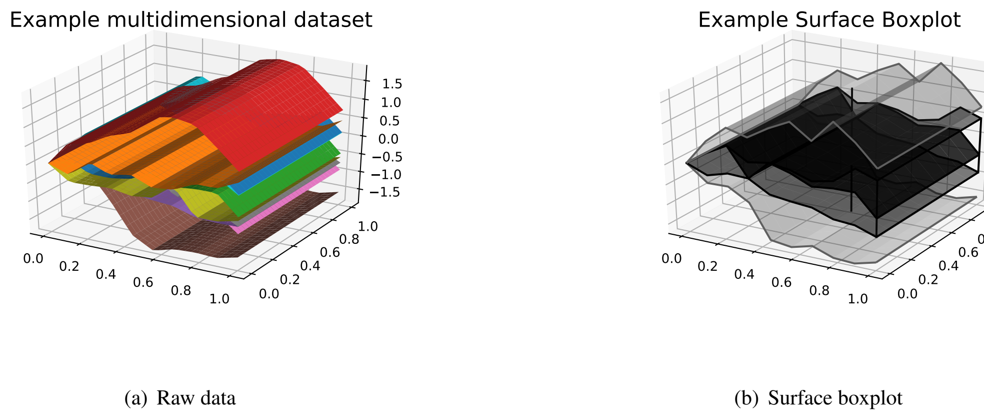


Figure 2.6: Enhanced boxplot of the dataset shown in 2.5(a). The darkest pink color represents the $C_{0,75}$ while the lightest, the $C_{0,25}$

Moreover, there exists an enhanced functional boxplot in which the 25 % and 75 % central regions are provided as well (Figure 2.6) and a surface boxplot [19], in which $I \subset \mathbb{R}^2$. To illustrate the surface boxplot, a functional dataset with a two-dimensional domain space was generated extending the values of the dataset shown in 2.5(a) along other axis, resulting in 2.7(a). Its surface boxplot is included in 2.7(b).



(a) Raw data (b) Surface boxplot

Figure 2.7: A dataset of multidimensional functional data is shown alongside its surface boxplot.

2.3. Magnitude-Shape Plot

Outliers in functional spaces are difficult to detect due to the diverse characteristics to consider. There are two big families of outliers: magnitude outliers (flagged by the boxplot) and shape outliers (the boxplot is inadequate). As a consequence, other tools are needed.

Dai and Genton (2018) [20] [21] contributed to the functional data toolbox with the magnitude-shape plot. It is another graphic method that helps visualizing both magnitude and shape outlyingnes of univariate and multivariate functional data. Given a functional dataset, the shape outlyingness of these functional data is found on the vertical axis, while both the level and the direction of the magnitude outlyingness are plotted on the horizontal axis or plane. Moreover, it provides a criterion to identify various types of outliers that could lead to severe biases in modeling or forecasting functional data.

Directional outlyingness

Note that outlyingness functions are equivalent to statistical depths in an inverse sense. If the depth function for a multivariate random variable $\mathbf{X}(t)$ with distribution function $F_{\mathbf{X}(t)}$ is denoted by $d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)$, its outlyingess is given by:

$$o\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) = \frac{1}{d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)} - 1. \tag{2.14}$$

The magnitude-shape plot measures centrality of functional data by considering both level and direction of deviation from the central region. It adds direction to the conventional concept of outlyingness, which is crucial in describing centrality of multivariate functional data [22]. To capture both magnitude and direction of outlyingness, direction is added to the outlyingness function as follows:

$$\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) = o\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) \cdot \mathbf{v}(t) = \left\{\frac{1}{d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)} - 1\right\} \cdot \mathbf{v}(t), \tag{2.15}$$

where d can be any conventional depth measure, and $\mathbf{v}(t) = (\mathbf{X}(t) - \mathbf{Z}(t)) / \|\mathbf{X}(t) - \mathbf{Z}(t)\|$, being $\mathbf{Z}(t)$ the unique median of $F_{\mathbf{X}(t)}$ with respect to d and $||\cdot||$ is the L_2 norm. In other words, $\mathbf{v}$ is the unit vector pointing from $\mathbf{Z}(t)$ to $\mathbf{X}(t)$ and basically, indicates the spatial sign of $\{\mathbf{X}(t) - \mathbf{Z}(t)\}$.

For functional data, there are 3 different measures of directional outlyingness:

1. Mean directional outlyingness ($\mathbf{MO}$):

$$\mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I \mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) \cdot w(t)dt \tag{2.16}$$

   $w(t)$ is a weight function defined on I. $\mathbf{MO}$ describes the relative position, both distance and direction, of $\mathbf{X}$ on average to the center curve. Its norm, $||\mathbf{MO}||$, is regarded as the magnitude outlyingness of $\mathbf{X}$.

2. Variation of directional outlyingness (VO):

$$VO\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I \|\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) - \mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right)\|^2 \cdot w(t)dt \tag{2.17}$$

   It measures the change of $\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)$ in terms of both the norm and direction across the whole interval. It is regarded as shape outlyingness. Functional data are usually classified by their shapes rather than scales because variation outlyingness accounts for both pointwise outlyingness and change in their directions.

3. Functional directional outlyingness (FO):

$$FO\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I \|\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)\|^2 \cdot w(t)dt \tag{2.18}$$

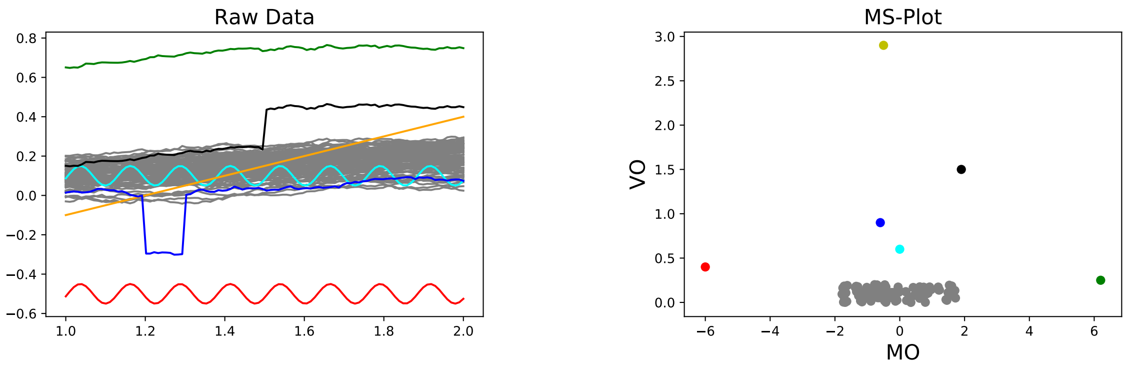
   It represents the total outlyingness and the concept is similar to the the one of classical functional depth. However, classical functional depth maps $\mathbf{X} \in C\left(I, \mathbb{R}^q\right)$ to the compact interval $[0,1] \in \mathbb{R}$ whereas the functional directional outlyingess maps $\mathbf{X}$ to $\left(\mathbf{MO}^T, VO\right) \in \mathbb{R}^q \times \mathbb{R}^+$ which gives more flexibility to analyze curves.

The functional directional outlyingness is linked to the other first two measures with the relationship:

$$FO\left(\mathbf{X}, F_{\mathbf{X}}\right) = \left\|\mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right)\right\|^2 + VO\left(\mathbf{X}, F_{\mathbf{X}}\right) \tag{2.19}$$

The above decomposition of the functional directional oulyingness in magnitude and shape provides great flexibility for describing centrality of functional data and diagnosing potentially abnormal curves. When the curves are parallel, the shape otlyingness (VO) is zero and a quadratic relationship can be observed between the functional and magnitude outlyingness: $FO = \|\mathbf{MO}\|^2$.

The magnitude-shape plot indeed shows a scatter group of points $\left(\mathbf{MO}^T, VO\right)$ for a sample of functional data. It is used to illustrate the centrality of curves with a response space up to two dimensions. When the dimension is higher, the points are defined by $\left(\|\mathbf{MO}\|^T, VO\right)$. The overall magnitude outlyingness is still presented, however the shape outlyingness is shown without direction.



(a) Esta es una subfigura1 (b) Esta es otra subfigura2

Figure 2.8: A group of curves with various types of outliers and its MS-plot.

In Figure 2.8 a functional dataset is plotted alongside its corresponding magnitude-shape plot to illustrate the basic concepts. The cluster of grey points found in the below mid-part of the graph corresponds to the central curves; both magnitude and shape outlyingness are small. In the vertical axis the variation outlyingness is plotted, so shape outliers appear on the top part of the graphic. The magnitude outlyingness is plotted on the horizontal axis, so shifted outliers appear on the sides of the graphic. The side is decided according the direction of their shifts.

Additionally, the magnitude shape plot provides a frontier to separate regular data from outliers. The outlier detection method is designed with the directional outlyingness, more specifically, using the empirical discrete form of the magnitude and the shape outlyingness, $\mathbf{MO}_{T_k,n}$ and $VO_{T_k,n}$ respectively.

$$\mathbf{MO}_{T_k,n}\left(\mathbf{X}_n, F_{\mathbf{X},n}\right) = \sum_{i=1}^{k} \mathbf{O}_n\left(\mathbf{X}(t_i), F_{\mathbf{X}(t_i),n}\right) \cdot w_n(t_i). \tag{2.20}$$

The directional outlyingness maps one q-variate curve to a $(q+1)$-dimensional vector $\mathbf{Y}_{k,n} = \left(\mathbf{MO}_{T_k,n}{}^T, VO_{T_k,n}\right)^T$ which is well approximated with a multivariate normal distribution when $\mathbf{X}$ is generated by a stationary Gaussian process. Hardin and Rocke (2005) results [23] can be used under these suppositions to detect potential outliers from $\mathbf{Y}_{k,n}$. These results are indicated in the following steps:

1. Calculate the square robust Mahalanobis distance of $\mathbf{Y}_{k,n}$ based on a sample of size $h \leq n$:

$$RMD^2\left(\mathbf{Y}_{k,n}, \widetilde{\mathbf{Y}}^*_{k,n,J}\right) = \left(\mathbf{Y}_{k,n}, \widetilde{\mathbf{Y}}^*_{k,n,J}\right)^T \mathbf{S}^*_{k,n,J}{}^{-1}\left(\mathbf{Y}_{k,n}, \widetilde{\mathbf{Y}}^*_{k,n,J}\right), \tag{2.21}$$

   where J designates the group oh h points estimated by the Minimum Covariance Determinant algorithm, giving rise to the covariance matrix $\mathbf{S}^*_{k,n,J} = \sum_{i\in J}\left(\mathbf{Y}_{k,n,i} - \widetilde{\mathbf{Y}}^*_{k,n,J}\right)\left(\mathbf{Y}_{k,n,i} - \widetilde{\mathbf{Y}}^*_{k,n,J}\right)^T$ where $\widetilde{\mathbf{Y}}^*_{k,n,J} = h^{-1}\sum_{i\in J}\mathbf{Y}_{k,n,J}$.

2. Approximation of the tail of the distance distribution with a Fisher's F distribution as follows:

$$\frac{c(m-q)}{m(q+1)} RMD^2\left(\mathbf{Y}_{k,n}, \widetilde{\mathbf{Y}}^*_{k,n,J}\right) \sim F_{q+1,m-q}$$

   ,

   where c and m are real numbers used to determine the degrees of freedom of the F distribution and the scaling factor.

3. Flag a curve as an outlier when its distance satisfies:

$$\frac{c(m-q)}{m(q+1)} RMD^2\left(\mathbf{Y}_{k,n}, \widetilde{\mathbf{Y}}^*_{k,n,J}\right) > C,$$

   where C is a cutoff value chosen as the α-quantile of $F_{q+1,m-q}$; $\alpha = 0{,}993$ is used in the classical boxplot for detecting outliers under a normal distribution.

2.4. Clustering Algorithms

Functional depth also indicates how similar observations are and therefore, it can be used in functional classification and inference. For example, Flores, Lillo and Romo [24] used depth measures to perform homogeneity tests. Baillo, Cuevas and Fraiman [25] provide a survey of the literature concerning classification of functional data. This section is focused on some basic notions of clustering [26]. Cluster analysis is a collection of unsupervised classification techniques for grouping objects or segmenting datasets into subsets of data called clusters. Clustering methods try to assign similar objects that share common characteristics into the same cluster.

There are three categories of clustering algorithms: hierarchical, non-hierarchical or flat, and a mixed approach. In practice, their use is limited to their complexity, efficiency or availability in current software. Furthermore, the choice of the algorithm to run on a certain dataset depends on the sample size, structure or even the proper goals of the cluster analysis.

The methods of the k-means family are non-hierarchical partitioninng algorithms with good clustering results in shorter times and on larger datasets compared to hierarchical ones. As a consequence, they are the most popular algorithms found in exploratory analysis and data mining applications.

By using a clustering algorithm, the dataset $\mathbf{X} = \{\mathbf{X}_1, \mathbf{X}_2, , ..., \mathbf{X}_n\}$ is divided into k groups aiming at obtaining low within-cluster and high between-cluster heterogeneity. In other words, a cluster contains objects as similar to each other as possible and as far from other objects in other clusters as possible. To measure the closeness of the observations, a distance measure is needed. Usually, the standard L_2 distance is used but the distance can be calculated using statistics based on depth measures. The parameter indicating the number of clusters, k, is known or fixed a priori before running the algorithm. Clusters are described by their member objects and by their centers, which are usually the centroids. A centroid is the point that minimizes the sum of distances between itself and each point in the cluster. The prototype vector of cluster centroids is denoted by $\mathbf{C} = [\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_n]$, $\mathbf{C}_i \in \mathbb{R}^d$.

2.4.1. K-means

K-means (KM) , or alternatively Hard C-Means, is an iterative clustering algorithm that computes clusters in order to minimize the sum of distances from each object to its cluster centroid. In other words, the following function must be minimized:

$$J_{KM}\left(\mathbf{X}; \mathbf{C}\right) = \sum_{i=1}^{c} \sum_{j=1}^{n_i} D_{ij}^2, \quad \sum_{i=1}^{c} n_i = n, \tag{2.22}$$

where n_i indicates the number of observations in the i^{th} cluster and D_{ij} denotes the distance chosen between the j^{th} observation and the i^{th} center.

In each iteration, the observations are reassigned between clusters until a minimum point of J_{KM} is reached. The algorithm steps are captured in 2.1. The *random* function indicates the selection of k random trajectories from $\mathbf{X}$, *distance* calculates the distance between observation units and cluster centroids, the *partition* method assigns each observation to the cluster of the closest centroid and *centroids* updates the centroids using: $\mathbf{C}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} \mathbf{X}_j$, for $1 \leq i \leq k$.

K-means is a fast, robust, and easy to implement algorithm. It assigns each object to exactly one cluster, so it gives comparatively good results if clusters are distinct or well-separated. Nevertheless, it is not as reliable in finding overlapping clusters with regard to form or scattering. Also, it fails to cluster

```
input  : $\mathbf{X}$, optionally $\mathbf{C}$
output: $P$ (array of length $n$): a partition of $\mathbf{X}$, $\mathbf{C}$
1 $\mathbf{C} \leftarrow \mathbf{C}$ or random( $\mathbf{X}$ );
2 cond $\leftarrow$ true;
3 while cond do
4     $\mathbf{C}_{old} \leftarrow \mathbf{C}$;
5     $\mathbf{D} \leftarrow$ distance( $\mathbf{C}$, $\mathbf{X}$ );
6     $P \leftarrow$ partition( $\mathbf{D}$, $\mathbf{X}$ );
7     $\mathbf{C} \leftarrow$ centroids( $P$, $\mathbf{X}$ );
8     cond $\leftarrow$ not_equal( $\mathbf{C}$, $\mathbf{C}_{old}$ );
9 end
```

Algorithm 2.1: K-means algorithm.

noisy data and is not invariant to non-linear transformations of data.

2.4.2. Fuzzy K-means

The Fuzzy K-means (FKM) , or Soft C-Means, algorithm is an extension of KM that was introduced to overcome the aforementioned disadvantages of KM. It is a soft algorithm clustering fuzzy data and it assigns each object to different clusters with varying degrees of membership. These values range between 0 and 1. It is used in a wide area of applications although it has a relatively higher computational cost.

Analogously to KM, the Fuzzy K-Means computes clusters iteratively in order to minimize the following function:

$$J_{FKM}\left(\mathbf{X};\mathbf{U},\mathbf{V}\right) = \sum_{i=1}^{k}\sum_{j=1}^{n} u_{ij}^{m} \cdot D_{ij}^{2}. \tag{2.23}$$

The difference with respect to KM is the used of *weighted* square errors. $\mathbf{U}$ is the membership matrix $k \times n$ that represents the fuzzy clustering of the dataset $\mathbf{X}$. Each of its entrances, u_{ij}, indicates the membership value of the j^{th} observation to the i^{th} cluster. m is the fuzzifier parameter or weighting exponent, $m \in [1, \infty)$. As m approaches to 1, the clustering tends to become crisp, on the contrary, as m goes to ∞, the clusering becomes fuzzified. It is usually fixed as 2.

Fuzzy K-Means must be run under these three constrainsts:

1. $u_{ij} \in [0,1]\,, \quad 1 \leq i \leq k \quad and \quad 1 \leq j \leq n,$
2. $\sum_{i=1}^{k} u_{ij} = 1, \quad 1 \leq j \leq n,$
3. $0 < \sum_{j=1}^{n} u_{ij} < n, \quad 1 \leq i \leq k.$

The algorithm can be found in 2.2, in which the *random* and *distance* functions are the same as in

KM. The *membership_values* method calculates the membership values of data points to each cluster with: $u_{ij} = \left[\sum_{c=1}^{k} (D_{ij}/D_{cj})^{\frac{2}{m-1}}\right]^{-1}, \quad 1 \leq i \leq k$ and $1 \leq j \leq n$ and the *centroids* function updates the centroids using: $\mathbf{C}_i = \frac{\sum_{j=1}^{n} u_{ij}^m \mathbf{X}_j}{\sum_{j=1}^{n} u_{ij}^m}, \quad 1 \leq i \leq k$.

```
input  : $\mathbf{X}$, optionally $\mathbf{C}$
output: $\mathbf{U}$, $\mathbf{C}$
1 $\mathbf{C} \leftarrow \mathbf{C}$ or random( $\mathbf{X}$ );
2 cond $\leftarrow$ true;
3 while cond do
4     $\mathbf{C}_{old} \leftarrow \mathbf{C}$;
5     $\mathbf{D} \leftarrow$ distance( $\mathbf{C}$, $\mathbf{X}$ );
6     $\mathbf{U} \leftarrow$ membership_values( $\mathbf{D}$, $\mathbf{X}$ );
7     $\mathbf{C} \leftarrow$ centroids( $\mathbf{U}$, $\mathbf{X}$ );
8     cond $\leftarrow$ not_equal( $\mathbf{C}$, $\mathbf{C}_{old}$ );
9 end
```

Algorithm 2.2: Fuzzy K-means algorithm.

There are many other approaches to cluster data, for example using the coefficients of the B-Splines basis [27] instead of the actual observation values or the SeqClusFD algorithm in which the functional boxplot is involved [28].

3 Software Development Process

In the previous section, the core functionalities implemented were exposed in detail. In this one, the development of the Python package, *scikit-fda*, is going to be explained. First of all, an agile methodology which uses incremental, iterative work cycles was followed. These cycles were assessed by the developing team in regular meetings which were hold every week. In each cycle the phases of the Waterfall model (requirements analysis, software design, implementation, testing and integration) were completed for a specific functionality, before proceeding to the next increment.

3.1. Analysis

The package requirements were already specified from the beginning of the project started last year by former student Miguel Carbajo [9]. However, an overview is given to remind the main ones. The package is written in Python. It is a general purpose language and multi-paradigm, which implies flexibility in the implementations of a technique. Moreover, it is great for prototyping due to the dynamic typing and the possibility to use it as a Read-Eval-Print-Loop (REPL) with for example, Jupyter Notebooks. In addition, it has a rich ecosystem with a great variety of modules in different fields. Pandas can be found for statistical calculations, Matplotlib for plotting and, Numpy and Scipy stand out in Python scientific computing modules. Algorithms which are already implemented in those last packages can be reutilized and assure efficiency due to the lower-level languages they are programmed in, such as Fortran or C. Numpy adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. SciPy contains modules for optimization, linear algebra, integration, interpolation and other tasks common in science and engineering. Moreover, SciPy builds on the NumPy array object.

The package must be integrated with Python science environment, therefore, it follows scikit-learn API. Scikit stands for Scipy-Toolkits which are specialized science add-on packages for Scipy. They are developed separately and independently from the main distribution.

Furthermore, it is an open-source, scalable, software package which implies the presence of an easy mechanism to contribute to it, along with an extensive test-bench of unit tests and continuous integration procedures. Also, documentation is important both for the general audience the package is intended for and the potential developers of the package.

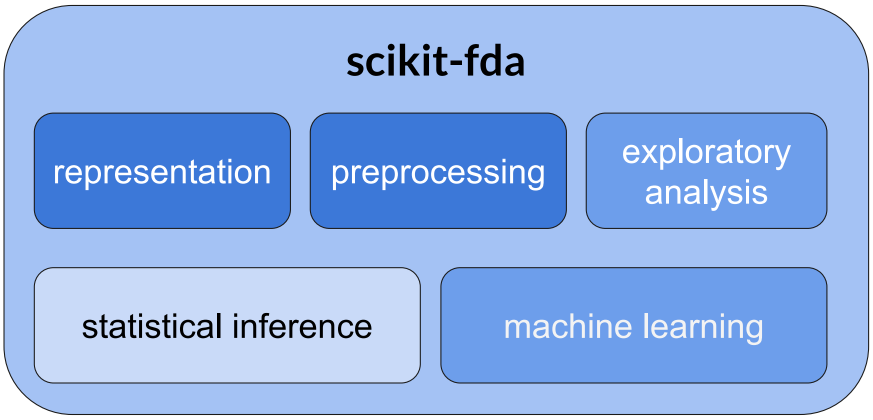


Figure 3.1: Division of scikit-fda functionality. The intensity of the colors reflects the level of functionality implemented, being the area of inference the one which requires more contribution.

Finally, the goal is to include as much functionality as possible to cover the areas shown in Figure 3.1 [29]. Regarding this project, the functionality implemented is proportional to the 360 hours of a degree work and it was decided just after each cycle. The specific analysis for the functionalities implemented consisted basically in researching to decide what to include in the package, both in the mathematical and computer science fields, to determine the most innovative, popular and viable methods.

3.2. Design

As commented in the Introduction 1.1, the package in its initial version included two modules containing both representations of functional data, as a grid (FDataGrid) or as a linear combination of basis functions (FDataBasis); along with a basic *math* module and a couple related to *smoothing* techniques. During this year, the package structure has considerably grown due to the number of people working on it. The following paragraphs describe the implementation of the concepts explained in Chapter 2.

Depth Measures

The three implemented depth functions: Fraiman and Muniz depth, Band depth and Modified Band depth, can be found in the *depth* module. This module is found in the *exploratory analysis* directory due to its applications. The three methods follow the signature *depth_name(fdatagrid, pointwise=False)*, where the first argument expects a FDataGrid object and the second one indicates wether to return also the pointwise univariate depth. The second parameter was added because, by default, in Equation 2.3, $w(t) = 1/|T_k|$, $1 \leq j \leq k$ and $p_i = 1/q$, $1 \leq i \leq q$. So, if the functional result is desired to be calculated with other weight values, it is possible to obtain it from the pointwise univariate depth.

Besides, the code is scalable since the adding of a new method does not imply to modify anything.

Regarding the BD and MBD, in Equation 2.10 the number of curves determining a band, h, can be any integer between 2 and H. The order of curves induced by the depths are very stable in H. So to avoid computational issues, $H = 2$ is used because a fast method was proposed by Sun and Genton (2012) [30] based on matrix (or, in higher dimensions, tensor) ranks. The pseudocodes of both functions are included. Both of them assume univariate functional datasets represented in a matrix $M, n \times k$. In 3.1, *na[i]* denotes the number of curves that are completely above the i^{th} curve, whereas *nb[i]* denotes the number of curves that are completely below the i^{th} curve, $1 \leq i \leq n$. *na*, *nb* and *depth* are n-dimensional vectors. In 3.2, *na*, *nb* and *match* are $n \times k$ matrices while *depth* stays the same, a vector of length n.

```
input  : M
output: depth
foreach $j \leftarrow 1$ to $k$ do
    R[,j] ← rank(M[,j]);
end
foreach $i \leftarrow 1$ to $n$ do
    na[i]← n - max(R[i,]);
    nb[i]← min(R[i,])- 1;
end
depth ← (na * nb + n- 1) / nchoose2 ;
```

Algorithm 3.1: Band Depth pseudocode.

```
input  : M
output: depth
foreach $j \leftarrow 1$ to $k$ do
    R[,j] ← rank(M[,j]);
end
na ← n - R;
nb ← R - 1;
match ← na * nb;
foreach $i \leftarrow 1$ to $n$ do
    proportion ← sum(match[i, ]) / k ;
end
depth ← (proportion + n - 1) / nchoose2 ;
```

Algorithm 3.2: Modified Band Depth pseudocode.

Boxplot

The boxplot functionality is implemented in two classes: Boxplot and SurfaceBoxplot. They can be found in the *boxplot* module, inside the *visualization* directory which in turn can be found in the *exploratory analysis* one. Both classes support FDataGrid objects with as many dimensions on the image as desired whereas the first class only admits one dimensional domains and the second one bidimensional domains. A graph for each dimension on the image is returned, so domain spaces only have sense up to two dimensions.

Both classes inherit from an abstract one, FDataBoxplot, whose attributes contain the descriptive statistics: median, central_evelope and outlying_envelope. To calculate them, the depth function and the factor to identify outliers can be customized. By default, the *modified_band_depth* function and the value 1.5 for the factor are applied. To obtain the graphic, the plot function must be called, and the colormap used can be chosen. In interactive mode, the plot is the default representation of the class.

Although the procedure to obtain the mathematical results is very similar in both cases, the plotting part is quite different. In the first case, a line is plotted for each observation while in the second one, a surface is shown. For clarity reasons, the SurfaceBoxplot does not show outliers nor has the possibility to produce an enhanced boxplot as in the case of the *Boxplot* class. Indeed, in the *Boxplot* class, any α central regions can be selected to appear.

Magnitude-Shape Plot

This plot is implemented in a class named the same way, MagnitudeShapePlot, in the *magnitude_shape_plot* module found in the *visualization* directory. This module also contains the method to calculate the directional outlyingness of the FDataGrid object considered. This method is used by the MagnitudeShapePlot class in which the depth function, along with the dimension and pointwise weights, can be customized.

Once the directional outlyingnes has been computed, the mean and the variation of the directional outlyingness are calculated to obtain the actual points of the graphic. The norm implemented by default for Equations 2.16, 2.17 and 2.18 is the L_2-norm defined as $\|f\| = \left(\int_I |f|^2 dx\right)^{\frac{1}{2}}$, where $\|\cdot\|_*$ denotes a vectorial norm (also the L_2-norm by default).

Finally, the outliers are calculated using the MinCovDet class provided by scikit-learn and the cutoff value can be adjusted by means of the parameter alpha.

Clustering Algorithms

The clustering functionality can be found in the *clustering* directory, inside *machine learning*. Specifically, a module called *base_kmeans* was created to include K-means algorithms: K-means and Fuzzy K-means, which are implemented in two classes named after them. Both classes inherit from BaseKMeans class which follows scikit-learn API with scrutiny. This latter class inherits from BaseEstimator, ClusterMixin and TransformerMixin contained in the aforementioned package and implements the *fit*, *transform* and *score* methods among others. This implementation follows the one found in scikit-learn of the KMeans class, which uses vectorial L_2-norm to compute distances. By default, the functional KMeans class included in the package also utilizes the L_2-norm (but for functions, as explained in the previous subsection), nevertheless, it admits any suitable distance function.

In order to visualize the results, a module named *clustering_plots* located in the *visualization* directory was created. It includes three methods: *plot_clusters*, *plot_cluster_lines* and *plot_cluster_bars*. The first one plots the raw data by colors, in which each color indicates a cluster. The other two methods are applicable only for the Fuzzy K-means class results which help to visualize the degree of membership of each observation to each cluster, by means of a kind of parallel coordinates plot or histogram plot respectively.

3.3. Coding, Documentation and Testing

Any open-source package needs to be composed of scalable source code in order for programmers to contribute in its development. It is always easier if there are guidelines that assure consistency and make the code more readable. As a consequence, standard PEP 8: The Style Guide for Python Code is followed. It contains coding conventions comprising the standard library in the main distribution, which include information about indentation, maximum line length (79 characters), blank lines, encodings (PEP 263) or naming conventions.

In turn, PEP 8 references PEP 257 standard for documentation. It describes docstrings (documentation strings) semantics and conventions. Docstrings are found at the beginning of all public modules, functions, classes and methods, and they should be kept short, simple and avoiding repetitions.

Another point to take into account for a collaborative software is testing in order to produce a quality product and detect bugs. The testing framework used is based on *unittest*, which is indeed the de facto standard in this area. It constitutes the Python language version of JUnit, Java's testing framework. It supports test automation, sharing setup and shutdown code for tests, or aggregation of tests into collections. Nevertheless, the tool employed for running the tests is *pytest*, which contains more features, including more informative tracebacks, stdout and stderror capturing, or stopping after a fixed number of failures; and supports more complex functional testing.

Another positive aspect of following standards, is the existence of tools that automatize their use. Personally, I used *PyCharm* to write the source code. It has a number of settings to configure the Python environment. The docstring format can be chosen among plain, reStructuredText, Epytext, Numpy or Google. The one selected is the Google standard which follows PEP8 and has a more *pythonic* syntax. PyCharm can understand the docstrings, aid with their generation and use them for quick fixes and coding assistance. *PyCharm* also allows to configure *Sphinx* working directory.

Sphinx is a documentation generator that converts reStructuredText, an extensible, markup language used by the Python community for technical documentation, into HTML websites or other formats such as pdf. It autogenerates documentation from the source code, writing mathematical notation or highlighting code. Moreover, it is linked with *doctest* that tests the code by running the examples embedded in the documentation and verifies if the expected results were produced.

3.4. Version Control and Continuous Integration

Due to the high number of potential contributors, there must be some kind of coordination among them. Fortunately, tools for version control are already spread. In this case, *Git* has been used. It is a distributed version-control system that tracks changes in the source code of any file during the software development and gives support to distributed, non-linear worflows. A Git directory is a repository with full history and version tracking abilities, independent of network or a central server access.

More specifically, the package can be found on *GitHub*, in `https://github.com/GAA-UAM/scikit-fda/wiki`, a web-based hosting service using Git for version control. Apart from Git functionalities, it offers its own features which include access control and regarding collaboration between programmers, bug tracking, feature request or task management. The repository can be read and cloned but writing is controlled by the owners.

The Git flow is based on branches and supports teams and projects where deployments are made regularly. It consists on the following steps: create a branch from the repository, add commits, open pull request, discuss and review code, deploy for testing and finally, merge. The repository contains two main branches: a *master* branch in which the releases available can be found and the *develop* branch, into which the *feature* branches are merged during the development process.

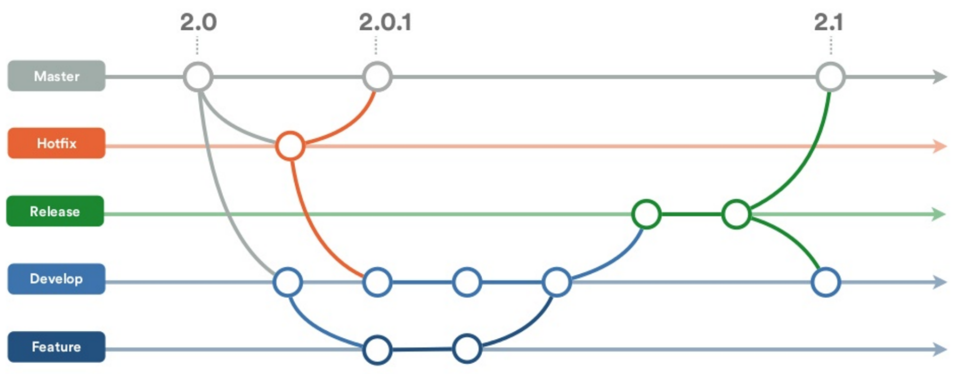


Figure 3.2: Git flow.

GitHub also provides some software as a service integrations to add extra features to projects. Travis CI can be found among the hosted continuous integration services used to build and test software projects. It gives full control over the build environment to adapt it to the code and runs the tests every time a push is done. Testing is not a just one-time task. Additionally, it gives support to more than one version of Python simultaneously.

Another hosting platform linked to GitHub is *Read the Docs* which generates documentation compiled with Sphinx. It simplifies the technical documentation by automatically building, versioning and hosting the generated documentation in its website. The package documentation can be found here.

4 Results

In this section the results of the functionality implemented are shown applied to the Canadian Weather dataset described in the introduction of Chapter 2. More specifically, only the temperatures are going to be studied, dealing with one dimensional functional data. The Canadian Weather dataset can be obtained from the *datasets* module. It contains functions to generate synthetic datasets or to retrieve specific datasets which are fetched from CRAN or UCR. In Figure 4.1, the raw data is plotted to show the curves to be analyzed. They are divided according to the target. In this case, it includes the different climates to which the weather stations belong to: Arctic, Atlantic, Continental and Pacific.

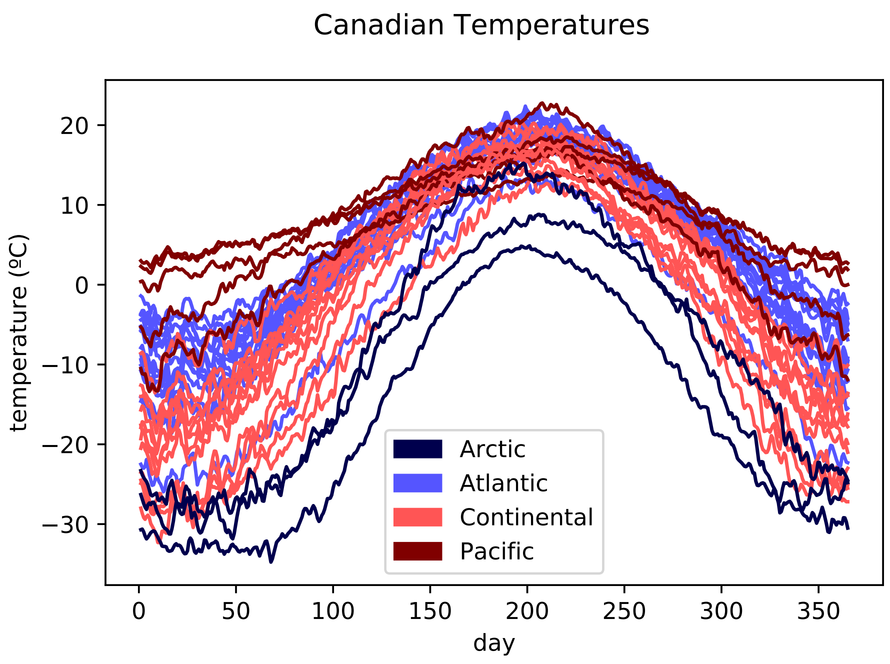
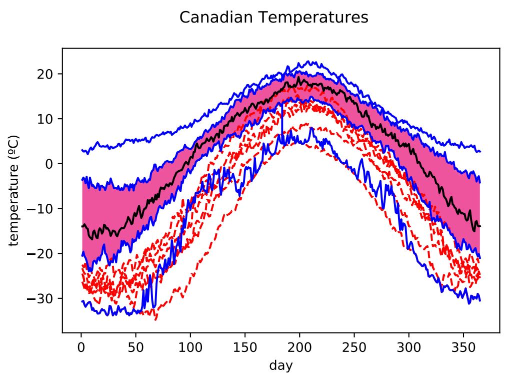


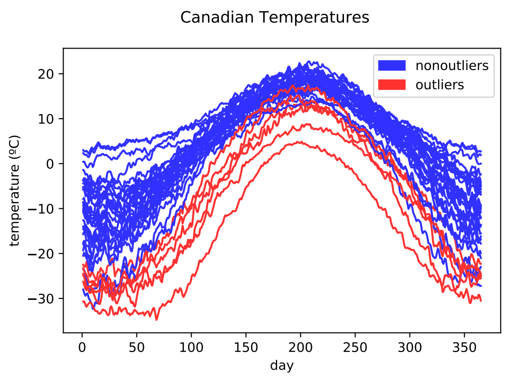
Figure 4.1: The temperatures of the Canadian Weather dataset.

Boxplot

The functional boxplot, Figure 4.2(a), is constructed based on this data. It can be observed the median in black, the central region (where the 50 % of the most central samples reside) in pink and the envelopes and whiskers in blue. The outliers detected, those samples with at least a point outside the outlying envelope, are represented with a red dashed line. In the right plot, Figure 4.2(b), the outliers (in red) are shown with respect to the other samples (in blue). Note their significantly lower values compared to the rest. This is the expected result due to the depth measure used, the modified band depth, which rank the samples according to their magnitude.



(a) Functional boxplot



(b) Outlier detection result

Figure 4.2: The boxplot applied to the Canadian temperatures and the distinction made of outler and regular curves.

If the band depth measure is used and other central regions are included, the result is shown in 4.3. 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.

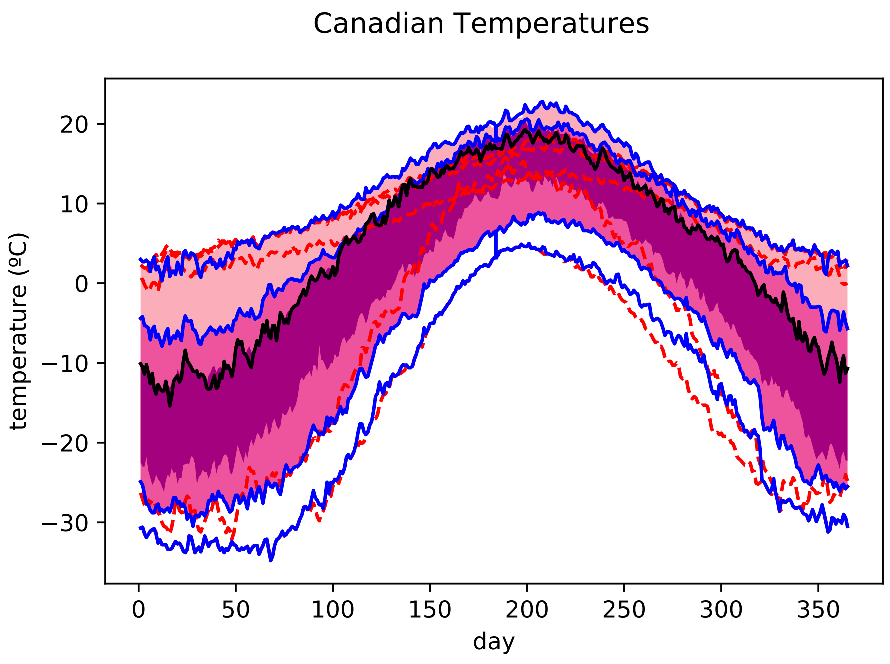
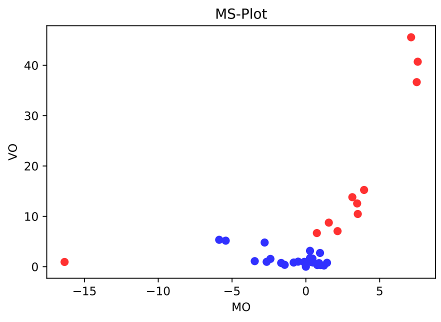


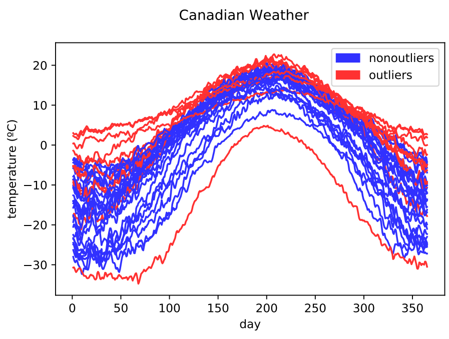
Figure 4.3: Enhanced boxplot of Canadian temperatures.

Magnitude-Shape Plot

Following the previous example, Figure 4.4 shows the magnitude-shape plot applied to the data along with its detected outliers plot. The band depth measure was used. Most of the curves pointed as outliers belong either to the Pacific or Arctic climates, not so common 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.



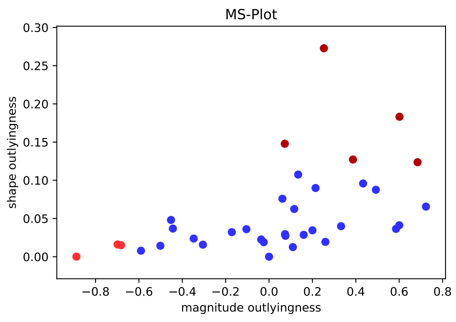
(a) MS-Plot



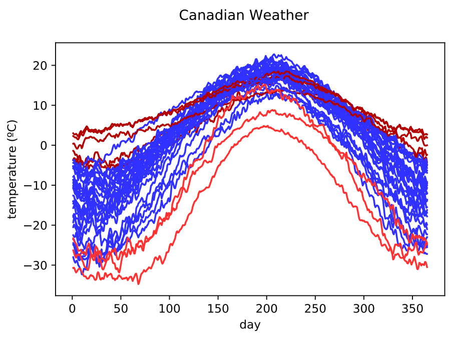
(b) Outlier detection result

Figure 4.4: Magnitude-Shape plot applied to the Canadian temperature data along with its detected outliers plot. The Modified Band depth is used.

In Figure 4.5, the same experiment is carried out but with the Fraiman and Muniz depth measure. The actual MS-Plot does not point out any observation as an outlier. 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. The same is done with the MBD in Figure 4.6.

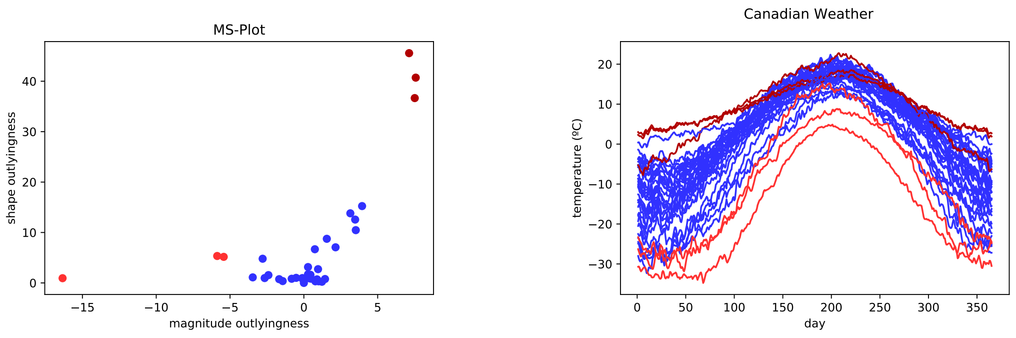


(a) Groupings of the MS-Plot points.



(b) Raw data according to the groupings.

Figure 4.5: Magnitude-Shape plot applied to the Canadian temperature. The Fraiman and Muniz depth is used. The points are divided into three groups.



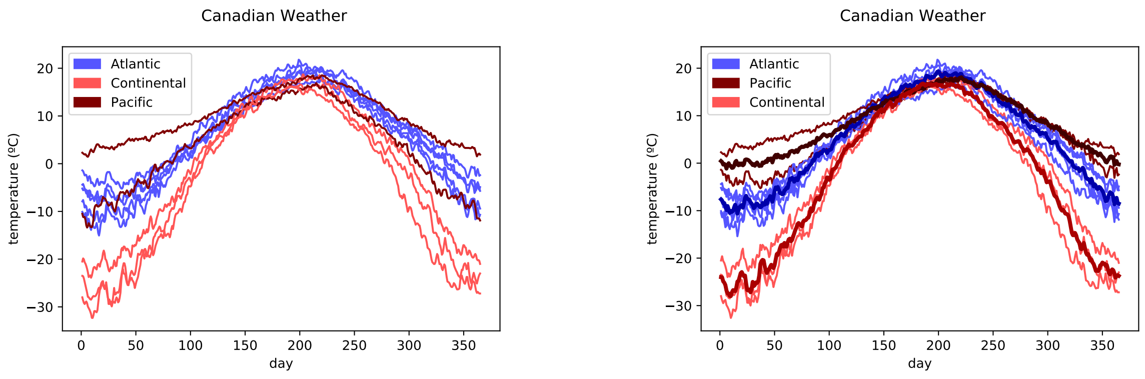
(a) Groupings of the MS-Plot points.

(b) Raw data according to the groupings.

Figure 4.6: Magnitude-Shape plot applied to the Canadian temperature. The MBD is used. The points are divided into three groups.

Clustering Algorithms

For the cluster analysis, the sample to be investigated consists in ten observations picked randomly from the above dataset. Figure 4.7(a) shows the raw data.



(a) Ten random observations of the Canadian Weather dataset.

(b) Raw data according to clusters. Cluster centroids are represented with the same colors and bigger linewidth.

Figure 4.7: Ten random observations of the Canadian Weather dataset and its division into three different clusters.

Note the ten curves chosen belong to three of the four possible climates. The number of clusters is set to three since there are three pronounced distinctions regarding form. Although the three groups are composed of bell-shaped curves, the continental ones are more acute and one of the Pacific climate is considerably shallower. The K-means results are plotted in Figure 4.7(b). The Fuzzy K-Means algorithm produces the same results as in Figure 4.7(b) if assigning to each observation the cluster with maximum degree of membership. The groupings have been made according to shape and magnitude.

Furthermore, two other *ad-hoc* plots have been implemented to better visualize every degree of membership of each observation. One of them appears in Figure 4.8 and is similar to parallel coordina-

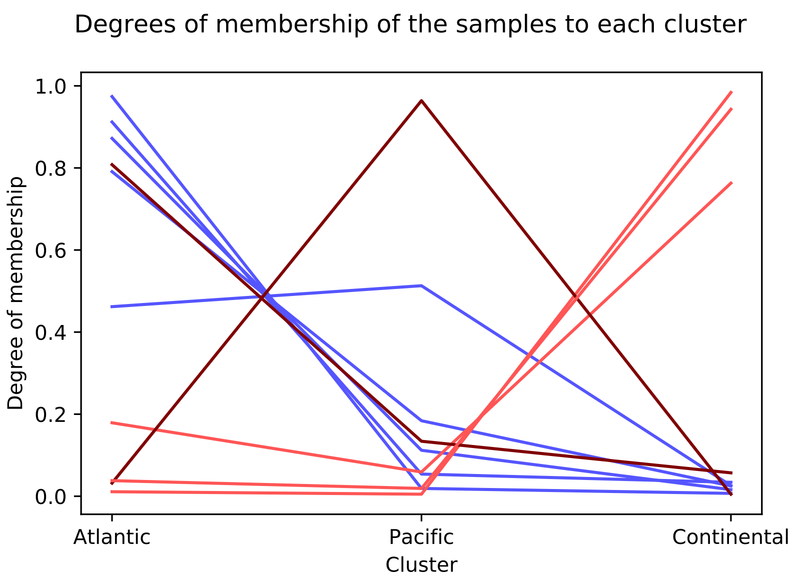
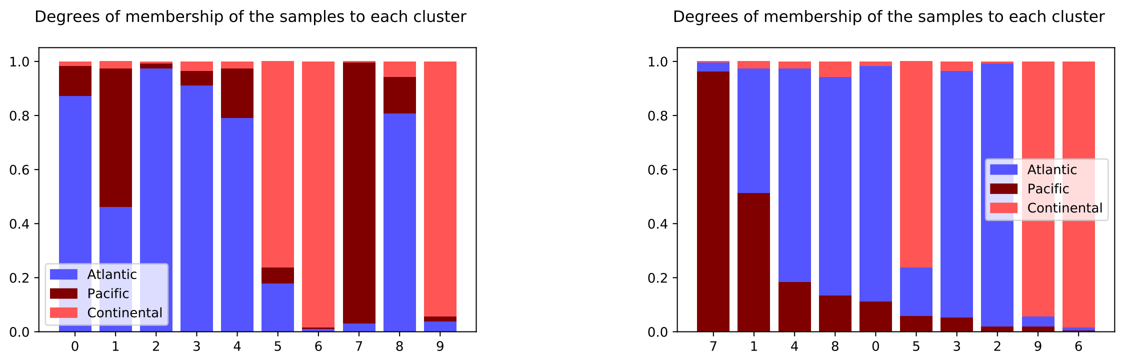


Figure 4.8: Plot implemented to show Fuzzy C-means algorithm results.

tes. The colors are the ones of the first plot (Figure 4.7(a)), dividing the samples by actual climate.

The other one, Figure 4.9, returns a barplot. Each sample is designated with a bar which is filled proportionally to its membership values with the color of each cluster.



(a) Without ordering.

(b) Ordered based to the Pacific climate.

Figure 4.9: Plot implemented to show Fuzzy K-means algorithm results.

5

Future Work and Conclusions

Figure 5.1: scikit-fda logo

Along the document, depth measures and its applications have been explained. Both a theoretic introduction and an implementation have been included. In addition, the final graphic results have been exposed. In regard with this more specialized area, more depth measures could be incorporated. There are many heterogeneous notions of depths which can give rise to different outputs in terms of the characteristic considered. The package could also contain other exploratory tools such as the mentioned outliergram, bagplot or rainbow plot, as well as, an extension of the MS-Plot to a higher dimension. Furthermore, distance measures could be built using the depths defined and more cluster techniques could be added.

In parallel, other two students were working in the project. They focused in preprocessing, which includes smoothing and registration techniques. These techniques approximate functions in order to deal with registered noise and variation in phase and amplitude, respectively. Also, some basic regression methods have been included to model the data. Regarding Figure 3.1 with the expected functionality of the package, more effort must be invested especially in statistical inference such as estimation and hypothesis testing of functional data.

Still the outcome of the project has accomplished the expectations. The final goal of the thesis was to develop a comprehensive Python package for Functional Data Analysis. So firstly, the functionality of the fda package initiated last year had to be expanded. A lot of work has been invested, not only by me but also by the other team members. This implied great collaboration which could be achieved thanks to communication in regular meetings and via GitHub, a very useful tool which helped in the coordination and regulation of the team regarding to code implementation. It also allows the supervision and approval of merge requests and the addressing of issues. Moreover, its integration with Travis CI allows to follow the continuous integration practice. Testing and documentation are also monitorized through GitHub web service.

Not so much along, the first release of the package was delivered under the name of **scikit-fda**. It has a BSD license and the logo is the one in Figure 5.1. The long term goal is to implement novel techniques so that the Python fda package evolves together with the field of Functional Data Analysis.

Bibliography

[1] J. O. Ramsay and B. W. Silverman, *Functional data analysis*. Springer series in statistics, New York: Springer, 2nd. ed., 2005.

[2] F. Ferraty and P. Vieu, *Nonparametric functional data analysis: theory and practice*. Springer series in statistics, New York: Springer, 2006.

[3] J. O. Ramsay and B. W. Silverman, “Functional data analysis - Software.” `http://www.psych.mcgill.ca/misc/fda/software.html`, 2017.

[4] M. Febrero-Bande and M. O. de la Fuente, “Statistical computing in functional data analysis: The R package fda.usc,” *Journal of Statistical Software*, vol. 51, no. 4, 2012.

[5] J. Goldsmith, F. Scheipl, L. Huang, *et al.*, “refund: Regression with functional data.” `https://CRAN.R-project.org/package=refund`, 2016.

[6] N. Tarabelloni *et al.*, “roahd: Robust analysis of high dimensional data.” `https://CRAN.R-project.org/package=roahd`, 2018.

[7] R. J. Hyndman and H. L. Shang, “Rainbow plots, bagplots, and boxplots for functional data,” *Journal of Computational and Graphical Statistics*, vol. 19, no. 1, pp. 29–45, 2010.

[8] F. Yaoand, B. Liu, H. Müller, *et al.*, “PACE: Principal Analysis by Conditional Expectation.” `http://www.stat.ucdavis.edu/PACE/`, 2015.

[9] M. Carbajo-Berrocal, “FDA-PY: desarrollo de un paquete Python para el análisis de datos funcionales,” 2018.

[10] T. Tian, “Functional data analysis in brain imaging studies,” *Frontiers In Psychology*, vol. 1, 2010.

[11] J. W. Tukey, “Mathematics and the picturing of data,” pp. 523–531, Canadian Mathematical Congress.

[12] R. Y. Liu, “On a notion of data depth based on random simplices,” *The Annals of Statistics*, vol. 18, no. 1, pp. 405–414, 1990.

[13] Y. Zuo and R. Serfling, “General notions of statistical depth function,” *Annals of Statistics*, vol. 28, no. 2, pp. 461–482, 2000.

[14] R. Fraiman and G. Muniz, “Trimmed means for functional data,” *Test*, vol. 10, no. 2, pp. 419–440, 2001.

[15] S. López-Pintado and J. Romo, “On the concept of depth for functional data,” *Journal of the American Statistical Association*, vol. 104, no. 486, pp. 718–734, 2009.

[16] F. Ieva and A. M. Paganoni, “Depth measures for multivariate functional data,” *Communications in Statistics - Theory and Methods*, vol. 42, no. 7, pp. 1265–1276, 2013.

[17] Y. Sun and M. G. Genton, “Functional boxplots,” *Journal of Computational and Graphical Statistics*, vol. 20, no. 2, pp. 316–334, 2011.

[18] A. Arribas-Gil and J. Romo, “Shape outlier detection and visualization for functional data: The outliergram,” *Biostatistics*, vol. 15, no. 4, pp. 603–619, 2014.

[19] M. G. Genton, C. Johnson, K. Potter, G. Stenchikov, and Y. Sun, “Surface boxplots,” *Stat*, vol. 3, no. 1, pp. 1–11, 2014.

[20] W. Dai and M. G. Genton, “Multivariate functional data visualization and outlier detection,” *Journal of Computational and Graphical Statistics*, vol. 27, no. 4, pp. 923–934, 2018.

[21] W. Dai and M. G. Genton, “An outlyingness matrix for multivariate functional data classification,” *Statistica Sinica*, vol. 28, no. 4, pp. 2435–2454, 2018.

[22] W. Dai and M. G. Genton, “Directional outlyingness for multivariate functional data,” *Computational Statistics and Data Analysis*, vol. 131, pp. 50–65, 2019.

[23] J. Hardin and D. M. Rocke, “The distribution of robust distances,” *Journal of Computational and Graphical Statistics*, vol. 14, no. 4, pp. 928–946, 2005.

[24] R. Flores, R. Lillo, and J. Romo, “Homogeneity test for functional data,” *Journal of Applied Statistics*, vol. 45, no. 5, pp. 868–883, 2018.

[25] A. Baíllo, A. Cuevas, and R. Fraiman, “Classification methods for functional data,” *The Oxford Handbook of Functional Data Analysis*, pp. 259–297, 01 2011.

[26] Z. Cebeci and F. Yıldız, “Comparison of k-means and fuzzy c-means algorithms on different cluster structures,” *Journal of Agricultural Informatics*, vol. 6, pp. 13–23, 10 2015.

[27] C. Abraham, P. A. Cornillon, E. Matzner-Løber, and N. Molinari, “Unsupervised curve clustering using b-splines,” *Scandinavian Journal of Statistics*, vol. 30, no. 3, pp. 581–595, 2003.

[28] A. Justel and M. Svarc, “Sequential clustering for functional data,” 2016.

[29] C. Ramos-Carreño, “Scikit-fda: A Python package for Functional Data Analysis.” III International Workshop on Advances in Functional Data Analysis, 05 2019.

[30] Y. Sun, M. G. Genton, and D. W. Nychka, “Exact fast computation of band depth for large functional datasets: How quickly can one million curves be ranked?,” *Stat*, vol. 1, no. 1, pp. 68–74, 2012.

Appendices

A

Documentation

Note

Click here to download the full example code

Boxplot

Shows the use of the functional Boxplot applied to the Canadian Weather dataset.

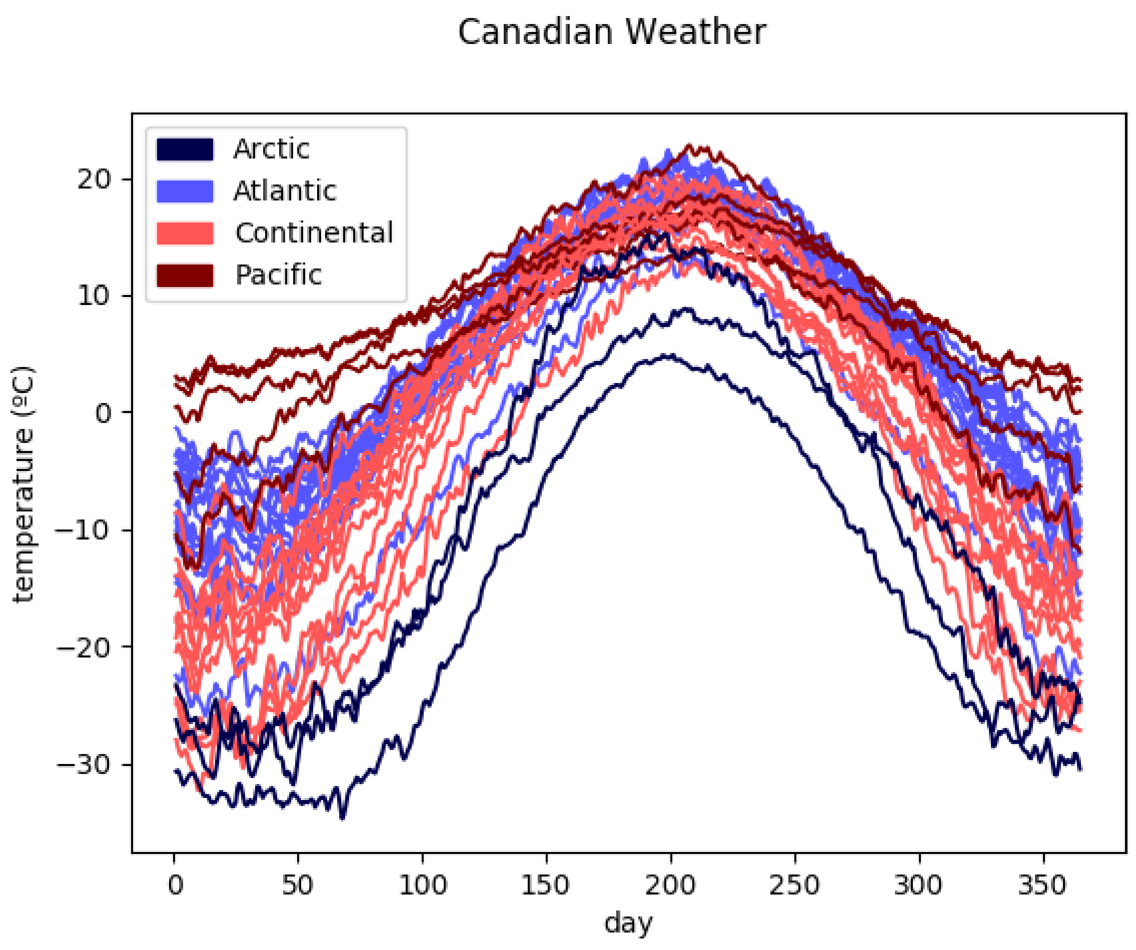
```
# Author: Amanda Hernando Bernabé
# License: MIT

# sphinx_gallery_thumbnail_number = 2

from skfda import datasets
from skfda import FDataGrid
from skfda.exploratory.depth import band_depth, fraiman_muniz_depth
import matplotlib.pyplot as plt
from skfda.exploratory.visualization.boxplot import Boxplot
import numpy as np
```

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 another FDataGrid is constructed with the desired values.

```
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = FDataGrid(data_matrix=fd.data_matrix[:, :, 0],
                            sample_points=fd.sample_points,
                            dataset_label=fd.dataset_label,
                            axes_labels=fd.axes_labels[0:2])
```

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.

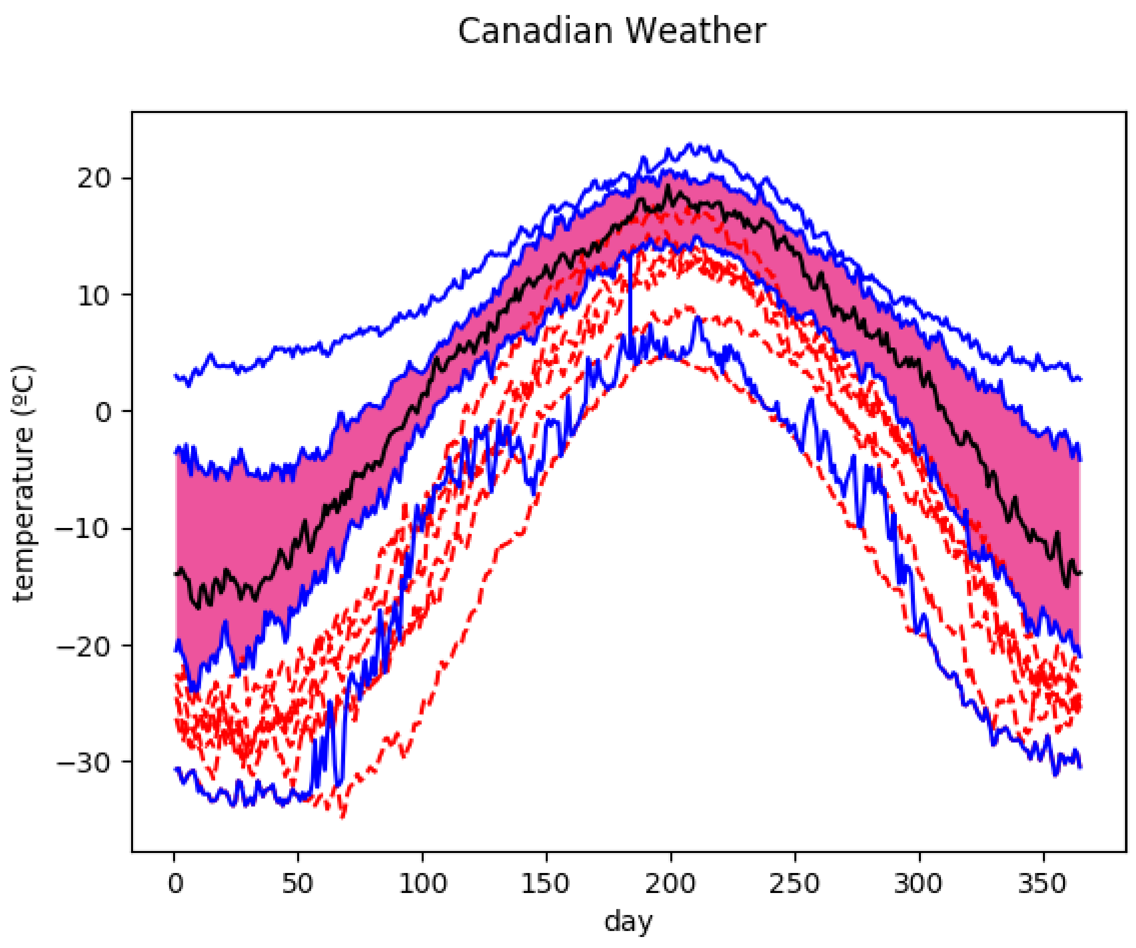
```
# Each climate is assigned a color. Defaults to grey.
colormap = plt.cm.get_cmap('seismic')
label_names = dataset["target_names"]
nlabels = len(label_names)
label_colors = colormap(np.arange(nlabels) / (nlabels - 1))

plt.figure()
fd_temperatures.plot(sample_labels=dataset["target"], label_colors=label_colors,
                     label_names=label_names)
```



We instantiate a `functional 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.

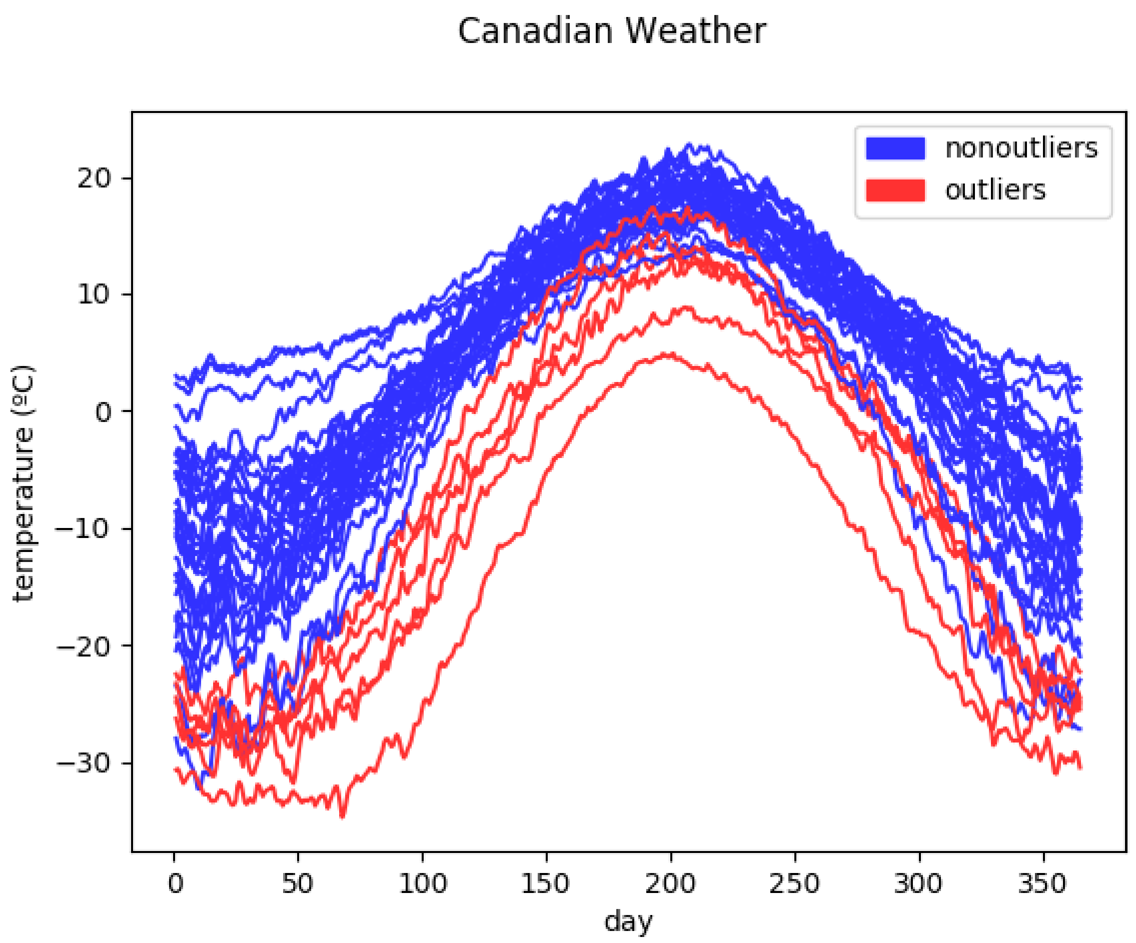
```
fdBoxplot = Boxplot(fd_temperatures)
fdBoxplot.show_full_outliers = True

plt.figure()
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.

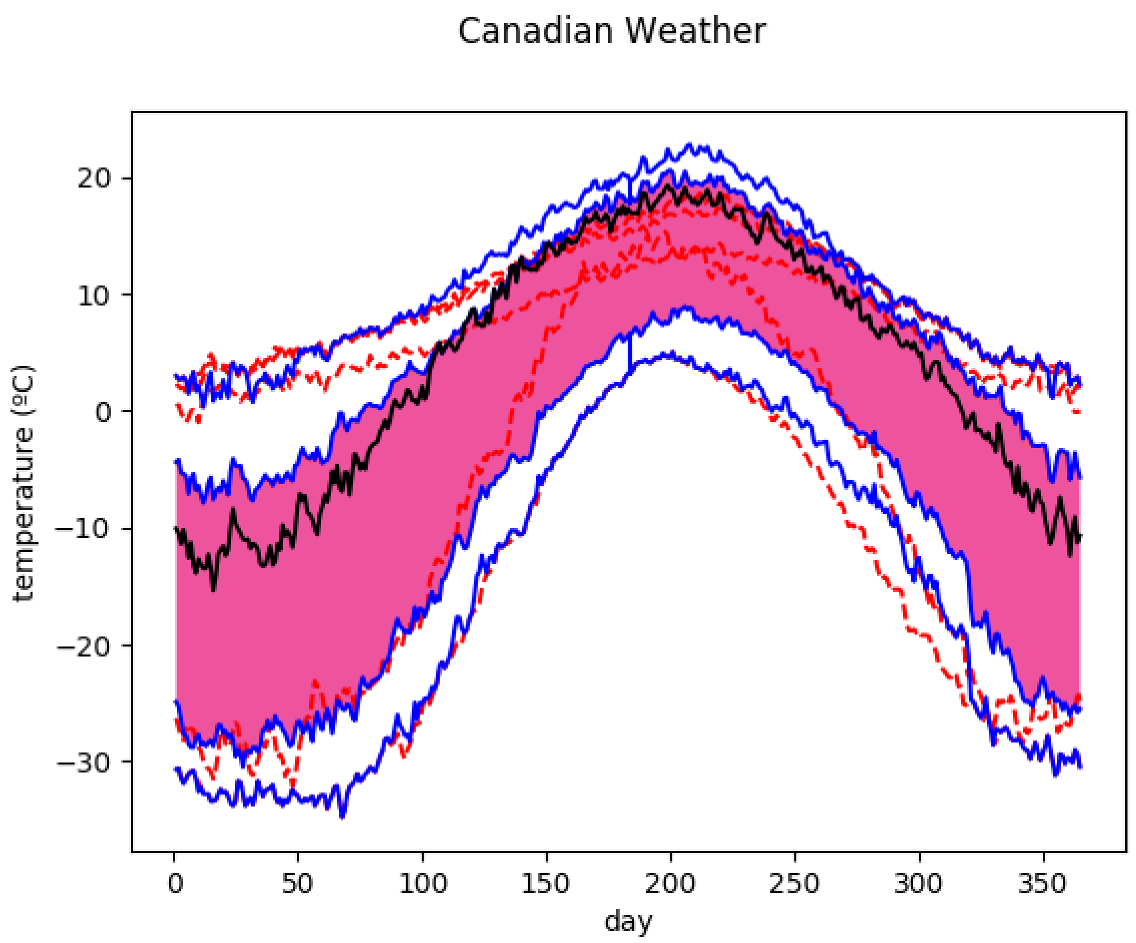
```
color = 0.3
outliercol = 0.7

plt.figure()
fd_temperatures.plot(sample_labels=fdBoxplot.outliers[0].astype(int),
                     label_colors=colormap([color, outliercol]),
                     label_names=["nonoutliers", "outliers"])
```



The curves pointed as outliers are are those curves with significantly lower values to the rest. This is the expected result due to the depth measure used, the `modified band depth` which rank the samples according to their magnitude.

The `functional boxplot object` admits any depth measure defined or customized by the user. Now the call is done with the `band depth measure` 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.

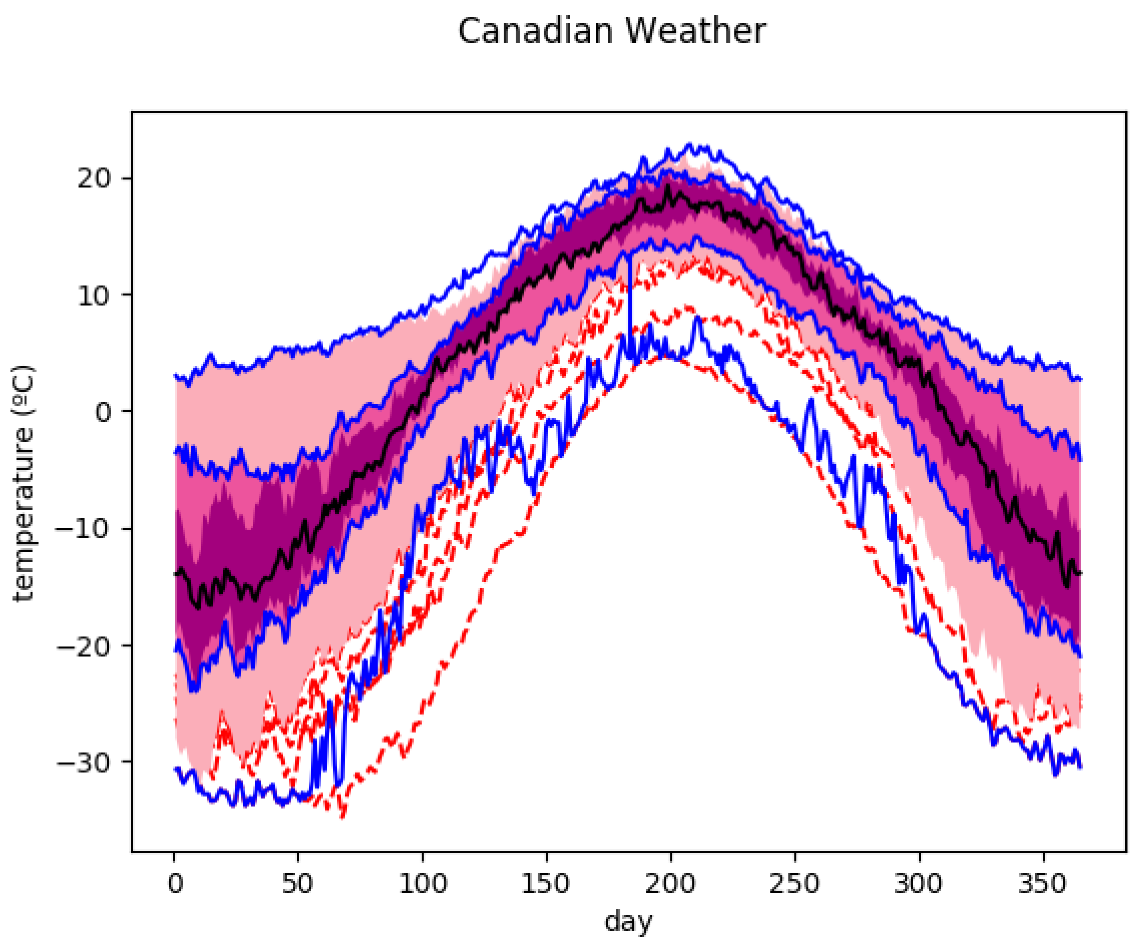
```
fdBoxplot = Boxplot(fd_temperatures, method=band_depth, factor = 0.4)
fdBoxplot.show_full_outliers = True

plt.figure()
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 `Fraiman and Muniz depth measure` is used and the 25% and 75% central regions are specified.

```
fdBoxplot = Boxplot(fd_temperatures,  method=fraiman_muniz_depth,
                    prob = [0.75, 0.5, 0.25])
plt.figure()
fdBoxplot.plot()
```



The above two lines could be replaced just by fdBoxplot since the default representation of the `boxplot object` is the image of the plot. However, due to generation of this notebook it does not show the image and that is why the plot method is called.

Total running time of the script: (0 minutes 2.578 seconds)

Download Python source code: plot_boxplot.py

Download Jupyter notebook: plot_boxplot.ipynb

Note

Click here to download the full example code

Surface Boxplot

Shows the use of the surface boxplot, which is a generalization of the functional boxplot for FDataGrid whose domain dimension is 2.

```
# Author: Amanda Hernando Bernabé
# License: MIT

# sphinx_gallery_thumbnail_number = 3

import numpy as np
import matplotlib.pyplot as plt
from skfda import FDataGrid
from skfda.exploratory.visualization.boxplot import SurfaceBoxplot, Boxplot
from skfda.datasets import make_sinusoidal_process, make_gaussian_process
```

In order to instantiate a `surface boxplot object`, a functional data object with bidimensional domain must be generated. In this example, a FDataGrid representing a function $f : \mathbb{R}^2 \longmapsto \mathbb{R}^2$ is constructed to show also the support of a multivariate dimensional image. The first dimension of the image contains sinusoidal processes and the second dimension, gaussian ones.

First, the values are generated for each dimension with a function $f : \mathbb{R} \longmapsto \mathbb{R}$ implemented in the `make_sinusoidal_process method` and in the `make_gaussian_process method`, respectively. Those functions return FDataGrid objects whose 'data_matrix' store the values needed.

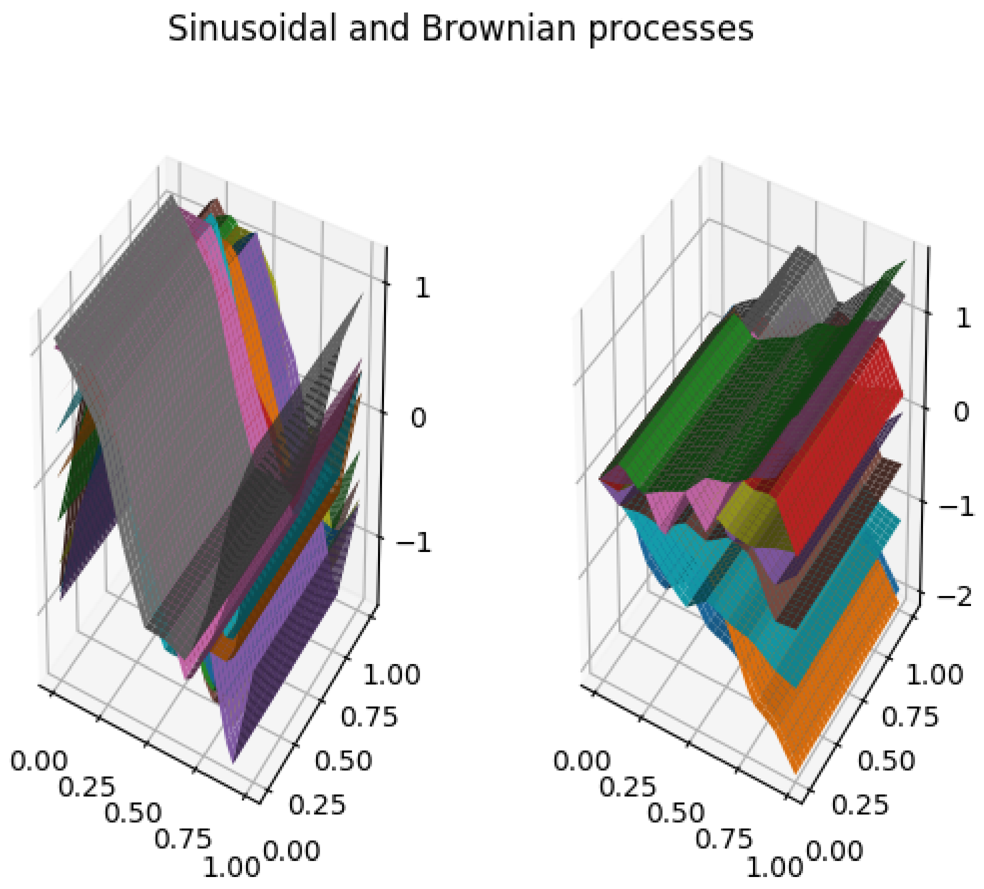
```
n_samples = 10
n_features = 10

fd1 = make_sinusoidal_process(n_samples = n_samples, n_features=n_features,
                              random_state=5)
fd1.dataset_label = "Sinusoidal process"
fd2 = make_gaussian_process(n_samples = n_samples, n_features=n_features,
                              random_state=1)
fd2.dataset_label = "Brownian process"
```

After, those values generated for one dimension on the domain are propagated along another dimension, obtaining a three-dimensional matrix or cube (two-dimensional domain and one-dimensional image). This is done with both data matrices from the above FDataGrids.

```
cube1 = np.repeat(fd1.data_matrix, n_features).reshape(
    (n_samples, n_features, n_features))
cube2 = np.repeat(fd2.data_matrix, n_features).reshape(
    (n_samples, n_features, n_features))
```

Finally, both three-dimensional matrices are merged together and the FDataGrid desired is obtained. The data is plotted.

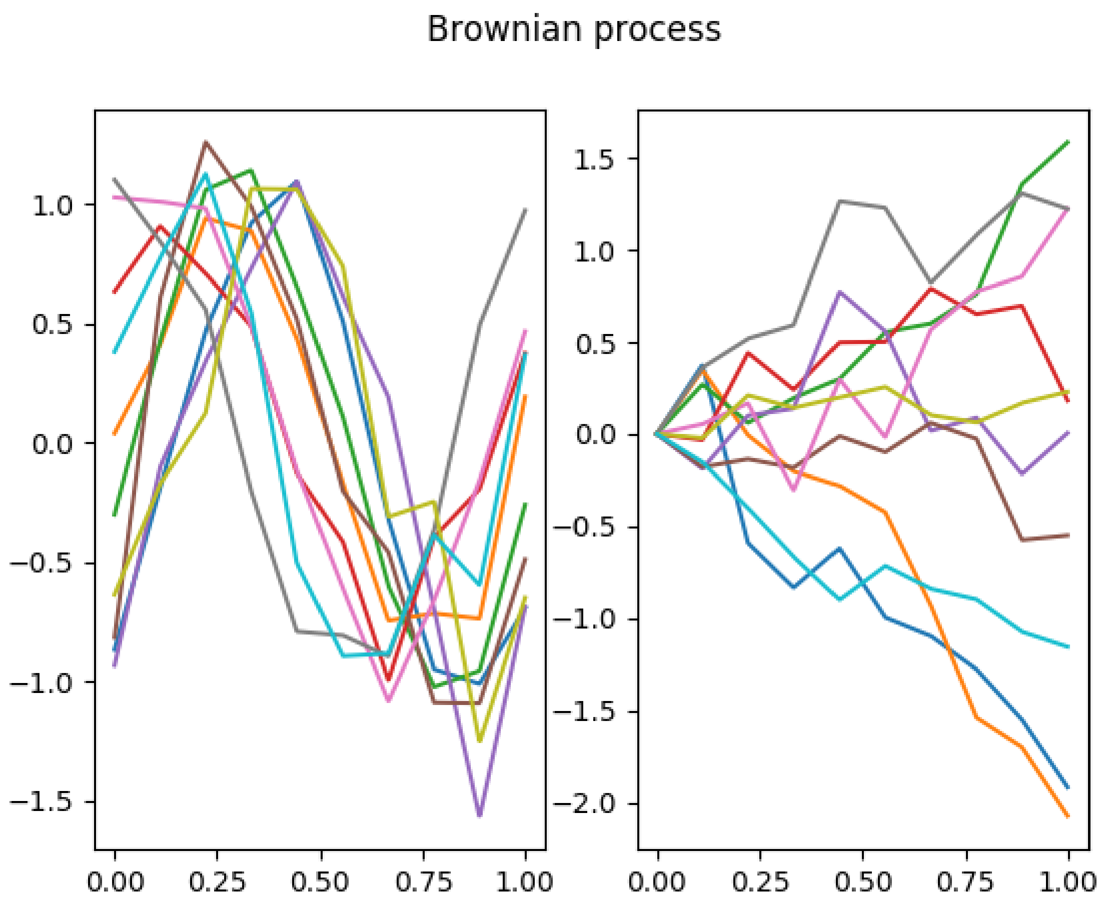
```
cube_2 = np.empty((n_samples, n_features, n_features, 2))
cube_2[:, :, :, 0] = cube1
cube_2[:, :, :, 1] = cube2

fd_2 = FDataGrid(data_matrix=cube_2, sample_points=np.tile(fd1.sample_points, (2,1)),
                 dataset_label = "Sinusoidal and Brownian processes")

plt.figure()
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.

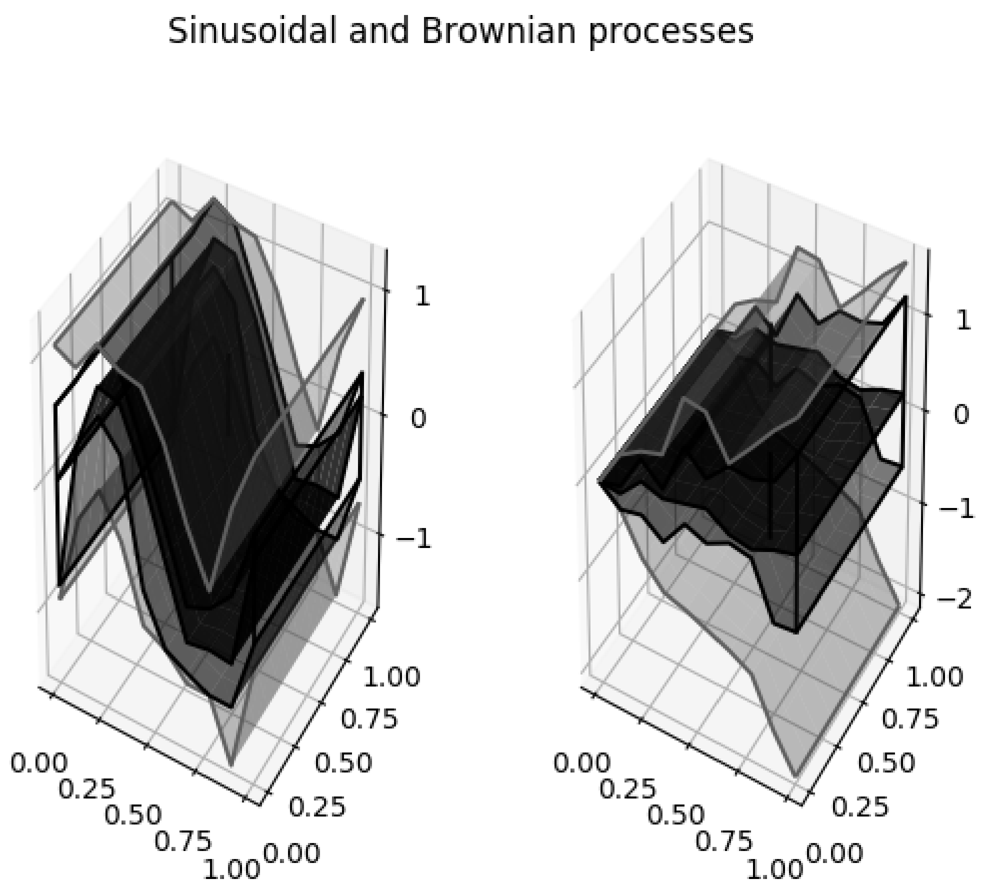
For this reason, the profiles of the surfaces, which are contained in the first two generated functional data objects, are plotted below, to help to visualize the data.

```
fig, ax = plt.subplots(1,2)
fd1.plot(ax=[ax[0]])
fd2.plot(ax=[ax[1]])
```



To terminate the example, the instantiation of the SurfaceBoxplot object is made, showing the surface boxplot which corresponds to our FDataGrid representing a function $f : \mathbb{R}^2 \longmapsto \mathbb{R}^2$ with a sinusoidal process in the first dimension of the image and a gaussian one in the second one.

```
surfaceBoxplot = SurfaceBoxplot(fd_2)
plt.figure()
surfaceBoxplot.plot()
```



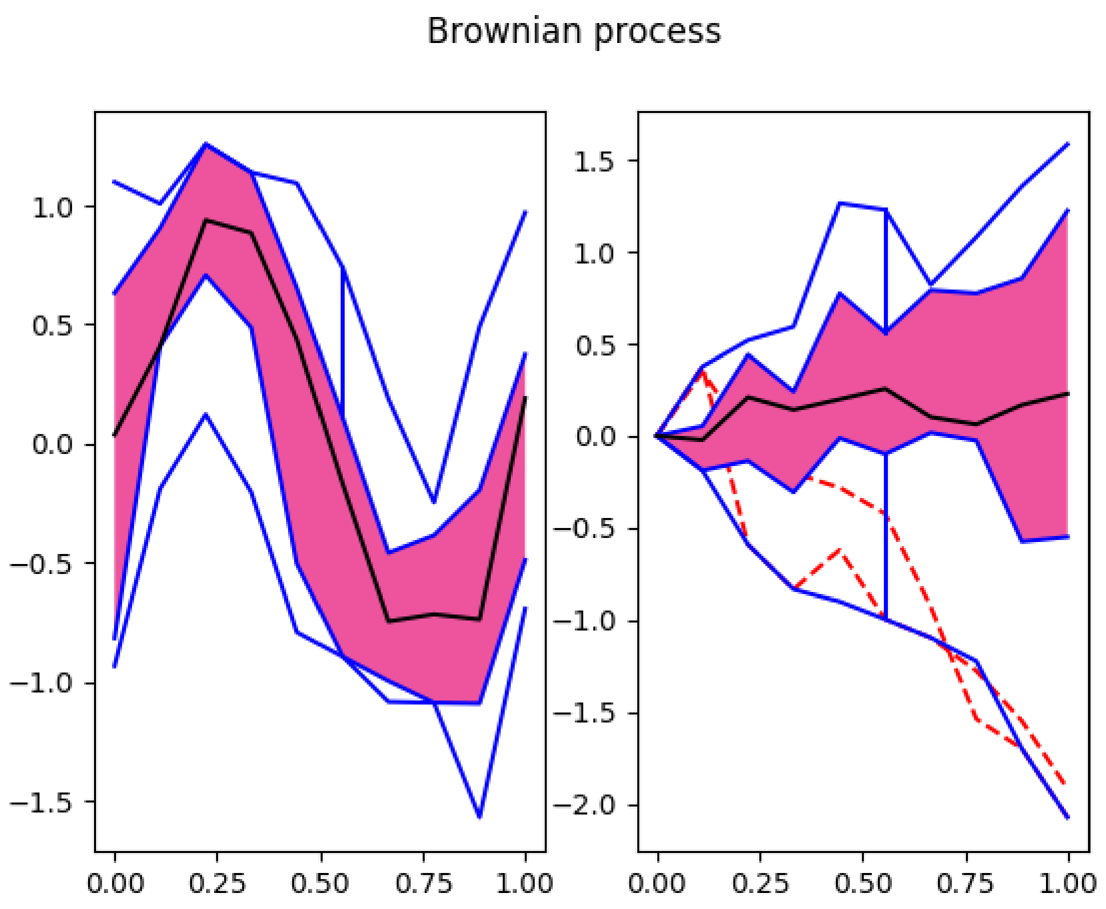
The default representation of the object its the graph.

```
surfaceBoxplot
```

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 correponding profiles, we can obtain also the functional boxplot for one-dimensional data with the `fdboxplot function` passing as arguments the first two FdataGrid objects. The profile of the surface boxplot is obtained.

```
fig, ax = plt.subplots(1,2)
boxplot1 = Boxplot(fd1)
boxplot1.plot(ax=[ax[0]])
boxplot2 = Boxplot(fd2)
boxplot2.plot(ax=[ax[1]])
```



Total running time of the script: (0 minutes 4.461 seconds)

Download Python source code: plot_surface_boxplot.py

Download Jupyter notebook: plot_surface_boxplot.ipynb

Note

Click here to download the full example code

Magnitude-Shape Plot

Shows the use of the MS-Plot applied to the Canadian Weather dataset.

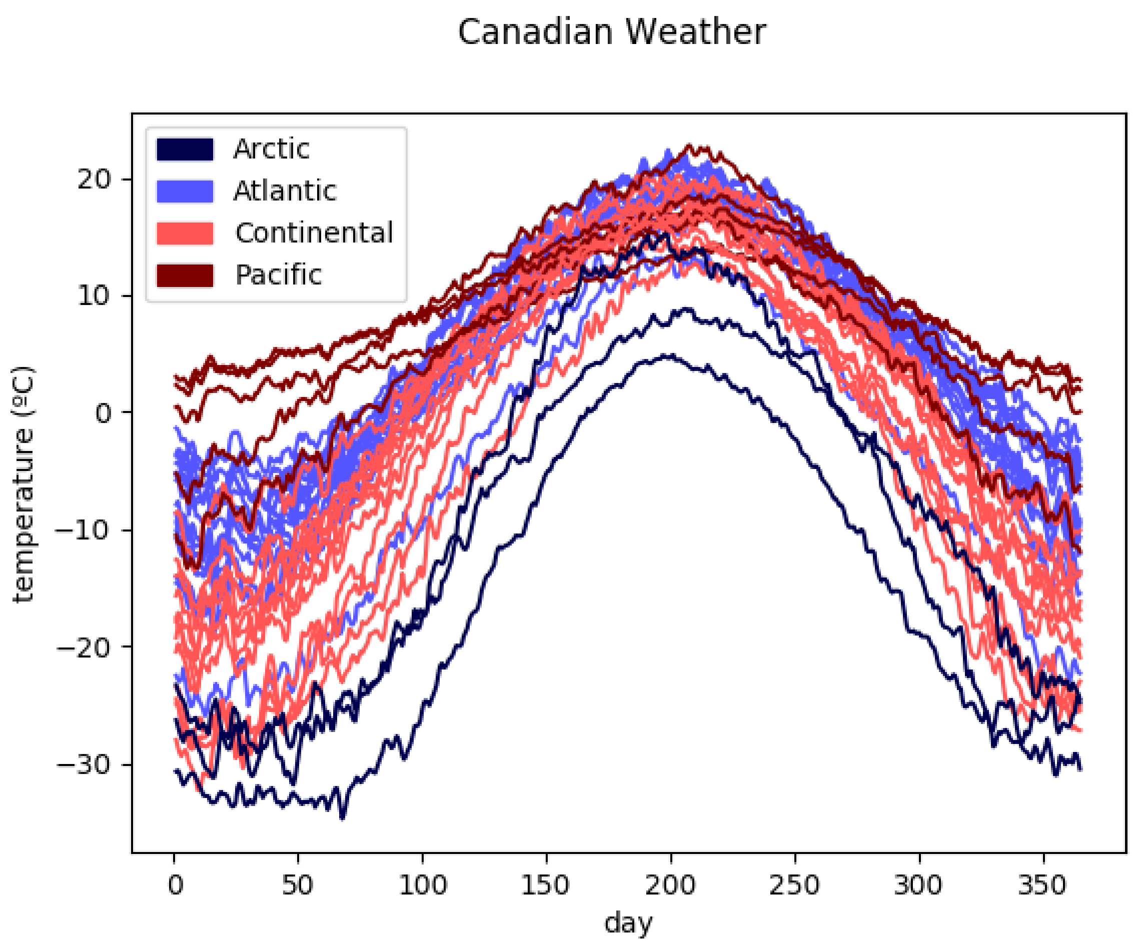
```
# Author: Amanda Hernando Bernabé
# License: MIT

# sphinx_gallery_thumbnail_number = 2

from skfda import datasets
from skfda import FDataGrid
from skfda.exploratory.depth import fraiman_muniz_depth
from skfda.exploratory.visualization.magnitude_shape_plot import (
    MagnitudeShapePlot)
import matplotlib.pyplot as plt
import numpy as np
```

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 another FDataGrid is constructed with the desired values.

```
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = FDataGrid(data_matrix=fd.data_matrix[:, :, 0],
                            sample_points=fd.sample_points,
                            dataset_label=fd.dataset_label,
                            axes_labels=fd.axes_labels[0:2])
```

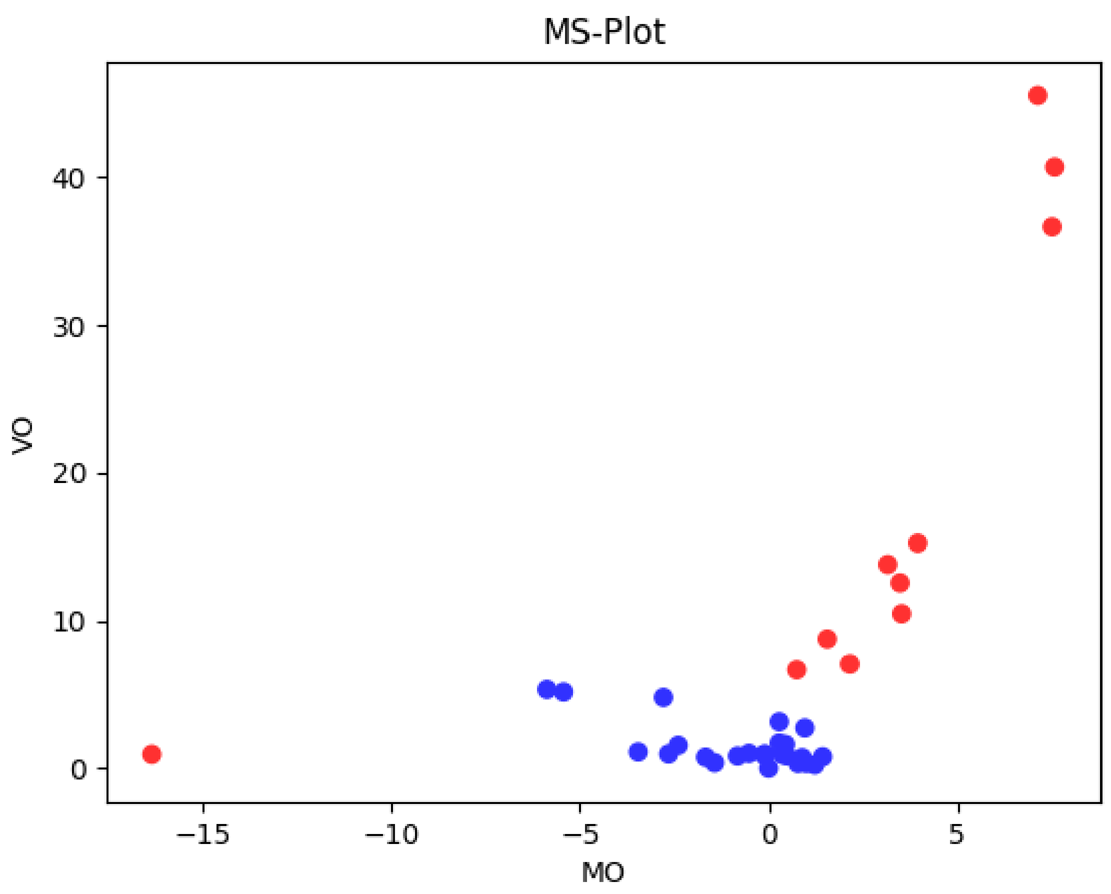
The data is plotted to show the curves we are working with. They are divided according to the target. In this case, it includes the different climates to which the weather stations belong to.

```
# 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))

plt.figure()
fd_temperatures.plot(sample_labels=dataset["target"], label_colors=label_colors,
                     label_names=label_names)
```



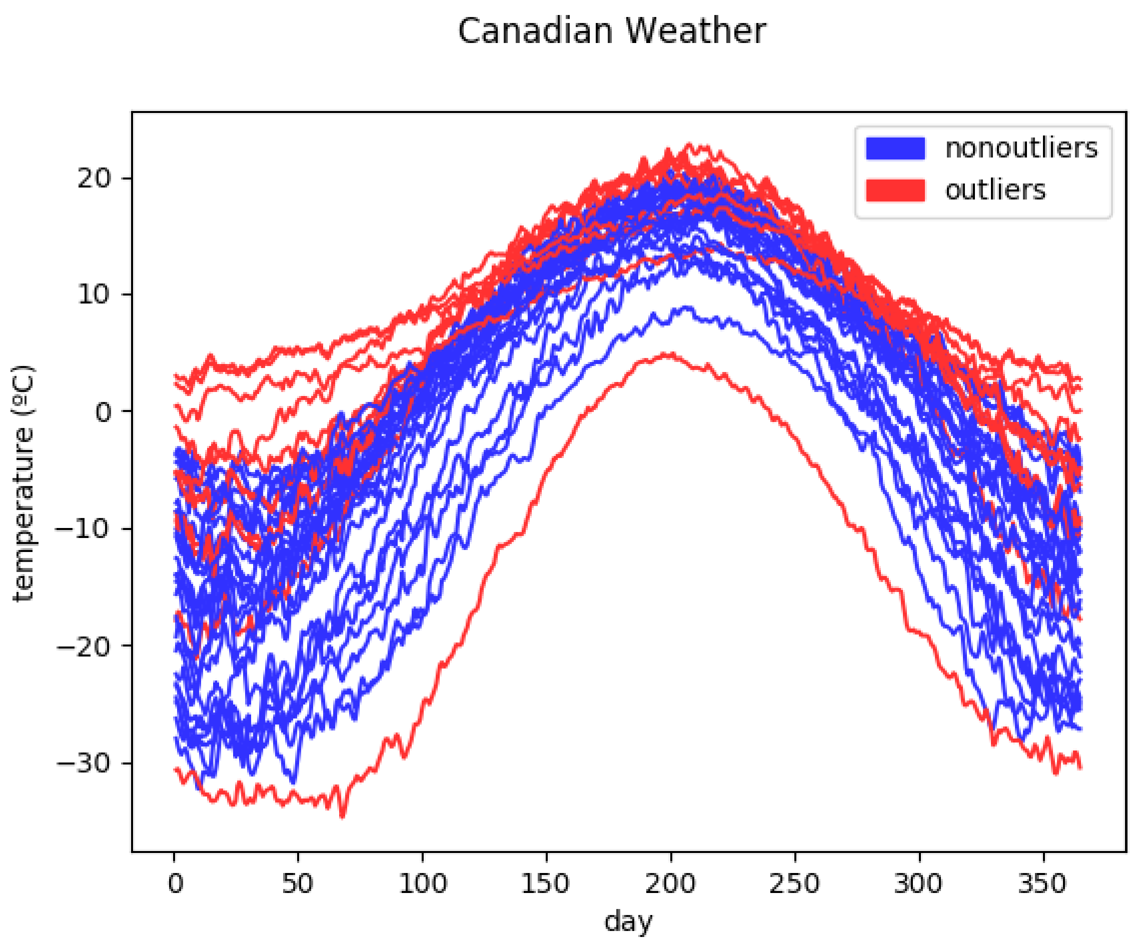
The MS-Plot is generated. In order to show the results, the `plot method` is used. Note that the colors have been specified before to distinguish between outliers or not. In particular the tones of the default colormap, (which is 'seismic' and can be customized), are assigned.

```
msplot = MagnitudeShapePlot(fdatagrid=fd_temperatures)

color = 0.3
outliercol = 0.7

plt.figure()
msplot.color = color
msplot.outliercol = outliercol
msplot.plot()
```



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.

```
plt.figure()
fd_temperatures.plot(sample_labels=msplot.outliers,
                     label_colors=msplot.colormap([color, outliercol]),
                     label_names = ['nonoutliers', 'outliers'])
```

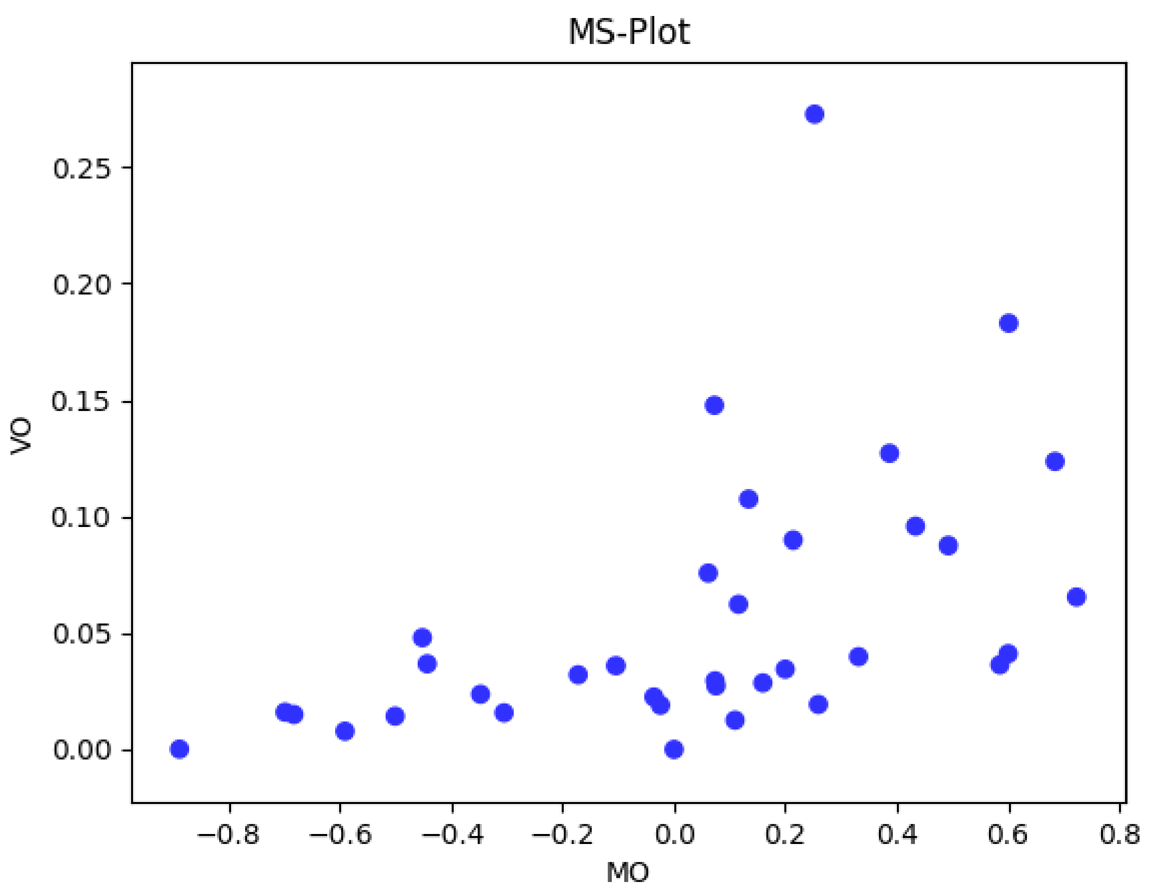


We can observe that most of the curves pointed as outliers belong either to the Pacific or Arctic climates which are not the common ones found in Canada. The Pacific temperatures are much smoother and the Arctic ones much lower, differing from the rest in shape and magnitude respectively.

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.

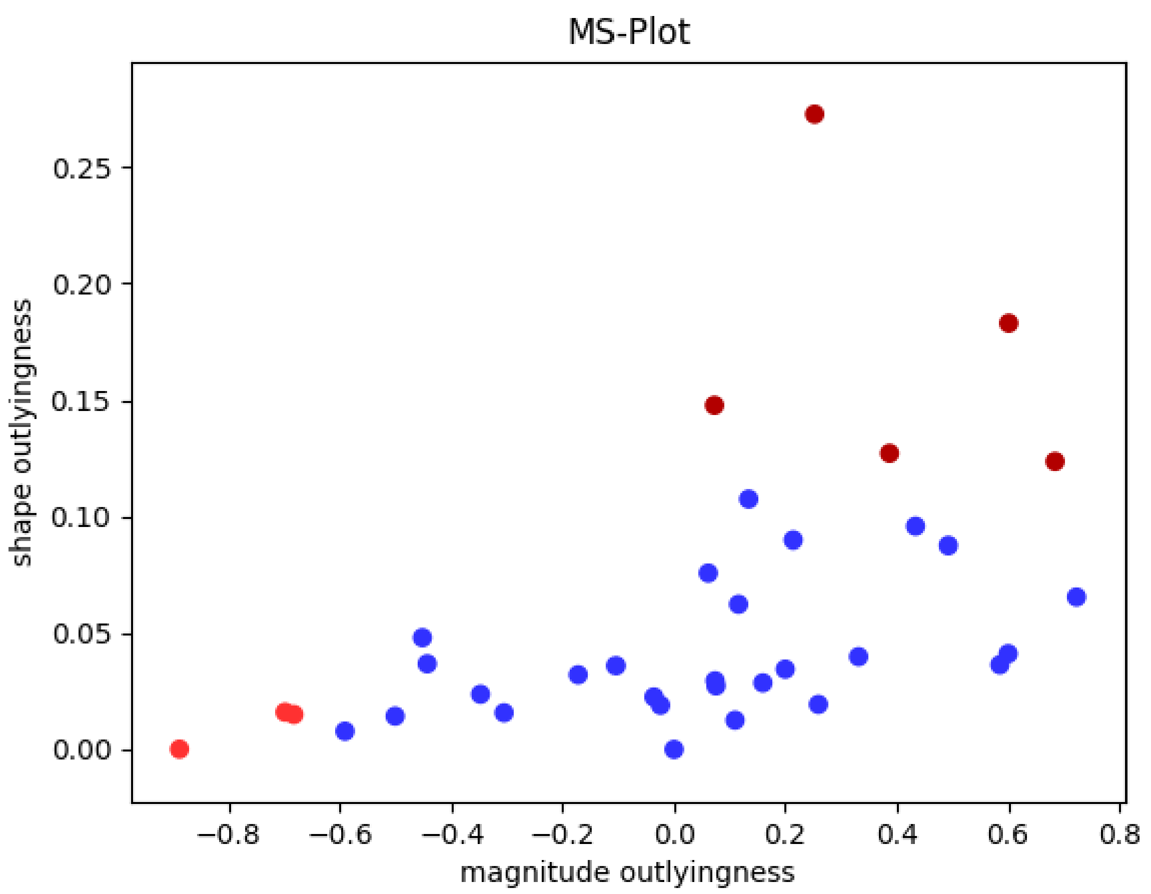
Now we use the `Fraiman and Muniz depth measure` in the MS-Plot.

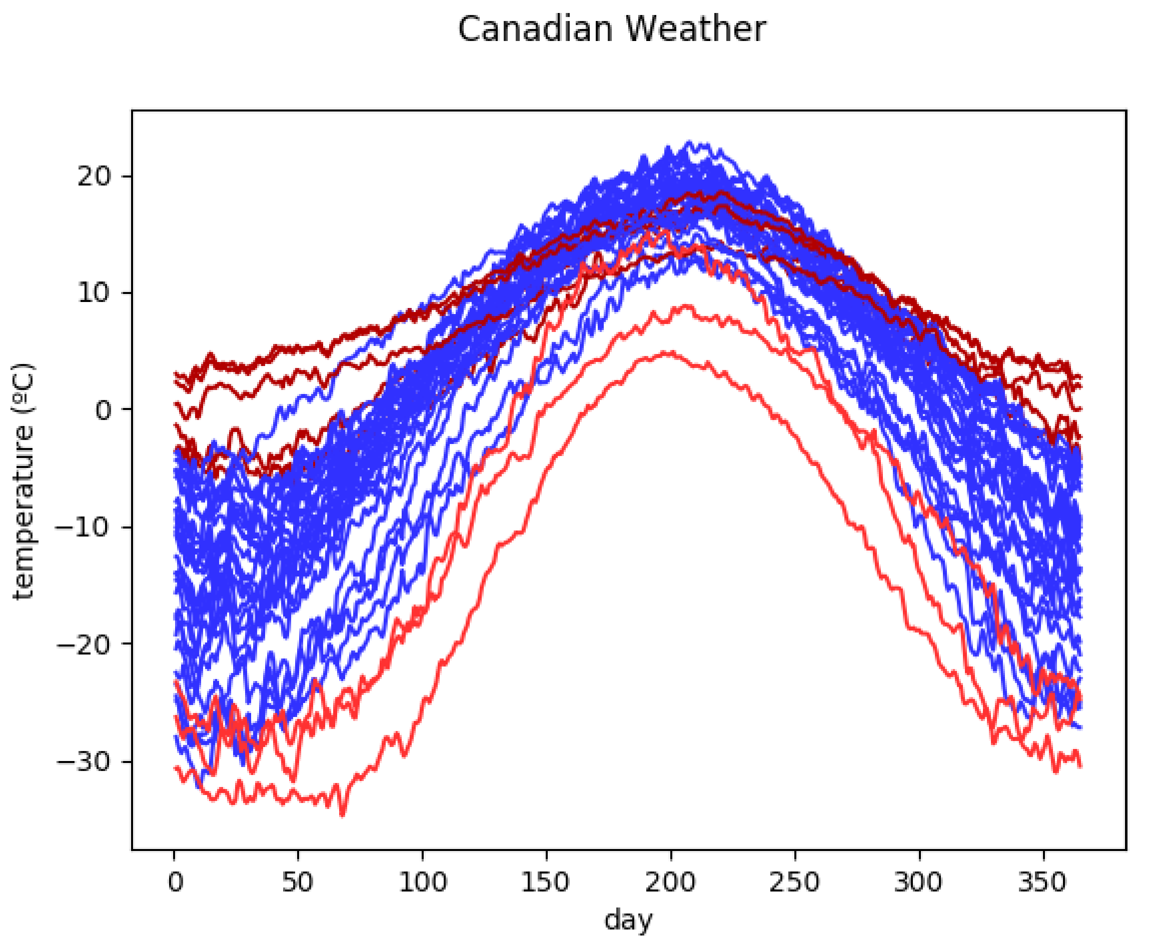
```
msplot = MagnitudeShapePlot(fdatagrid=fd_temperatures,
                            depth_method = fraiman_muniz_depth)

plt.figure()
msplot.color = color
msplot.outliercol = outliercol
msplot.plot()
```



We can observe that 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.

```
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

plt.figure()
plt.scatter(msplot.points[:, 0], msplot.points[:, 1], c=colormap(colors))
plt.title("MS-Plot")
plt.xlabel("magnitude outlyingness")
plt.ylabel("shape outlyingness")

labels = np.copy(msplot.outliers)
labels[group1] = 1
labels[group2] = 2

plt.figure()
fd_temperatures.plot(sample_labels=labels,
                     label_colors=colormap([color, outliercol, 0.9]))
```





Total running time of the script: (0 minutes 3.186 seconds)

Download Python source code: plot_magnitude_shape.py

Download Jupyter notebook: plot_magnitude_shape.ipynb

Note

Click here to download the full example code

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.

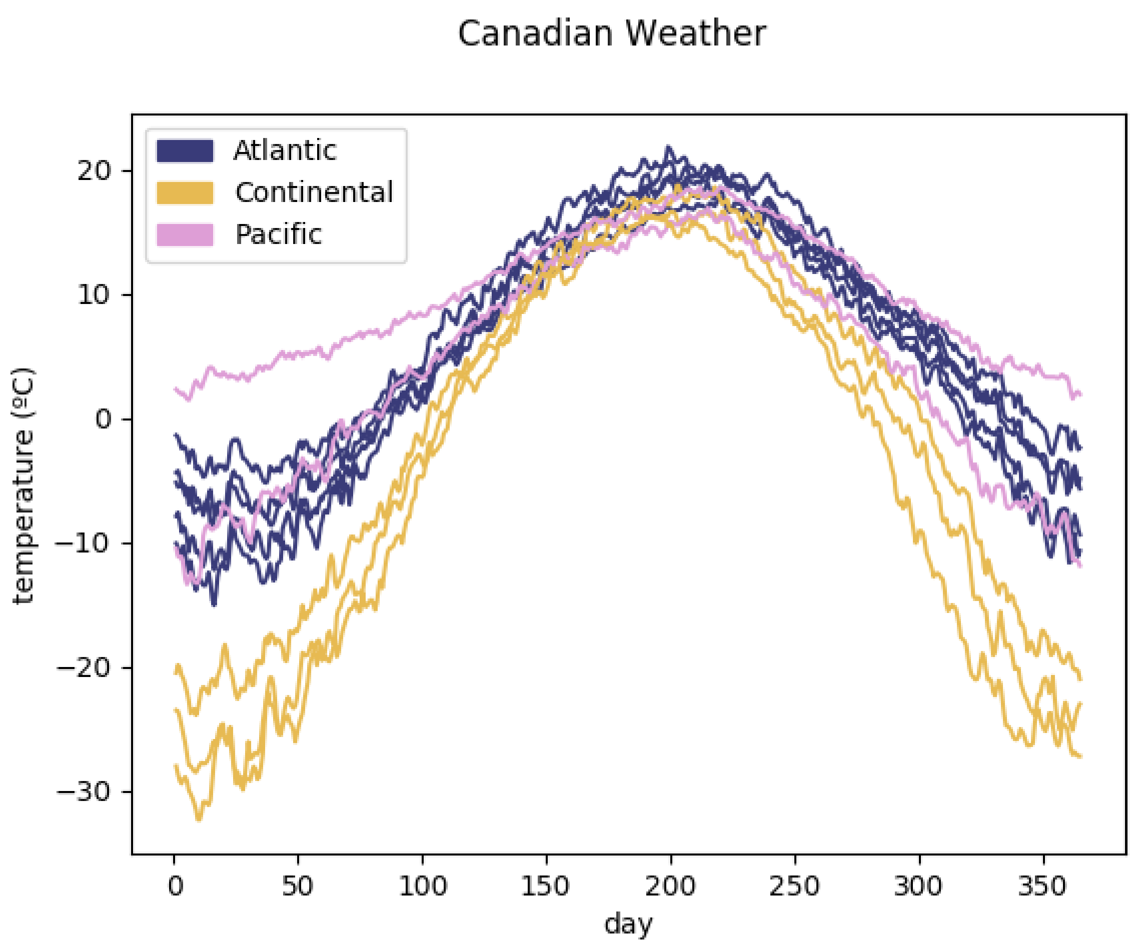
```
# Author: Amanda Hernando Bernabé
# License: MIT

# sphinx_gallery_thumbnail_number = 6

from skfda import datasets
from skfda.representation.grid import FDataGrid
from skfda.ml.clustering.base_kmeans import KMeans
from skfda.exploratory.visualization.clustering_plots import *
```

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 another FDataGrid is constructed with the desired values.

```
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = FDataGrid(data_matrix=fd.data_matrix[:, :, 0],
                            sample_points=fd.sample_points,
                            dataset_label=fd.dataset_label,
                            axes_labels=fd.axes_labels[0:2])

# 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.

```
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))

plt.figure()
fd.plot(sample_labels=indexer, label_colors=climate_colors, label_names=climates)
```



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 obatain always the same result for the example.

```
n_clusters = n_climates
seed = 2
```

First, the class `K-Means` is instantiated with the desired. parameters. Its `fit method` is called , resulting in the calculation of several attributes which include among others, the the number of cluster each sample belongs to (labels), and the centroids of each cluster. The labels are obtaiined calling the method `predict`

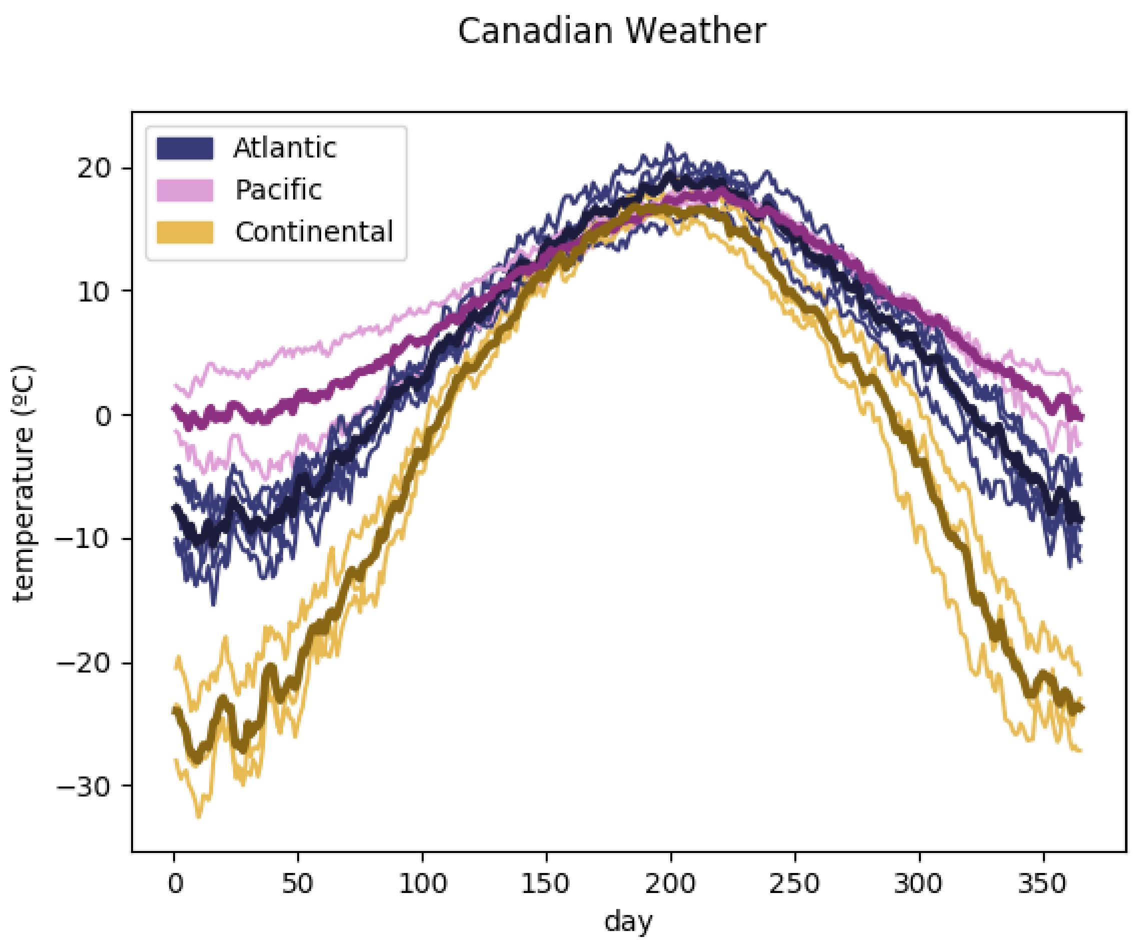
```
kmeans = KMeans(n_clusters=n_clusters, random_state=seed)
kmeans.fit(fd)
print(kmeans.predict(fd))
```

Out:

```
[0 1 0 0 0 2 2 1 0 2]
```

To see the information in a graphic way, the method `plot_clusters` can be used found in the visualization directory.

```
# 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 `FuzzyKMeans`. 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.

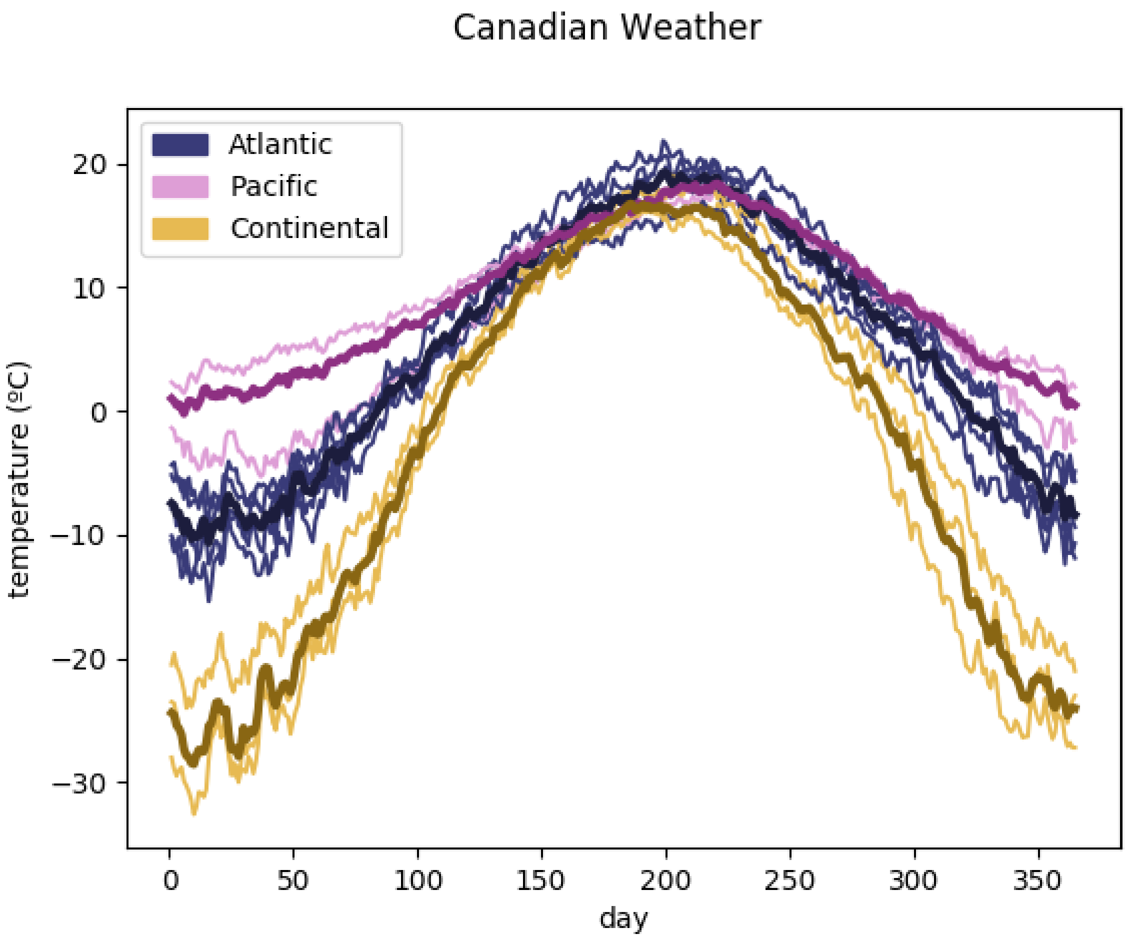
```
fuzzy_kmeans = FuzzyKMeans(n_clusters=n_clusters, random_state=seed)
fuzzy_kmeans.fit(fd)
print(fuzzy_kmeans.predict(fd))
```

Out:

```
[[0.872 0.112 0.016]
 [0.462 0.513 0.026]
 [0.974 0.019 0.007]
 [0.912 0.054 0.034]
 [0.791 0.184 0.025]
 [0.179 0.059 0.763]
 [0.011 0.005 0.984]
 [0.032 0.964 0.005]
 [0.808 0.134 0.057]
 [0.038 0.019 0.943]]
```

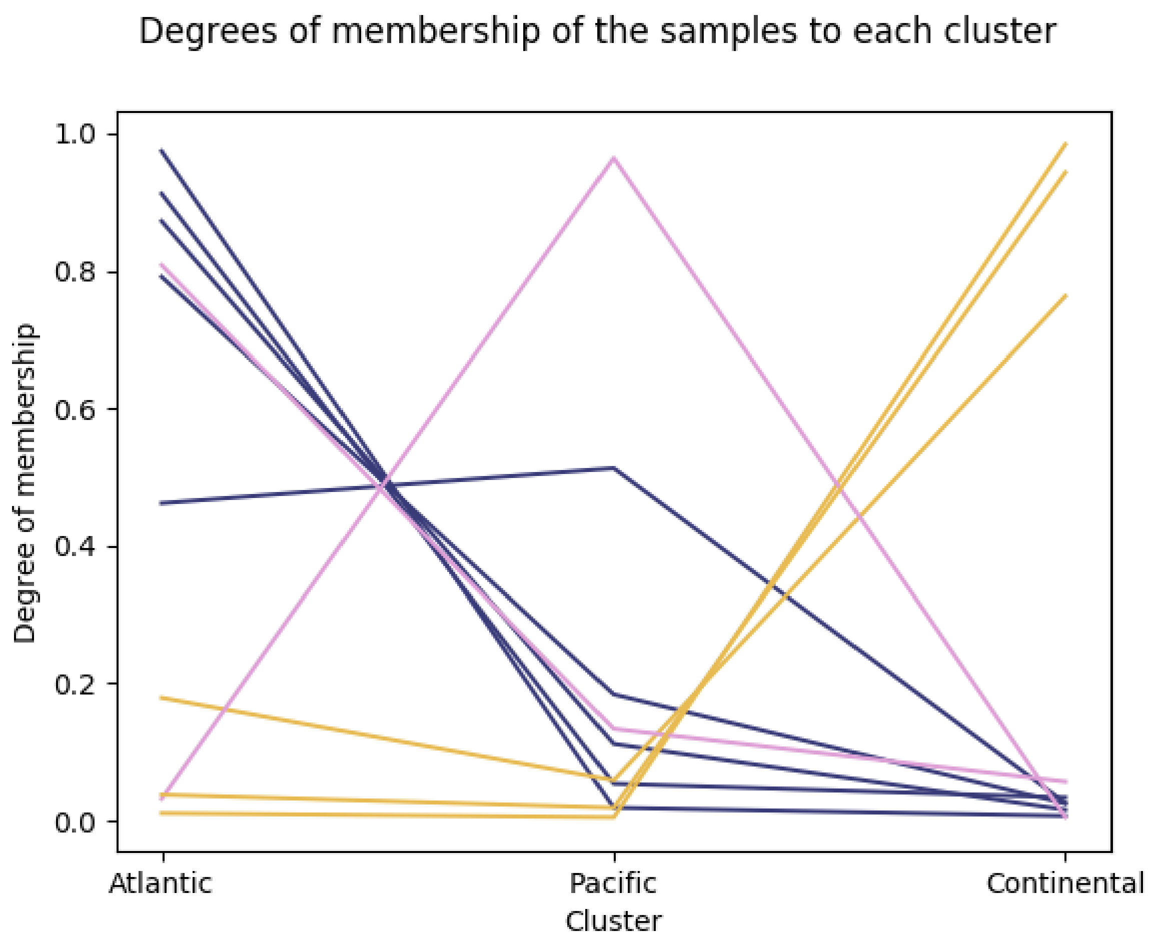
To see the information in a graphic way, the method `plot_clusters` can be used. It assigns each sample to the cluster whose membership value is the greatest.

```
plot_clusters(fuzzy_kmeans, fd, cluster_colors=cluster_colors,
              cluster_labels=cluster_labels)
```



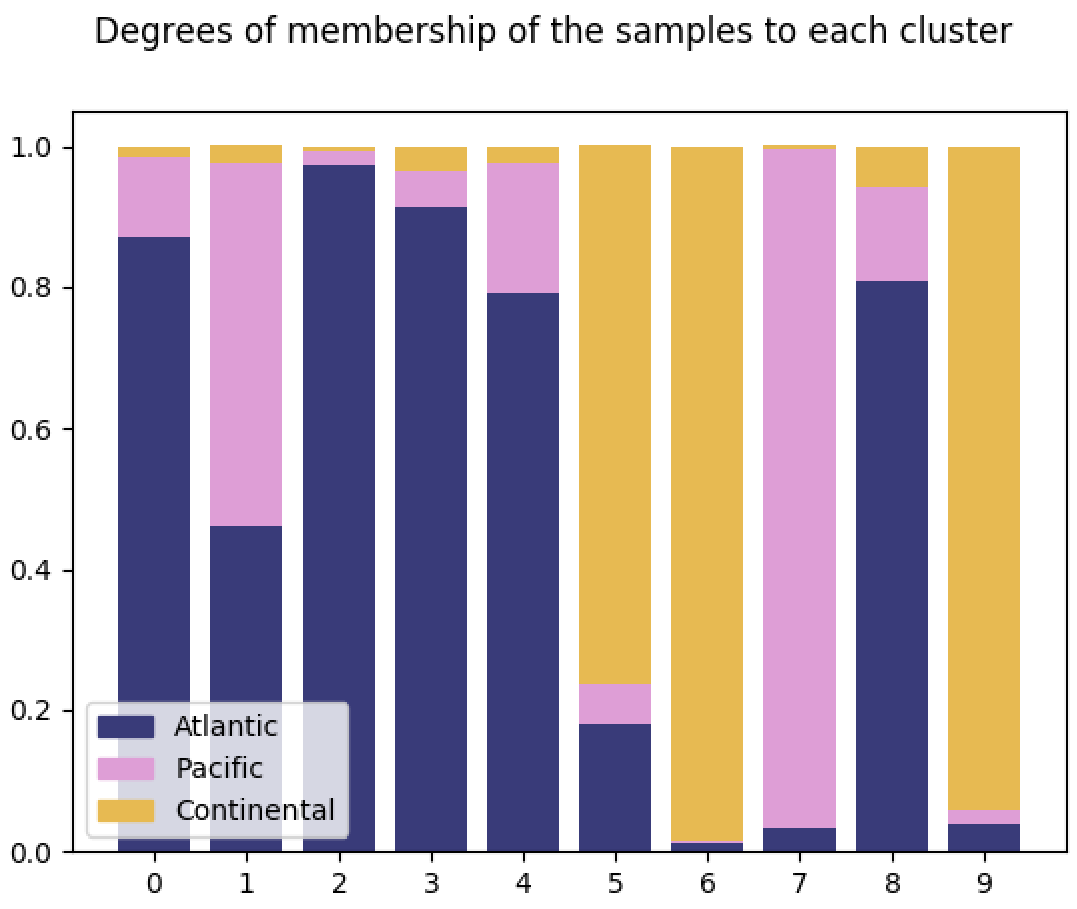
Another plot implemented to show the results in the class `Fuzzy K-Means` is the below one, which is similar to parallel coordinates. It is recommended to assign colors to each of the samples in order to identify them. In this example, the colors are the ones of the first plot, dividing the samples by climate.

```
colors_by_climate = colormap(indexer / (n_climates - 1))

plt.figure()
plot_cluster_lines(fuzzy_kmeans, fd, cluster_labels=cluster_labels,
                   sample_colors=colors_by_climate)
```



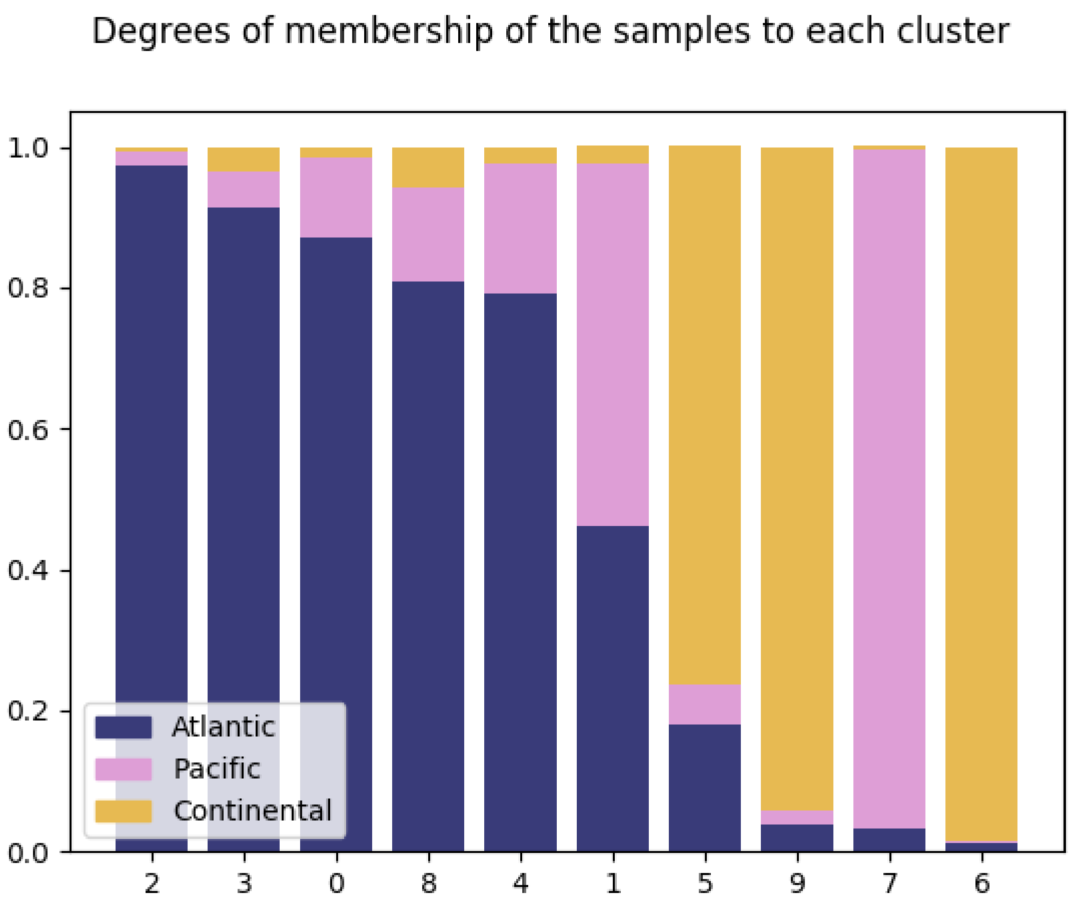
Lastly, the function `plot_cluster_bars` found in the module `clustering_plots`, returns a barplot. Each sample is designated with a bar which is filled proportionally to the membership values with the color of each cluster.

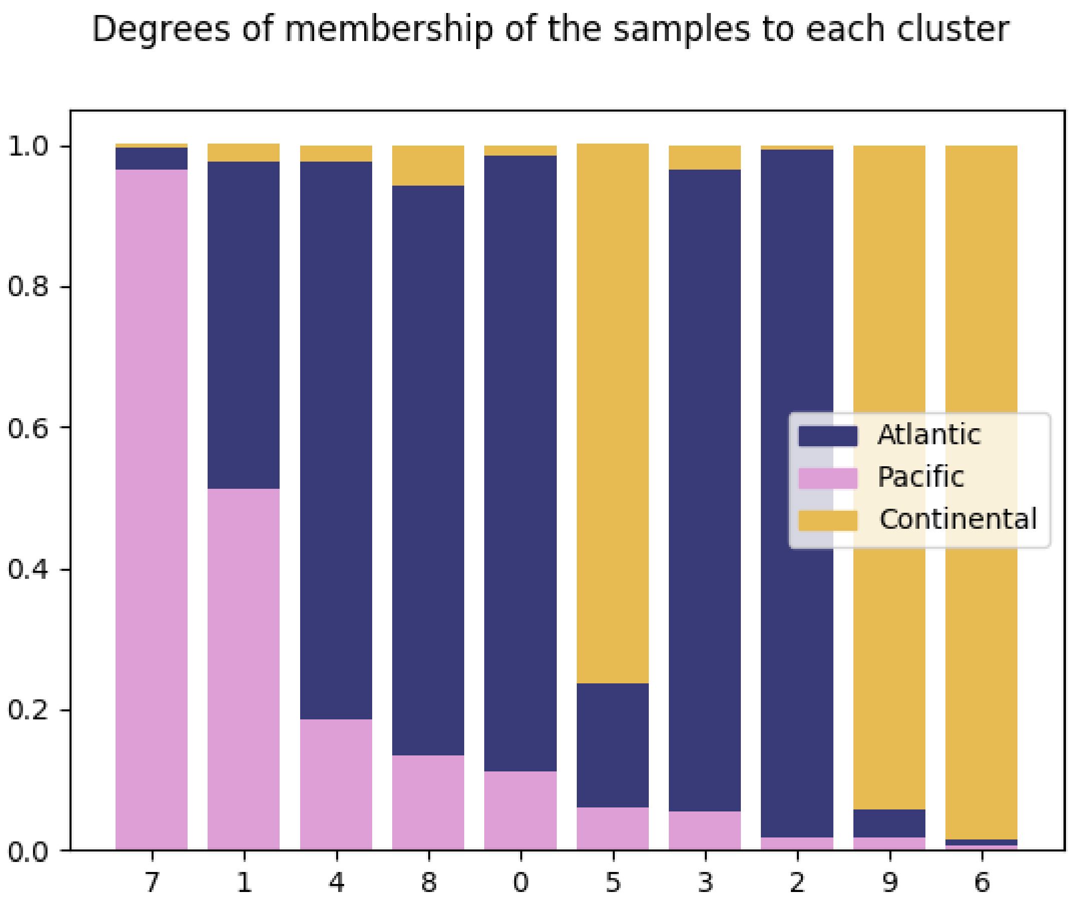
```
plt.figure()
plot_cluster_bars(fuzzy_kmeans, fd, cluster_colors=cluster_colors,
                       cluster_labels=cluster_labels)
```

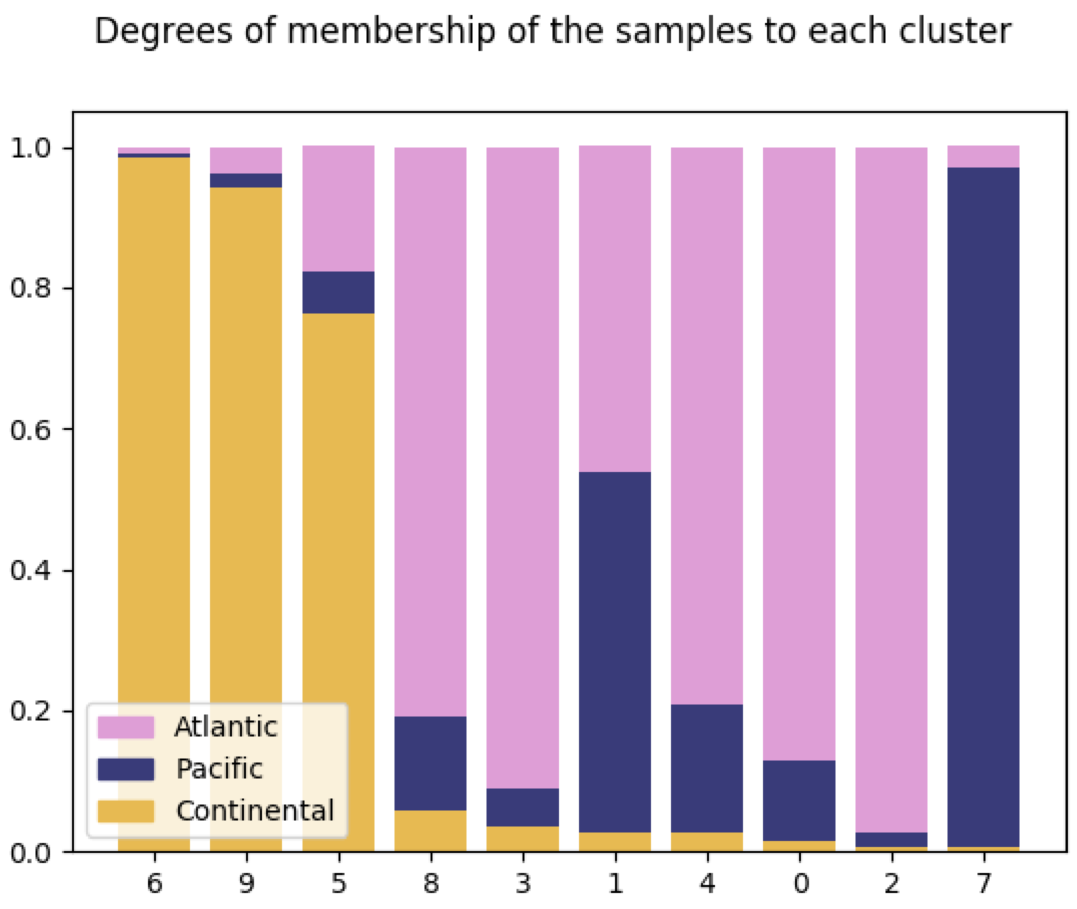


The possibility of sorting the bars according to a cluster is given specifying the number of cluster, which belongs to the interval [0, n_clusters).

```
plt.figure()
plot_cluster_bars(fuzzy_kmeans, fd, sort=0, cluster_colors=cluster_colors,
                       cluster_labels=cluster_labels)

plt.figure()
plot_cluster_bars(fuzzy_kmeans, fd, sort=1, cluster_colors=cluster_colors,
                       cluster_labels=cluster_labels)

plt.figure()
plot_cluster_bars(fuzzy_kmeans, fd, sort=2, cluster_colors=cluster_colors,
                       cluster_labels=cluster_labels)
```


Degrees of membership of the samples to each cluster
1.0
0.8
0.6
0.4
0.2
0.0
2
3
0
8
4
1
5
9
7
6
Atlantic
Pacific
Continental


Degrees of membership of the samples to each cluster
1.0
0.8
0.6
0.4
0.2
0.0
7
1
4
8
0
5
3
2
9
6
Atlantic
Pacific
Continental



Total running time of the script: (0 minutes 4.532 seconds)

Download Python source code: plot_clustering.py

Download Jupyter notebook: plot_clustering.ipynb

Gallery generated by Sphinx-Gallery

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.

`skfda.exploratory.depth.band_depth`(fdatagrid)	Implementation of Band Depth for functional c
`skfda.exploratory.depth.modified_band_depth`(…)	Implementation of Modified Band Depth for fu
`skfda.exploratory.depth.fraiman_muniz_depth`(…)	Implementation of Fraiman and Muniz (FM) De

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.

skfda.exploratory.depth.band_depth

`skfda.exploratory.depth.band_depth`(*fdatagrid*, *pointwise=False*) [source]

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 also returned. Defaults to False.

Returns: Array containing the band depth of the samples.

Return type: depth (numpy.darray)

Returns: Array containing the band depth of the samples at each point of discretisation. Only returned if pointwise equals to True.

Return type: depth_pointwise (numpy.darray, optional)

Examples

Univariate setting:

```
>>> 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)
>>> band_depth(fd)
array([[ 0.5       ],
       [ 0.83333333],
       [ 0.5       ],
       [ 0.5       ]])
```

Multivariate Setting:

```
>>> data_matrix = [[[[1, 3], [2, 6]], [[23, 54], [43, 76]], [[2, 45], [12, 65]]],
...                [[[21, 34], [8, 16]], [[67, 43], [32, 21]], [[10, 24], [3,
12]]],
...                [[[4, 6], [4, 10]], [[45, 48], [38, 56]], [[8, 36], [10, 28]]]]
>>> sample_points = [[2, 4, 6], [3, 6]]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> band_depth(fd)
array([[ 0.66666667,  0.66666667],
       [ 0.66666667,  0.66666667],
       [ 1.        ,  1.        ]])
```

skfda.exploratory.depth.modified_band_depth

`skfda.exploratory.depth.modified_band_depth`(*fdatagrid*, *pointwise=False*) [source]

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 also returned. Defaults to False.

Returns: Array containing the modified band depth of the samples.

Return type: depth (numpy.darray)

Returns: Array containing the modified band depth of the samples at each point of discretisation. Only returned if pointwise equals to True.

Return type: depth_pointwise (numpy.darray, optional)

Examples

Univariate setting specifying pointwise:

```
>>> 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)
>>> modified_band_depth(fd, pointwise = True)
(array([[ 0.5       ],
       [ 0.83333333],
       [ 0.72222222],
       [ 0.66666667]]), array([[[ 0.5       ],
        [ 0.5       ],
        [ 0.5       ],
        [ 0.5       ],
        [ 0.5       ],
        [ 0.5       ]],
<BLANKLINE>
       [[ 0.83333333],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.83333333]],
<BLANKLINE>
       [[ 0.5       ],
        [ 0.5       ],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.83333333]],
<BLANKLINE>
       [[ 0.83333333],
        [ 0.83333333],
        [ 0.83333333],
        [ 0.5       ],
        [ 0.5       ],
        [ 0.5       ]]]))
```

Multivariate Setting without specifying pointwise:

```
>>> data_matrix = [[[[1, 3], [2, 6]], [[23, 54], [43, 76]], [[2, 45], [12, 65]]],
...                [[[21, 34], [8, 16]], [[67, 43], [32, 21]], [[10, 24], [3,
12]]],
...                [[[4, 6], [4, 10]], [[45, 48], [38, 56]], [[34, 78], [10, 28]]]]
>>> sample_points = [[2, 4, 6], [3, 6]]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> modified_band_depth(fd)
array([[ 0.66666667,  0.72222222],
       [ 0.72222222,  0.66666667],
       [ 0.94444444,  0.94444444]])
```

skfda.exploratory.depth.fraiman_muniz_depth

`skfda.exploratory.depth.fraiman_muniz_depth`(*fdatagrid*, *pointwise=False*) [source]

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 adding the previously computed depth for each of its points.

Parameters:

- **fdatagrid** (*FDataGrid*) – Object over whose samples the FM depth is going to be calculated.
- **pointwise** (*boolean, optional*) – Indicates if the pointwise depth is also returned. Defaults to False.

Returns: Array containing the FM depth of the samples.

Return type: depth (numpy.darray)

Returns: Array containing the FM depth of the samples at each point of discretisation. Only returned if pointwise equals to True.

Return type: depth_pointwise (numpy.darray, optional)

Examples

Univariate setting specifying pointwise:

```
>>> 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)
>>> fraiman_muniz_depth(fd, pointwise = True)
(array([[ 0.5       ],
       [ 0.75      ],
       [ 0.91666667],
       [ 0.875     ]]), array([[[ 0.5 ],
        [ 0.5 ],
        [ 0.5 ],
        [ 0.5 ],
        [ 0.5 ],
        [ 0.5 ]],
<BLANKLINE>
       [[ 0.75],
        [ 0.75],
        [ 0.75],
        [ 0.75],
        [ 0.75],
        [ 0.75]],
<BLANKLINE>
       [[ 0.75],
        [ 0.75],
        [ 1.  ],
        [ 1.  ],
        [ 1.  ],
        [ 1.  ]],
<BLANKLINE>
       [[ 1.  ],
        [ 1.  ],
        [ 1.  ],
        [ 0.75],
        [ 0.75],
        [ 0.75]]]))
```

Multivariate Setting without specifying pointwise:

```
>>> data_matrix = [[[[1, 3], [2, 6]], [[23, 54], [43, 76]], [[2, 45], [12, 65]]],
...                [[[21, 34], [8, 16]], [[67, 43], [32, 21]], [[10, 24], [3,
12]]],
...                [[[4, 6], [4, 10]], [[45, 48], [38, 56]], [[34, 78], [10, 28]]]]
>>> sample_points = [[2, 4, 6], [3, 6]]
>>> fd = FDataGrid(data_matrix, sample_points)
>>> fraiman_muniz_depth(fd)
array([[ 0.72222222,  0.66666667],
       [ 0.66666667,  0.72222222],
       [ 0.77777778,  0.77777778]])
```

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.

The base abstract class from which the others inherit is FDataBoxplot.

`skfda.exploratory.visualization.boxplot.FDataBoxplot`([...])	Abstract class inherited by the Boxp

If the dimension of the domain is 1, the following class must be used. See Boxplot Example for detailed explanation.

`skfda.exploratory.visualization.boxplot.Boxplot`(...)	Representation of the functional boxplot.

If the dimension of the domain is 2, this one. See Surface Boxplot Example for detailed explanation.

`skfda.exploratory.visualization.boxplot.SurfaceBoxplot`(...)	Representation of the surface boxp

skfda.exploratory.visualization.boxplot.Boxplot

class **`skfda.exploratory.visualization.boxplot.Boxplot`**(*fdatagrid, method=<function modified_band_depth>, prob=[0.5], factor=1.5*) [source]

Representation of the functional boxplot.

Class implementing the functionl 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.

Type: FDataGrid

`median`

contains the median/s.

Type: array, (fdatagrid.ndim_image, nsample_points)

`central_envelope`

contains the central envelope/s.

Type: array, (fdatagrid.ndim_image, 2, nsample_points)

`outlying_envelope`

contains the outlying envelope/s.

Type: array, (fdatagrid.ndim_image, 2, nsample_points)

`colormap`

Colormap from which the colors to represent the central regions are selected.

Type: matplotlib.colors.LinearSegmentedColormap

`central_regions (array, (fdatagrid.ndim_image * ncentral_regions, 2,`

nsample_points)): contains the central regions.

`outliers`

contains the outliers

Type: array, (fdatagrid.ndim_image, fdatagrid.nsamples)

`barcol`

Color of the envelopes and vertical lines.

Type: string

`outliercol`

Color of the ouliers.

Type: string

`mediancol`

Color of the median.

Type: string

`show_full_outliers`

If False (the default) then only the part outside the box is plotted. If True, complete outling curves are plotted

Type: boolean

Example

Function $f : \mathbb{R} \longmapsto \mathbb{R}$.

```
>>> 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, dataset_label="dataset",
axes_labels=["x_label", "y_label"])
>>> Boxplot(fd)
Boxplot(
    FDataGrid=FDataGrid(
        array([[[ 1. ],
                [ 1. ],
                [ 2. ],
                [ 3. ],
                [ 2.5],
                [ 2. ]],
<BLANKLINE>
               [[ 0.5],
                [ 0.5],
                [ 1. ],
                [ 2. ],
                [ 1.5],
                [ 1. ]],
<BLANKLINE>
               [[-1. ],
                [-1. ],
                [-0.5],
                [ 1. ],
                [ 1. ],
                [ 0.5]],
<BLANKLINE>
               [[-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([[[ 0.5,  0.5,  1. ,  2. ,  1.5,  1. ],
            [-1. , -1. , -0.5,  1. ,  1. ,  0.5]]]),
    outlying envelope=array([[[ 1.  ,  1.  ,  2.  ,  3.  ,  2.25,  1.75],
            [-1.  , -1.  , -0.5 , -0.5 ,  0.25, -0.25]]]),
    central_regions=array([[[ 0.5,  0.5,  1. ,  2. ,  1.5,  1. ],
            [-1. , -1. , -0.5,  1. ,  1. ,  0.5]]]),
    outliers=array([[ 1.,  0.,  0.,  1.]]))
```

`__init__`(*fdatagrid, method=<function modified_band_depth>, prob=[0.5], factor=1.5*) [source]

Initialization of the Boxplot class.

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.

Methods

`__init__`(fdatagrid[, method, prob, factor])	Initialization of the Boxplot class.
`plot`([fig, ax, nrows, ncols])	Visualization of the functional boxplot of the fda

Attributes

`central_envelope`	
`central_regions`	
`colormap`	
`factor`	
`fdatagrid`	
`median`	
`outliers`	
`outlying_envelope`	
`show_full_outliers`	

skfda.exploratory.visualization.boxplot.SurfaceBoxplot

class **skfda.exploratory.visualization.boxplot.SurfaceBoxplot**(*fdatagrid, method=<function modified_band_depth>, factor=1.5*) [source]

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.ndim_image, lx, ly)

central_envelope

contains the central envelope/s.

Type: array, (fdatagrid.ndim_image, 2, lx, ly)

outlying_envelope

contains the outlying envelope/s.

Type: array,(fdatagrid.ndim_image, 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 $f : \mathbb{R}^2 \longmapsto \mathbb{R}^2$.

```
>>> data_matrix = [[[[1, 4], [0.3, 1.5], [1, 3]], [[2, 8], [0.4, 2], [2, 9]]],
...                [[[2, 10], [0.5, 3], [2, 10]], [[3, 12], [0.6, 3], [3, 15]]]]
>>> sample_points = [[2, 4], [3, 6, 8]]
>>> fd = FDataGrid(data_matrix, sample_points, dataset_label= "dataset",
...                axes_labels=["x1_label", "x2_label", "y1_label", "y2_label"])
>>> SurfaceBoxplot(fd)
SurfaceBoxplot(
    FDataGrid=FDataGrid(
        array([[[[  1. ,   4. ],
                 [  0.3,   1.5],
                 [  1. ,   3. ]],
<BLANKLINE>
                [[  2. ,   8. ],
                 [  0.4,   2. ],
                 [  2. ,   9. ]]],
<BLANKLINE>
<BLANKLINE>
               [[[  2. ,  10. ],
                 [  0.5,   3. ],
                 [  2. ,  10. ]],
<BLANKLINE>
                [[  3. ,  12. ],
                 [  0.6,   3. ],
                 [  3. ,  15. ]]]]),
        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', 'y1_label', 'y2_label'],
        extrapolation=None,
        interpolator=SplineInterpolator(interpolation_order=1,
smoothness_parameter=0.0, monotone=False),
        keepdims=False),
    median=array([[[ 1. ,  0.3,  1. ],
            [ 2. ,  0.4,  2. ]],
<BLANKLINE>
           [[ 4. ,  1.5,  3. ],
            [ 8. ,  2. ,  9. ]]]),
    central envelope=array([[[[ 1. ,  0.3,  1. ],
             [ 2. ,  0.4,  2. ]],
<BLANKLINE>
            [[ 1. ,  0.3,  1. ],
             [ 2. ,  0.4,  2. ]]],
<BLANKLINE>
<BLANKLINE>
           [[[ 4. ,  1.5,  3. ],
             [ 8. ,  2. ,  9. ]],
<BLANKLINE>
            [[ 4. ,  1.5,  3. ],
             [ 8. ,  2. ,  9. ]]]]),
    outlying envelope=array([[[[ 1. ,  0.3,  1. ],
             [ 2. ,  0.4,  2. ]],
<BLANKLINE>
            [[ 1. ,  0.3,  1. ],
             [ 2. ,  0.4,  2. ]]],
<BLANKLINE>
<BLANKLINE>
           [[[ 4. ,  1.5,  3. ],
             [ 8. ,  2. ,  9. ]],
<BLANKLINE>
            [[ 4. ,  1.5,  3. ],
             [ 8. ,  2. ,  9. ]]]]))
```

`__init__`(*fdatagrid*, *method=<function modified_band_depth>*, *factor=1.5*) [source]

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.

Methods

`__init__`(fdatagrid[, method, factor])	Initialization of the functional boxplot.
`plot`([fig, ax, nrows, ncols])	Visualization of the surface boxplot of the fdatagrid (ndim

Attributes

`boxcol`	
`central_envelope`	
`colormap`	
`factor`	
`fdatagrid`	
`median`	
`outcol`	
`outlying_envelope`	

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 in the below function.

`skfda.exploratory.visualization.magnitude_shape_plot.directional_outlyingness`(...)	Computes t

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.

`skfda.exploratory.visualization.magnitude_shape_plot.MagnitudeShapePlot`(...)	Implementation o

skfda.exploratory.visualization.magnitude_shape_plot.directional_o

skfda.exploratory.visualization.magnitude_shape_plot.directional_outlyingness(*fdatagrid, depth_method=<function modified_band_depth>, dim_weights=None, pointwise_weights=None*) [source]

Computes the directional outlyingness of the functional data.

Furthermore, it calculates both the mean and the variation of the directional outlyingness of the samples in the data set, which are also returned.

The first one, 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 second one, 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:

$$\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) = \left\{\frac{1}{d\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right)} - 1\right\} \cdot \mathbf{v}(t)$$

where $\mathbf{X}$ is a stochastic process with probability distribution F, d a depth function and $\mathbf{v}(t) = \left\{\mathbf{X}(t) - \mathbf{Z}(t)\right\} / \|\mathbf{X}(t) - \mathbf{Z}(t)\|$ is the spatial sign of $\left\{\mathbf{X}(t) - \mathbf{Z}(t)\right\}$, $\mathbf{Z}(t)$ denotes the median and $\|\cdot\|$ denotes the L_2 norm.

From the above formula, we define the mean directional outlyingness as:

$$\mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I \mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) \cdot w(t)dt;$$

and the variation of the directional outlyingness as:

$$VO\left(\mathbf{X}, F_{\mathbf{X}}\right) = \int_I \|\mathbf{O}\left(\mathbf{X}(t), F_{\mathbf{X}(t)}\right) - \mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right)\|^2 \cdot w(t)dt$$

where $w(t)$ a weight function defined on the domain of $\mathbf{X}$, I.

Then, the total functional outlyingness can be computed using these values:

$$FO\left(\mathbf{X}, F_{\mathbf{X}}\right) = \|\mathbf{MO}\left(\mathbf{X}, F_{\mathbf{X}}\right)\|^2 + VO\left(\mathbf{X}, F_{\mathbf{X}}\right).$$

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`.
- **dim_weights** (*array_like, optional*) – an array containing the weights of each of the dimensions of the image. Defaults to the same weight for each of the dimensions: 1/ndim_image.
- **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:

tuple containing:

dir_outlyingness (numpy.array((fdatagrid.shape))): List containing the values of the directional outlyingness of the FDataGrid object.

mean_dir_outl (numpy.array((fdatagrid.nsamples, 2))): List containing the values of the magnitude outlyingness for each of the samples.

variation_dir_outl (numpy.array((fdatagrid.nsamples,))): List containing the values of the shape outlyingness for each of the samples.

Return type: (tuple)

Example

```
>>> 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)
>>> directional_outlyingness(fd)
(array([[[ 1. ],
        [ 1. ],
        [ 1. ],
        [ 1. ],
        [ 1. ],
        [ 1. ]],
<BLANKLINE>
       [[ 0. ],
        [ 0. ],
        [ 0. ],
        [ 0. ],
        [ 0. ],
        [ 0. ]],
<BLANKLINE>
       [[-1. ],
        [-1. ],
        [-0.2],
        [-0.2],
        [-0.2],
        [-0.2]],
<BLANKLINE>
       [[-0.2],
        [-0.2],
        [-0.2],
        [-1. ],
        [-1. ],
        [-1. ]]]), array([[ 1.66666667],
       [ 0.        ],
       [-0.73333333],
       [-1.        ]]), array([ 0.74074074,  0.        ,  0.36740741,  0.53333333]))
```

skfda.exploratory.visualization.magnitude_shape_plot.Magnitude

class **`skfda.exploratory.visualization.magnitude_shape_plot.MagnitudeShapePlot`**(*fdatagrid, depth_method=<function modified_band_depth>, dim_weights=None, pointwise_weights=None, alpha=0.993, assume_centered=False, support_fraction=None, random_state=0*) [source]

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 ($\|\mathbf{MO}\|$) is plotted in the x-axis, and the variation of the directional outlyingness (VO) in the y-axis.

Considering $\mathbf{Y} = \left(\mathbf{MO}^T, VO\right)^T$, the outlier detection method is implemented as described below.

First, the square robust Mahalanobis distance is calculated based on a sample of size $h \leq fdatagrid.nsamples$:

$$RMD^2\left(\mathbf{Y}, \mathbf{\tilde{Y}}^*_J\right) = \left(\mathbf{Y} - \mathbf{\tilde{Y}}^*_J\right)^T \mathbf{S}^*_J{}^{-1} \left(\mathbf{Y} - \mathbf{\tilde{Y}}^*_J\right)$$

where J denotes the group of h samples that minimizes the determinant of the corresponding covariance matrix, $\mathbf{\tilde{Y}}^*_J = h^{-1}\sum_{i\in J}\mathbf{Y}_i$ and $\mathbf{S}^*_J = h^{-1}\sum_{i\in J}\left(\mathbf{Y}_i - \mathbf{\tilde{Y}}^*_J\right)\left(\mathbf{Y}_i - \mathbf{\tilde{Y}}^*_J\right)^T$. The sub-sample of size h controls the robustness of the method.

Then, the tail of this distance distribution is approximated as follows:

$$\frac{c(m-p)}{m(p+1)} RMD^2\left(\mathbf{Y}, \mathbf{\tilde{Y}}^*_J\right) \sim F_{p+1, m-p}$$

where p is the dmension of the image, and c and m are parameters determining the degrees of freedom of the F-distribution and the scaling factor.

$$c = E\left[s^*_{jj}\right]$$

where s^*_{jj} are the diagonal elements of MCD and

$$m = \frac{2}{CV^2}$$

where CV is the estimated coefficient of variation of the diagonal elements of the MCD shape estimator.

Finally, we choose a cutoff value to determine the outliers, C , as the α 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.

`fdatagrid`

Object to be visualized.

Type: FDataGrid

`depth_method`

Method used to order the data. Defaults to `modified band depth`.

Type: depth measure, optional

`dim_weights`

an array containing the weights of each of the dimensions of the image.

Type: array_like, 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, respecively.

Type: 1-D array, (fdatagrid.nsamples,)

`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

```
>>> 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)
>>> MagnitudeShapePlot(fd)
MagnitudeShapePlot(
    FDataGrid=FDataGrid(
        array([[[ 1. ],
                [ 1. ],
                [ 2. ],
                [ 3. ],
                [ 2.5],
                [ 2. ]],
<BLANKLINE>
               [[ 0.5],
                [ 0.5],
                [ 1. ],
                [ 2. ],
                [ 1.5],
                [ 1. ]],
<BLANKLINE>
               [[-1. ],
                [-1. ],
                [-0.5],
                [ 1. ],
                [ 1. ],
                [ 0.5]],
<BLANKLINE>
               [[-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=None,
        axes_labels=None,
        extrapolation=None,
        interpolator=SplineInterpolator(interpolation_order=1, smoothness_parameter=0.0,
monotone=False),
        keepdims=False),
    depth_method=modified_band_depth,
    dim_weights=None,
    pointwise_weights=None,
    alpha=0.993,
    points=array([[ 1.66666667,  0.74074074],
           [ 0.        ,  0.        ],
           [-0.73333333,  0.36740741],
           [-1.        ,  0.53333333]]),
    outliers=array([0, 0, 0, 0]),
    colormap=seismic,
    color=0.2,
    outliercol=(0.8,),
    xlabel='MO',
    ylabel='VO',
    title='MS-Plot')
```

__init__(*fdatagrid, depth_method=<function modified_band_depth>, dim_weights=None, pointwise_weights=None, alpha=0.993, assume_centered=False, support_fraction=None, random_state=0*)
[source]

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 `modified band depth`.
- **dim_weights** (*array_like, optional*) – an array containing the weights of each of the dimensions of the image.
- **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.

Methods

`__init__`(fdatagrid[, depth_method, …])	Initialization of the MagnitudeShapePlot class.
`plot`([ax])	Visualization of the magnitude shape plot of the fdatagrid.

Attributes

`alpha`	
`color`	
`colormap`	
`depth_method`	
`dim_weights`	
`fdatagrid`	
`outliercol`	
`outliers`	
`points`	
`pointwise_weights`	

Clustering

Functions to cluster functional data in a FDataGrid object.

This module contains functions to group observations in such a way that those in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Modules:

KMeans algorithms

The following classes implement both, the K-Means and the Fuzzy K-Means algorithms respectively. They both inherit from the `BaseKMeans class`.

`skfda.ml.clustering.base_kmeans.KMeans`([...])	
`skfda.ml.clustering.base_kmeans.FuzzyKMeans`([...])	

In order to show the results in a visual way, the module `clustering_plots` can be used.

See Clustering Example for detailed explanation.

skfda.ml.clustering.base_kmeans.KMeans

class **`skfda.ml.clustering.base_kmeans.KMeans`**(*n_clusters=2, init=None, metric=<function pairwise_distance.<locals>.pairwise>, n_init=1, max_iter=100, tol=0.0001, random_state=0*) [source]

`__init__`(*n_clusters=2, init=None, metric=<function pairwise_distance.<locals>.pairwise>, n_init=1, max_iter=100, tol=0.0001, random_state=0*) [source]

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_marix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.ndim_image). Defaults to None, and the centers are initialized randomly.
- **metric** (*optional*) – metric that acceps two FDataGrid objects and returns a matrix with shape (fdatagrid1.nsamples, fdatagrid2.nsamples). Defaults to *pairwise_distance(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.

Methods

`__init__`([n_clusters, init, metric, n_init, …])	Initialization of the KMeans class.
`fit`(X[, y, sample_weight])	Computes K-Means clustering calculating the at
`fit_predict`(X[, y, sample_weight])	Compute cluster centers and predict cluster inde
`fit_transform`(X[, y, sample_weight])	Compute clustering and transform X to cluster-c

`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, sample_weight])	Predict the closest cluster each sample in X belo
`score`(X[, y, sample_weight])	Opposite of the value of X on the K-means obje
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X to a cluster-distance space.

skfda.ml.clustering.base_kmeans.FuzzyKMeans

class **`skfda.ml.clustering.base_kmeans.FuzzyKMeans`**(*n_clusters=2, init=None, metric=<function pairwise_distance.<locals>.pairwise>, n_init=1, max_iter=100, tol=0.0001, random_state=0, fuzzifier=2, n_dec=3*) [source]

> **`__init__`**(*n_clusters=2, init=None, metric=<function pairwise_distance.<locals>.pairwise>, n_init=1, max_iter=100, tol=0.0001, random_state=0, fuzzifier=2, n_dec=3*) [source]

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_marix must be of the shape (n_clusters, fdatagrid.ncol, fdatagrid.ndim_image). Defaults to None, and the centers are initialized randomly.
- **metric** (*optional*) – metric that acceps two FDataGrid objects and returns a matrix with shape (fdatagrid1.nsamples, fdatagrid2.nsamples). Defaults to *pairwise_distance(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.
- **n_dec** (*int*, *optional*) – designates the number of decimals of the labels returned in the fuzzy algorithm. Defaults to 3.

Methods

`__init__`([n_clusters, init, metric, n_init, ...])	Initialization of the FuzzyKMeans class.

`fit`(X[, y, sample_weight])	Computes Fuzzy K-Means clustering calculating
`fit_predict`(X[, y, sample_weight])	Compute cluster centers and predict cluster inde
`fit_transform`(X[, y, sample_weight])	Compute clustering and transform X to cluster-c
`get_params`([deep])	Get parameters for this estimator.
`predict`(X[, sample_weight])	Predict the closest cluster each sample in X belo
`score`(X[, y, sample_weight])	Opposite of the value of X on the K-means obje
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Transform X to a cluster-distance space.

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_plots.plot_clusters`(...)	Plot of the FDataGrid
`skfda.exploratory.visualization.clustering_plots.plot_cluster_lines`(...)	Implementation of the
`skfda.exploratory.visualization.clustering_plots.plot_cluster_bars`(...)	Implementation of the

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.

skfda.exploratory.visualization.clustering_plots.plot_cluste

skfda.exploratory.visualization.clustering_plots.plot_clusters(*estimator, X, fig=None, ax=None, nrows=None, ncols=None, sample_labels=None, cluster_colors=None, cluster_labels=None, center_colors=None, center_labels=None, center_width=3, colormap=<matplotlib.colors.LinearSegmentedColormap object>*) [source]

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 leglend.

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.
- **ax** (*list of axis objects*) – axis over where the graphs are plotted. If None, see param fig.
- **nrows** (*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.
- **ncols** (*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 of the fdatagrid.
- **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 colors*) – 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)

skfda.exploratory.visualization.clustering_plots.plot_cluster_

`skfda.exploratory.visualization.clustering_plots.plot_cluster_lines`(*estimator, X, fig=None, ax=None, sample_colors=None, sample_labels=None, cluster_labels=None, colormap=<matplotlib.colors.LinearSegmentedColormap object>, xlabel=None, ylabel=None, title=None*) [source]

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** (*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.
- **ax** (*axis 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.
- **xlabel** (*str*) – Label for the x-axis. Defaults to “Sample”.
- **ylabel** (*str*) – Label for the y-axis. Defaults to “Degree of membership”.
- **title** (*str*, *optional*) – Title for the figure where the clustering results are ploted. Defaults to “Degrees of membership of the samples to each cluster”.

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)

skfda.exploratory.visualization.clustering_plots.plot_cluster_

skfda.exploratory.visualization.clustering_plots.plot_cluster_bars(*estimator, X, fig=None, ax=None, sort=-1, sample_labels=None, cluster_colors=None, cluster_labels=None, colormap=<matplotlib.colors.LinearSegmentedColormap object>, xlabel=None, ylabel=None, title=None*) [source]

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.
- **ax** (*axis object, optional*) – axis 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 decrementing 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.
- **xlabel** (*str*) – Label for the x-axis. Defaults to “Sample”.
- **ylabel** (*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)