Bachelor thesis

Dimensionality Reduction for Functional Regression

David del Val Mończolí

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Computer Science

BACHELOR THESIS

Dimensionality Reduction for Functional Regression

Author: David del Val Moncholí
Advisor: Alberto Suárez González

junio 2023

David del Val Moncholí
Dimensionality Reduction for Functional Regression

David del Val Moncholí
Calle El Molino, 15
Capella (Huesca)
Spain

PRINTED IN SPAIN

Agradecimientos

Me gustaría agradecer toda la ayuda recibida por los miembros del equipo de desarrollo de *scikit-learn*. En especial a Carlos Ramos, que ha atendido todas mis cuestiones y dudas sobre programación y diseño y ha revisado minuciosamente todos los *pull request*. Gracias a él he aprendido mucho sobre el desarrollo de software de calidad y, en particular, sobre el desarrollo de software en Python.

Por otro lado, quería agradecer su dedicación a mi tutor Alberto Suárez. No solo ha dedicado su tiempo a mejorar mis conocimientos sobre aprendizaje automático y datos funcionales sino que también, junto a Carlos Ramos, me ha ayudado durante el proceso de desarrollo. Sus aportaciones y comentarios han sido de gran ayuda para garantizar el rigor tanto del código como del presente documento.

Por último, quisiera agradecer a mi padre. Su apoyo durante todo el año ha sido invaluable. Nunca habría podido llegar a donde estoy ahora si no hubiese sido por él.

Resumen

El análisis de datos funcionales (FDA por sus siglas en inglés) es la rama de la estadística que estudia cantidades que dependen en parámetros continuos, como las variaciones de temperatura o la evolución de la actividad cardiaca a lo largo del tiempo. Estos datos son, en principio, infinito dimensionales ya que las funciones observadas pueden ser evaluadas en cualquier punto de su dominio. En este contexto puede ser muy útil identificar componentes que capturen información relevante. Estos pueden ser utilizados para formular métodos más sencillos, interpretables y, en ocasiones, precisos que aquellos basados en los datos originales.

En este trabajo se exploran dos métodos de reducción de dimensionalidad. El análisis de componentes principales (PCA por sus siglas en inglés) elige las componentes buscando maximizar su varianza mientras que mínimos cuadrados parciales (PLS por sus siglas en inglés) maximiza la covarianza entre las componentes de dos grupos o bloques de variables. Estos métodos de reducción de dimensionalidad se pueden extender para tratar datos funcionales, resumiendo la información de trayectorias infinito-dimensionales en espacios de dimensión finita. Asimismo, estas dos técnicas pueden ser aplicadas a problemas de regresión funcional, a fin de solventar problemas como la multicolinearidad o la irregularidad excesiva de los coeficientes del modelo. En particular, para mitigar este último problema, se incorporan términos de penalización de rugosidad al criterio de optimización.

Este proyecto busca diseñar e implementar herramientas computaciones para las versiones funcionales de PCA y PLS, a fin de integrarlas en la la librería *scikit-fda*. Esta es una librería de Python para el análisis de datos funcionales. En particular, busca ser una alternativa a opciones en R como *fda* o *fda.usc* en uno de los lenguajes de programación más utilizados en el ámbito del análisis de datos y el aprendizaje automático. Asimismo, esta librería proporciona una interfaz compatible con *scikit-learn*, que es familiar para muchos usuarios de Python en contextos científicos.

Finalmente, se incluyen tres ejemplos del uso de estos métodos en conjuntos de datos reales a fin de ilustrar su eficacia. Los primeros dos ejemplos muestran las capacidades de regularización de los dos métodos de regresión considerados. Por otra parte, el último ejemplo presenta un escenario en el que PLS obtiene mejores resultados que PCA, demostrando el valor de PLS en este tipo de problemas.

Palabras clave

Análisis de datos funcionales, Python, análisis de componentes principales, mínimos cuadrados parciales, regresión por componentes principales, regresión por mínimos cuadrados parciales

Abstract

Functional data analysis (FDA) is the branch of statistics that studies quantities that depend on continuous parameters, such as temperature variations at a geographical location or the heart's electrical activity in the cardiac cycle. These data are, in principle, infinitely-dimensional because the functions observed can be evaluated at any point within their domain. In this context, it can be helpful to identify components that capture relevant information. These components can then be utilized to formulate models that are simpler, typically more interpretable and, in some cases, more accurate than those based on the original observations.

In this work, two dimensionality reduction techniques are explored. In Principal component analysis (PCA), the components are obtained by maximizing the explained variance for a set of variables. Partial least squares (PLS) seeks to maximize the covariance between the components of two groups or blocks of variables. These dimensionality reduction techniques can be extended to deal with functional data. They provide a way to summarize the information of trajectories in an infinitely-dimensional space by projecting onto a finite-dimensional one. Moreover, one can apply these techniques in functional regression methods to address problems such as multicollinearity or excessive roughness of the fitted coefficients. In particular, to address this last issue, roughness penalty terms can be added to the optimization criteria of the dimensionality reduction process to smooth the projection functions.

This project aims to design and implement computational tools for functional PCA and PLS, and integrate them into the library *scikit-fda*. *Scikit-fda* is a Python library for data analysis and machine learning with functional data. This library seeks to provide an alternative to popular R packages such as *fda* or *fda.usc* in one of the most commonly used languages for data analysis and machine learning. Furthermore, this library provides a compatible interface with *scikit-learn*. Thus, the interface of the library should be familiar to many Python scientific users, and functionality of *scikit-learn* can be reused.

Finally, three examples in real datasets showcase the effectiveness of these methods. The first two examples demonstrate the regularization capabilities of each of the regression methods considered. In turn, the last example focuses on how PLS might be considerably superior to PCA in some circumstances.

Keywords

Functional data analysis, Python, principal component analysis, partial least squares, principal component regression, partial least squares regression, roughness penalties

Table of Contents

1 Introduction **1**
1.1 Goals and Scope 2
1.2 Structure of the document 2

2 State of the Art **3**
2.1 Principal Component Analysis 6
2.1.1 Regularized Principal Components 8
2.1.2 Functional Principal Components (FPC) Regression 9
2.2 Partial Least Squares (PLS) 10
2.2.1 Functional Partial Least Squares (FPLS) for dimensionality reduction 10
2.2.2 FPLS Regularization 13
2.2.3 FPLS Regression 14
2.3 Computational tools for FDA 15
2.3.1 The *scikit-fda* package 15

3 Software Development Process **17**
3.1 Analysis 17
3.1.1 Functional Requirements 17
3.1.2 Non-functional requirements 19
3.1.3 Use cases 20
3.2 Design 22
3.2.1 Representation module 22
3.2.2 Miscellaneous module 24
3.2.3 Machine Learning and Preprocessing modules 25
3.3 Development methodology 27
3.3.1 Project organization 27
3.3.2 Code quality assurance 29
3.3.3 Documentation 31
3.4 Testing 33
3.4.1 Doctests 33
3.4.2 Unit tests 33
3.4.3 Functionality tests 34
3.4.4 Continuous integration and testing 34

4 Results **35**

4.1 FPCR applied to the Canadian Weather dataset 35
4.2 FPLSR applied to the Canadian Weather dataset 37
4.3 Comparison between FPLSR and FPCR in the Tecator dataset 38

5 Conclusions and future work **39**

Bibliography **44**

Appendices **45**

A Multivariate PLS **47**
A.1 The steps of multivariate NIPALS 47
A.2 Latent Variables Interpretation 48
A.3 Multivariate NIPALS properties 50
A.4 Multivariate PLS regression 51

B Roughness penalty **53**

C Gantt diagram **59**

D Additional Design Diagrams **61**

Lists

List of equations

2.1 Scalar response functional regression model 3
2.2 Discretized scalar response functional regression model 4
2.3 Data matrix definitions for discretized representation 4
2.4 Finite basis expansion of functions 5
2.5 Data matrix definition for basis expansion representation 5
2.6 Scalar response model using a basis expansion representation 5
2.7 Basis coefficients calculated using the inner product matrix 5
2.8 Multivariate PCA optimization problem 6
2.9 Functional PCA optimization problem 7
2.10a Variance approximation in the discretized representation 7
2.10b L^2 norm approximation in the discretized representation 7
2.11 Penalized variance definition 8
2.12 Penalized variance formula for discretized functions 9
2.13 Ortogonality relationship between regularized weights 9
2.14 Orthogonality restriction between PLS components 10
2.15 Covariance approximation in functional PLS 11
2.16 Functional PLS scores 11
2.17 Functional PLS loadings 12
2.18 Functional PLS rotational matrices 13
2.19 Penalized covariance when X is functional 13
2.20 Sample penalized covariance when X is functional 13
2.21 Sample penalized covariance when X is functional after basis change 13
2.22 Penalized covariance when both blocks are functional. 14
2.23 Sample penalized covariance when both blocks are functional 14

A.1 Multivariate PLS weights optimization criteria 47
A.2 Multivariate PLS weights eigenvalue problem 47
A.3 Multivariate PLS loadings 48
A.4 NIPALS results arranged into matrices 50
A.5 Multivariate canonical NIPALS decomposition 50
A.6 Multivariate canonical NIPALS rotations 51
A.7 Multivariate canonical NIPALS scores projection 51

A.8a Multivariate regression NIPALS data reconstruction 51
A.8b Multivariate regression NIPALS data projection 51
A.9 Multivariate regression NIPALS **Y** prediction 51
A.10 Regression matrix for multivariate PLS regression. 52

B.1 Second derivative penalization 53
B.2 Linear differential operator 53
B.3 Second derivative penalization 53
B.4 Penalty matrix derivation 54
B.5 Penalty matrix definition 54
B.6 Second derivative approximation 55
B.7 Differentiation weights 55
B.8 Differentiation values 56
B.9 Matrix expression for differentiation values 56
B.10 Differentiaion weights for linear differential operator 56
B.11 Discretization of L^2 norm 57
B.12 Condition for 0-value entries in the penalty matrix 57
B.13 Optimized discretization of L^2 norm 57
B.14 Penalty definition in terms of **P** 58

List of figures

2.1 Trajectories of the Tecator dataset 4
2.2 Principal components of the AEMET dataset 8
2.3 Diagram of *scikit-fda* modules 16

3.1 FPC use case diagram 21
3.2 FPLS use case diagram 21
3.3 Modified modules in *scikit-fda* 22
3.4 Representation module design changes 23
3.5 Call graph for penalty matrix calculation 24
3.6 Design overview of the Machine Learning and Preprocessing modules 26
3.7 Development cycle diagram 27
3.8 Gitflow branching structure 28
3.9 Example of a call graph 31

4.1 Trajectories of the Canadian Weather dataset 35
4.2 Results of FPCR in the Canadian Weather dataset 36
4.3 FPCR weights for the Canadian Weather dataset 36

4.4 FPLS results on the Canadian Weather dataset . . . 37
4.5 FPLS weights on the Canadian Weather dataset . . . 37
4.6 FPLSR and FPCR applied to the Tcator dataset . . . 38

A.1 Latent variable models for multivariate NIPALS . . . 49

C.1 Gantt diagram of the project . . . 60

D.1 Detailed design of the new *`Basis`* subclasses . . . 62
D.2 Detailed design of the FPCA and FPCR functionality . . . 63
D.3 Detailed design of the FPLS functionality . . . 64
D.4 Detailed design of the FPLSR functionality . . . 65

1 Introduction

Functional data appears naturally in many different areas of application. For example, in biology, functional observations appear in a wide array of topics, from fertility rates (number of descendants per female) in insects [1] to temporal gene expression [2]. In medicine, studies such as muscle-joint coordination [3], heart rate analysis [4] or human growth [5] also rely heavily on functional data analysis. In economics, it has been successfully applied to the analysis of cash flow [6] and the price dynamics of online auctions [7]. In linguistics, speech analysis research has employed functional data analysis to study speech production variability [8] or tongue movements [9].

Functional data analysis (FDA) is the branch of statistics that studies quantities that depend on continuous parameters. In contrast to multivariate data, in which each observation consists of a finite number of components, a functional observation can take values over an infinite parameter space. This entails some complications. For instance, since the random process can only be sampled at discrete points, some information loss is unavoidable. One of the hypothesis that help in this situation is the assumption that the sampled trajectories are continuous and, in some cases, have a certain degree of underlying smoothness. Working under this assumption, the trajectories can be evaluated at any point within their domain by interpolating from the measured values.

Additionally, since the measurements at nearby points convey similar information, they cannot be treated as independent variables. The high redundancy can lead to multicollinearity issues when standard multivariate methods for statistical analysis and machine learning are directly applied to such data. In these situations, a solution is to apply dimensionality reduction techniques. The functional versions of these techniques provide a conversion from functional to multivariate observations. In particular, we will cover two of these methods: principal component analysis (PCA) and partial least squares (PLS). The difference between them lies in how they select the new multivariate variables (components). PCA seeks to maximize the variance of the components, while PLS works with two blocks of variables and tries to maximize the covariance between the components of each block.

The aim of this work is to employ PCA and PLS to solve functional linear regression problems. Linear regression is one of the cornerstones of statistics. However, its application to functional data requires some adaptations to avoid the multicollinearity issues mentioned before. The use of a basis expansion or principal components in regression problems has been explored extensively (see [10] or [11]). However, the development of partial least squares (PLS) is more novel. The first definition of functional PLS was presented in [12], while a comparison with the principal components approach was conducted in [13].

The wide rage of applications in which functional data appears has spurred the development of computational tools for FDA, with the R packages *fda* [14] and *fda.usc* [15] being some of the most complete ones. However, there are few libraries available in Python, with limited functionality. Given the widespread adoption of Python in data analysis and machine learning projects, it is important to develop computation tools that offer functionality comparable to these R packages in this language. The *scikit-fda* library seeks to fulfill this purpose while providing a seamless integration with *scikit-learn* and extending the capabilities of the scientific Python ecosystem .

1.1. Goals and Scope

The main goal of this project is to extend the functionality available in the Python library *scikit-fda*, adding functional regression methods based on dimensionality reduction techniques. The two dimensionality reduction techniques chosen are principal component analysis and partial least squares.

The *scikit-fda* library is an open source project with a strong focus on code quality. Rigorous coding standards, style guides and code review process were followed to ensure consistency and readability across the entire codebase. Code performance and correctness were ensured by means of extensive testing, incorporated in a continuous integration and testing workflow. Finally, since the library is intended for the entire FDA research community, the documentation had to not only describe the software but also provide clear and concise examples of its usage.

1.2. Structure of the document

The main section of this document is divided into four chapters that follow this introduction. In Chapter 2, the theoretical framework for the new functionality is described, along with a description of the other available libraries and *scikit-fda* itself.

In Chapter 3, the software development process is described, from the analysis phase to the implementation and validation. Three examples using the new functionality are presented in Chapter 4. Finally, Chapter 5 closes the main section with the conclusions and some remarks about future related work.

Aside from the main document, four appendices are included. The first two aim to provide additional context for multivariate PLS (Appendix A) and roughness penalty techniques (Appendix B). The last two include the Gantt diagram (Appendix C) and additional design diagrams (Appendix D).

2

State of the Art

In functional data analysis (FDA) each sample is a trajectory that has been obtained from a random process $t \to X(t)$, where t takes values in $\mathcal{T} \subseteq \mathbb{R}$. We will denote the number of samples as N. Then, the collections of trajectories sampled can be denoted as $\{x_i\}_{n=1}^{N}$. In most cases, these samples have been observed in a grid of M points $\mathbf{t} = (t_1, \ldots, t_M)^\intercal$. Then, each sample is characterized by its values at these points. Therefore, it can be expressed as a vector $x_i(\mathbf{t}) = (x_i(t_1), \ldots, x_i(t_M))^\intercal$. In the following, we will assume that M is large enough so that the functional character of the samples is apparent. The opposite case is covered by sparse functional data analysis techniques [10, Chap. 7].

One of the objectives of FDA is to study how multivariate statistical methods can be adapted to the functional setting. In particular, the goal of the present work is to take advantage of dimensionality reduction techniques to address functional linear regression problems. When dealing with functional data, the conventional multivariate models have to be adapted. The noise is still assumed to be independent to the regressor variable. However, depending on whether the predictor and predicted variables are multivariate or functional, different models (see [10, Chap. 4]) should be used. Let us consider the scalar-on-function model:

$$Y = \int_{\mathcal{T}} \beta(t) X(t)\, dt + \varepsilon. \tag{2.1}$$

In this model, the predictor variable (X) is an stochastic process, the predicted variable (Y) is a scalar, and the noise ε is scalar and independent from X. Therefore, the observations used to fit this model are N pairs: $\{(x_i, y_i)\}_{i=1}^{N}$, where the first component corresponds to a trajectory ($t \to x_i(t)$), while the second is an scalar.

A problem where this model could be applied is predicting the fat content of meat samples from its absorbance curves. In the Tecator dataset (fetched from `http://lib.stat.cmu.edu/datasets/tecator`), these data are provided for $N = 215$ meat samples. A plot of the absorbance curves is included in Figure 2.1.

The parameter that must be fitted in model (2.1) is a function β. Therefore, our goal is to obtain a function $\hat{\beta} \in L^2(\mathcal{T})$ such that the model is able to predict the values of Y accurately. For example, the least squares criterion consists of finding $\hat{\beta}$ such that the variance of ε is minimal.

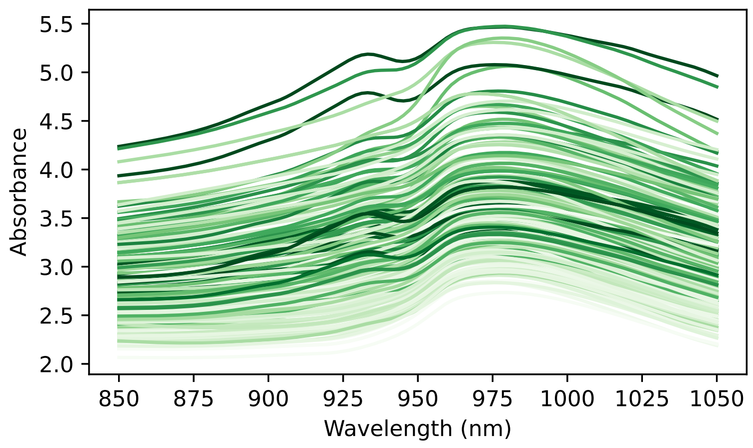


Figure 2.1: Tecator dataset. Each trajectory represents the absorbance of a sample depending on the wavelength. The saturation of the color corresponds to its fat content.

The most straightforward approach to find β would be to discretize the functions in a grid of points $\{t_m\}_{m=1}^M$ and use some integration weights $\{a_m\}_{m=1}^M$ to estimate the integral (see [16, sec. 5.2]). This would lead to the following set of N equations (one for each sample):

$$y_i = \sum_{m=1}^{M} a_m x_i(t_m)\beta(t_m) + \varepsilon_i, \quad i = 1, \dots, N. \tag{2.2}$$

To understand how these equations relate to a multivariate model, it is insightful to consider the following definitions:

$$\mathbf{X} = \begin{pmatrix} x_1(t_1) & \dots & x_1(t_M) \\ \vdots & \ddots & \vdots \\ x_N(t_1) & \dots & x_N(t_M) \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta(t_1) \\ \vdots \\ \beta(t_M) \end{pmatrix}, \quad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_N \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}. \tag{2.3}$$

If we also define $\mathbf{A} = \text{diag}(a_1, \dots, a_M)$ a diagonal matrix, the equations can be expressed as $\mathbf{y} = \mathbf{XA}\boldsymbol{\beta} + \boldsymbol{\epsilon}$. In light of this expression, it is immediate that the equations in (2.2) can be fitted using traditional multivariate methods by means of a simple variable change considering $\tilde{\mathbf{X}} = \mathbf{XA}$. For example, the ordinary least squares estimator applied to this problem is $\hat{\boldsymbol{\beta}} = \mathbf{A}^{-1}(\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y}$. However, this approach has several shortcomings.

Firstly, since the trajectories of X are assumed to be smooth, the $\{X(t_m)\}_{m=1}^M$ variables are not independent and collinearity issues can arise. Secondly, if we use fine grids, as soon as $M > N$, the range of $\mathbf{X}$ is less than M and, as a result, $\mathbf{X}^\intercal\mathbf{X}$ has no inverse. This can be overcome using the Moore-Penrose inverse. However, since in this situation there are more unknowns than equations, several possible $\hat{\beta}$ fit the data perfectly. Furthermore, the resulting $\hat{\beta}$ functions can be very noisy, which is generally an undesirable trait. Since we have assumed that the random trajectories are smooth, it is to be expected that their influence on the response should also be smooth when expressed as a function of t. In order to obtain smooth solutions, roughness penalizations can be incorporated, using the methods outlined in Appendix B.

An alternative approach is to express the functions in a certain basis and work with their coefficients in said basis. By choosing a finite basis, the search for the functional parameter $\beta(t)$ is converted into a search on a multivariate space. However, in doing so, the functional space for the search of the parameter is restricted to the span of the basis functions. In particular, if we choose a basis $\{\phi_k\}_{k=1}^K$, we can express the trajectories of X in that basis using K coefficients for each trajectory. To do so, we define $\{\xi_{ik}\}_{k=1}^K$ as the coefficients of the i-th sample. Similarly, we consider $\{b_k\}_{k=1}^K$ the coefficients of β. Therefore, we have the following expansions in terms of the basis functions:

$$x_i(t) = \sum_{k=1}^{K} \xi_{ik}\phi_k(t), \qquad \beta(t) = \sum_{k=1}^{K} b_k\phi_k(t). \tag{2.4}$$

These coefficients can now be arranged into matrices in a similar way as in (2.3). Note that we will maintain the same notation for the matrices and vectors as in (2.3) even though the definitions are slightly different.

$$\mathbf{X} = \begin{pmatrix} \xi_{11} & \dots & \xi_{1K} \\ \vdots & \ddots & \vdots \\ \xi_{N1} & \dots & \xi_{NK} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} b_1 \\ \vdots \\ b_M \end{pmatrix}, \quad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_N \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}. \tag{2.5}$$

To express (2.1) using a basis expansion in a matrix form, we must first define the inner product matrix of the basis. Consider $g_{ij} = \int \phi_i(t)\phi_j(t)dt$. Arranging these values in a matrix one obtains $\mathbf{G} = (g_{ij})$, the inner product matrix of the basis. Then, from the model in (2.1), we obtain that, for each $n = 1, \dots, N$:

$$\begin{aligned} y_n &= \int \left(\sum_{i=1}^{K} x_{ni}\,\phi_i(t) \sum_{j=1}^{K} b_j\phi_j(t)dt \right) + \varepsilon_n = \\ &= \sum_{1\le i,j\le K} x_{ni}b_j \int \phi_i(t)\phi_j(t)dt + \varepsilon_n = \sum_{1\le i,j\le K} x_{ni}b_j g_{ij} + \varepsilon_n \end{aligned} \tag{2.6}$$

Expressing these equations in a matrix form, one obtains $\mathbf{y} = \mathbf{XG}\boldsymbol{\beta} + \boldsymbol{\epsilon}$. As with the discretized representation, this model can be fitted using multivariate techniques by means of the change of variable $\tilde{\mathbf{X}} = \mathbf{XG}$. Using ordinary least squares again, one can obtain $\boldsymbol{\beta} = \mathbf{G}^{-1}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$.

Note that the definition of $\mathbf{G}$ also provides a simple approach to calculate the basis coefficients using the formula in (2.7). In order to apply this formula, $\mathbf{G}$ has to be invertible, which holds true for all inner product matrix. As long as the basis functions are linearly independent, $\mathbf{G}$ must have maximum range.

$$\begin{pmatrix} \xi_{n,1} \\ \vdots \\ \xi_{n,K} \end{pmatrix} = \mathbf{G}^{-1} \begin{pmatrix} \int x_n(t)\phi_1(t)dt \\ \vdots \\ \int x_n(t)\phi_K(t)dt \end{pmatrix}. \tag{2.7}$$

Note that, sometimes, an orthonormal basis is chosen so that $\mathbf{G} = \mathbf{I}_K$. On the contrary, it is also possible to use two different basis to represent X and β. In that case, the uses of **G** in (2.6) are replaced by the inner product matrix between the two basis.

One advantage of this basis expansion is that, as long as $\{\phi_k\}_{k=1}^K$ are smooth, it is possible to obtain a smooth approximation of the desired functions with fewer coefficients than in the discretized approach. Therefore, the issues in the discretized approach caused by K being larger than N do not usually arise using a basis expansion.

However, in order to take advantage of this basis expansion, one must first decide which basis to use. One possibility is to employ Fourier or B-Spline basis (see [10, Sec. 1.1]). These are generic basis that are generally able to provide good approximations for arbitrary functions when sufficient basis functions are considered. However, there is no guarantee that the most important information for the regression model will be encoded in the first K coefficients of the corresponding basis expansion. An alternative is to look for a basis that is adapted to the problem and captures most of the relevant information of the data in a few components. This is the role that dimensionality reduction techniques fulfill. Dimensionality reduction techniques can be defined as methods that obtain projections of the original data onto fewer dimensions, while losing little information. In the functional setting, these directions are functions. Moreover, this set of directions extracted constitutes the basis we are looking for.

In this document, two dimensionality reduction techniques will be considered. The first of them, principal component analysis (PCA) corresponds to summarizing the variance information in X. On the contrary, partial least squares (PLS) seeks to maximize the covariance information between X and Y present in the components. These two methods will be covered in detail in Sections 2.1 and 2.2.

2.1. Principal Component Analysis

The multivariate version of PCA aims to find orthogonal directions such the projections of the data along them have the highest variance. Formally, let **X** be the $N \times M$ data matrix (N samples and M features) and $\mathbf{w}_l$ the l-th projection direction. Then the optimization problem to solve is:

$$\begin{aligned} \mathbf{w}_l &= \max_{\mathbf{w}} \arg \left(\mathrm{var}(\mathbf{X}\mathbf{w}) \right) \text{ subject to} \\ \|\mathbf{w}\| &= 1 \\ \mathbf{w} &\perp \mathbf{w}_j \quad \text{for all } j < l \end{aligned} \tag{2.8}$$

Moreover, since all data are centered, the variance of the projections can easily be calculated as

$\mathrm{var}(\mathbf{Xw}) = \frac{1}{N}\mathbf{w}^\intercal\mathbf{X}^\intercal\mathbf{Xw}$. Additionally, it can be proven (see [11, p. 153]) that the desired projection directions are the eigenvectors of the covariance matrix $\mathbf{X}^\intercal\mathbf{X}$.

This criterion can be easily extended for functional data by changing the inner product to the L^2 inner product

$$\begin{aligned} w = \max_{w} \arg &\left(\mathrm{var}\left(\int X(t)w(t)\,dt\right)\right) \text{ subject to} \\ &\|w\|_{L^2} = 1 \\ &w \perp_{L^2} w_j \quad \text{for all } j < l, \end{aligned} \tag{2.9}$$

where $\|h\|_{L^2} = \left(\int h(t)^2 dt\right)^2$ denotes the L^2 norm and $\perp_{L^2}$ denotes orthogonality with respect to the L^2 inner product. However, the way in which the L^2 products should be estimated depends on the chosen representation of the functional data. As we introduced at the beginning of this chapter, there are two ways to represent functional data: as a basis expansion or discretized in a grid. The integrals of the L^2 inner product are calculated in a different way in each of these two situations

When the functions are discretized in a grid: $\{t_m\}_{m=1}^M$, an integral can be approximated with integration weights: $\{a_m\}_{m=1}^M$. These integration weights are then arranged diagonally in a matrix **A**. One example of these weights is the Simpson quadrature [16, p. 257]. Moreover, it is also convenient to define **L** such that $\mathbf{L}\mathbf{L}^\intercal = \mathbf{A}$. Using these definitions, the integrals in (2.9) can be approximated:

$$\mathrm{var}\left(\int X(t)w(t)\,dt\right) = \mathrm{var}\left(\sum_{m=1}^{M} X_m w_m a_m\right) = \frac{1}{N}\mathbf{w}^\intercal\mathbf{A}\mathbf{X}^\intercal\mathbf{X}\mathbf{A}\mathbf{w}. \tag{2.10a}$$

$$\|w\|_{L^2} = \left(\int w(t)^2 dt\right)^{1/2} = \left(\sum_{m=1}^{M} w_m w_m a_m\right)^{1/2} = (\mathbf{w}^\intercal\mathbf{A}\mathbf{w})^{1/2} = \|\mathbf{L}^\intercal\mathbf{w}\|. \tag{2.10b}$$

Therefore, it easily follows that the functional problem can solved by solving the multivariate problem with a different data matrix. Let $\tilde{\mathbf{w}}_l$ be the solutions to the multivariate problem with the data matrix $\tilde{\mathbf{X}} = \mathbf{XA}(\mathbf{L}^\intercal)^{-1}$. Then, $\mathbf{w}_l = (\mathbf{L}^\intercal)^{-1}\tilde{\mathbf{w}}_l$ corresponds to the desired w_l functions discretized in the grid.

When the functional data are given in a basis representation, a similar calculation can be carried out. However, in that case, the integration weights will be replaced by the inner product matrices of the basis involved. In particular, it is possible to use a different basis for X and w. In that case, two different matrices are required. Let **G** be the inner product matrix of the basis used to express w and, **J**, the inner product matrix between the basis for X and the basis for w. In this case, **L** is defined such that $\mathbf{L}\mathbf{L}^\intercal = \mathbf{G}$. We can then calculate $\tilde{\mathbf{w}}_l$, the solutions to the multivariate problem with the data matrix $\tilde{\mathbf{X}} = \mathbf{XJ}(\mathbf{L}^\intercal)^{-1}$. Then, $\mathbf{w}_l = (\mathbf{L}^\intercal)^{-1}\tilde{\mathbf{w}}_l$ corresponds to coefficients of the sought $\{w\}_{l=1}^L$ functions in the selected basis.

A detailed discussion of this whole procedure in terms of operator and eigenfunctions can be found in the eight chapter of [11].

2.1.1. Regularized Principal Components

Regularized Principal Components is a variation of the FPCA method described before that uses a different maximization criteria. Standard FPCA seeks components with the maximum variance. However, regularized PCA seeks smooth components with the maximum variance. In order to do so, a roughness penalty term (see Appendix B) is added to the maximization criterion.

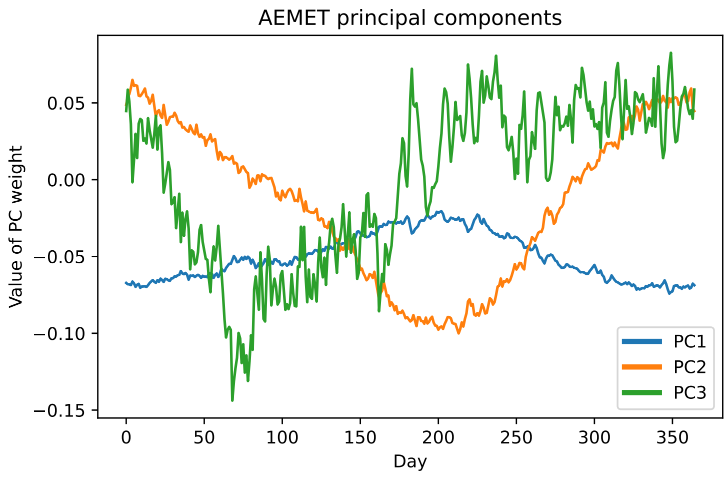


Figure 2.2: FPC projection directions for the temperature readings in the AEMET dataset from [15]

This is necessary in some cases since applying standard FPCA can lead sometimes to noisy or spiky projection directions. An example of this situation can be seen in Figure 2.2. In that case, the evolution of the average temperatures during a year in different weather stations has been analyzed. To see why the roughness of the obtained weight functions may be undesirable, it is useful to consider the following notion. In [11, p. 154], it is explained that the magnitude of a weight function in a given point can be interpreted as the contribution of that point to that principal component.

With this intuition in mind, one can interpret the weight plots in Figure 2.2. In particular, the third principal component is particularly noisy. The spikes indicate that data that corresponds to almost consecutive days have a very different significance. Considering that the average temperatures should evolve smoothly across the year, this indicates that the information is being extracted from random noise in the measurements, rather that from the original data.

The regularized version of FPCA is based on defining the notion of a penalized variance. While non-regularized FPCA seeks components with the maximum variance, when regularization is applied, the algorithm maximizes the penalized variance instead. Given a linear differential operator P and a regularization parameter λ, the penalized variance is defined as

$$\mathrm{PVAR}(w) = \frac{\mathrm{var}\left(\int X(t)w(t)dt\right)}{\|w\|_{L^2}^2 + \lambda\,\mathrm{PEN}_P(w)}, \tag{2.11}$$

where PEN_P is the penalization of P as defined in Appendix B.

If we are working with a discretized representation, by substituting here equations (2.10a), (2.10b) and (B.14), the variance, norm and penalization can be estimated. Additionally, we can consider the Cholesky decomposition of the matrix $\mathbf{S} + \lambda\mathbf{P}$. That is to say a matrix $\mathbf{L}$ such that $\mathbf{L}\mathbf{L}^\intercal = \mathbf{S} + \lambda\mathbf{P}$. By also considering the variable change $\mathbf{w} = (\mathbf{L}^\intercal)^{-1}\tilde{\mathbf{w}}$, one can rewrite (2.11) as

$$N \cdot \mathrm{PVAR}(w) = \frac{\mathbf{w}^\intercal\mathbf{S}\mathbf{X}^\intercal\mathbf{X}\mathbf{S}\,\mathbf{w}}{\mathbf{w}^\intercal\mathbf{S}\mathbf{w} + \lambda\,\mathbf{w}^\intercal\mathbf{P}\mathbf{w}} = \frac{\mathbf{w}^\intercal\mathbf{S}\mathbf{X}^\intercal\mathbf{X}\mathbf{S}\,\mathbf{w}}{\mathbf{w}^\intercal(\mathbf{S}+\lambda\mathbf{P})\mathbf{w}} = \frac{\mathbf{w}^\intercal\mathbf{S}\mathbf{X}^\intercal\mathbf{X}\mathbf{S}\,\mathbf{w}}{\mathbf{w}^\intercal\mathbf{L}\mathbf{L}^\intercal\mathbf{w}} = \frac{\tilde{\mathbf{w}}^\intercal\mathbf{L}^{-1}\mathbf{S}\mathbf{X}^\intercal\mathbf{X}\mathbf{S}(\mathbf{L}^{-1})^\intercal\,\tilde{\mathbf{w}}}{\tilde{\mathbf{w}}^\intercal\tilde{\mathbf{w}}}. \tag{2.12}$$

And that problem is equivalent to maximizing the numerator subject to the denominator being 1. This is easy to see as multiplying $\tilde{\mathbf{w}}$ by any constant does not change the result. Therefore, the PCA penalized problem in the grid representation can be reduced to the standard multivariate version. Let $\tilde{\mathbf{w}}_l$ be the solutions to the multivariate problem with the data matrix $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{S}(\mathbf{L}^\intercal)^{-1}$. Then $\mathbf{w}_l = (\mathbf{L}^\intercal)^{-1}\tilde{\mathbf{w}}_l$ is the solution to the regularized problem.

A very similar derivation can be carried out when the functions are expressed in a basis. This argument is available in [11, p. 181]. The only difference with the discretized version is the definitions of $\tilde{\mathbf{X}}$ and $\mathbf{L}$. In this case $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{J}(\mathbf{L}^\intercal)^{-1}$ and $\mathbf{L}\mathbf{L}^\intercal = \mathbf{G} + \lambda\mathbf{P}$, where $\mathbf{G}$ is the inner product matrix of the basis of the weights and $\mathbf{J}$ is the inner product matrix of the basis for X and the basis of the weights. Additionally, $\mathbf{P}$ represents now the basis version of the penalty matrix.

There is a major side effect of the variable change for $\mathbf{w}$. While standard FPCA leads to orthogonal weight functions, this property is lost when applying regularization. This is due to the fact that this variable change is actually changing the inner product with respect to which the directions are calculated. The weights are orthogonal with respect to this new inner product:

$$< w_i, w_j >_R = \int w_i(s)w_j(s)ds + \int P(w_i)(s)P(w_j)(s)ds = 0, \quad i \neq j. \tag{2.13}$$

2.1.2. Functional Principal Components (FPC) Regression

FPC regression is a regression technique that takes advantage of the FPCA methods to perform functional regression using the principal component basis. This method can be decomposed in two steps

Firstly, FPCA is carried out with the data available for the X stochastic process. In order to do so, any of the techniques in this section can be utilized. This step will result in obtaining a basis $\{\phi_k\}_{k=1}^K$.

Secondly, a linear regression model is fitted using the components. These components are the coefficients of the observed functional data in the basis $\{\phi_k\}_{k=1}^K$. After obtaining these coefficients, following the steps in (2.6), the problem is reduced to a multivariate regression.

2.2. Partial Least Squares (PLS)

Partial least squares (PLS) as a dimensionality reduction method is a particular case of the NIPALS (nonlinear iterative partial least squares) algorithm introduced in [17]. However, nowadays, outside of certain areas such as economics or chemometrics, these two terms are used almost interchangeably. In short, the algorithm used to solve the PLS problem is generally called NIPALS even if it is a special case of the original NIPALS algorithm

The goal in PLS is to extract two sets of weights, one for the X block and one for the Y block: $\{\mathbf{w}_l\}_{l=1}^L$ and $\{\mathbf{c}_l\}_{l=1}^L$. The components are defined as the projections of the data along these directions. That is to say, the components are $\mathbf{t}_l = \mathbf{X}\mathbf{w}_l$ and $\mathbf{u}_l = \mathbf{Y}\mathbf{c}_l$. These directions are selected such that all components of the same block are incorrelated:

$$\operatorname{corr}(\mathbf{t}_i, \mathbf{t}_j) = 0 \qquad \operatorname{corr}(\mathbf{u}_i, \mathbf{u}_j) = 0 \quad \text{if } i < j \tag{2.14}$$

Moreover, the weights are chosen to maximize $\operatorname{cov}^2(\mathbf{t}_l, \mathbf{u}_l)$. However, before defining how they are chosen, there are a couple of caveats that one should keep in mind. Firstly, this method is iterative in nature. From its initial conception, it has relied on applying a maximization criterion, modifying the data matrices and applying that said criterion again. Therefore, it is not possible to characterize the result of PLS as simply as in PCA.

Secondly, there are several versions of PLS. All of them share the same maximization goal but they establish different constraints between the components extracted in each iteration, leading to different residual matrices. This, in turn, leads to different final results. In the present work, we shall focus on the PLS methods derived from the NIPALS algorithm. However, there are other alternatives such as SIMPLS (see [18]) or PLS-SVD (see [19]).

Appendix A provides an introduction to the multivariate PLS methods for the readers unfamiliar with PLS. On the contrary, the following sections focus on the definitions of the functional method. In Section 2.2.1, PLS as a dimensionality reduction technique is introduced. Section 2.2.2 covers the incorporation of regularization into the PLS method and Section 2.2.3 covers its application to regression problems.

2.2.1. Functional Partial Least Squares (FPLS) for dimensionality reduction

There are several alternative formulations of functional PLS. On the one hand, it is possible to adapt the NIPALS algorithm to accept functional data. This approach has been covered in [12] and [13]. Additionally, there have been some efforts that rely on using a slightly different definition of PLS such as [20].

For our goals, the extension of NIPALS for functional data is more convenient. We will see how NIPALS is even able to mix functional and multivariate data. That is to say, one block (X or Y) can be functional, while the other can still be treated as multivariate data.

Moreover, adapting NIPALS to handle functional data will be rather easy. In fact, as long as no regularization is applied, it is sufficient to add the corresponding integration weights or inner product matrices in the correct places. The functional version of NIPALS has been included in Algorithm 2.1. The inner section of the `while` loop can be divided in the following steps:

1. **Weights calculation**. To simplify the analysis, assume that only the X block is functional. Then, the weight of the X block is a function $w(\cdot)$. As we know, the goal of NIPALS is to maximize the covariance between the projections. In this case, the projection in the X block is performed using the L^2 inner product. To reduce this situation to the multivariate case, the same logic as in PCA must be applied. To do so, the inner product matrices must be incorporated into the calculation.

   - $\mathbf{G}_{xw}$ represents the inner product matrix between the X process and the weight functions (basis representation) or the integration weights (grid representation)
   - $\mathbf{G}_{ww}$ represents the inner product matrix for the weight functions (basis representation) or the integration weights (grid representation).

   Additionally, a Cholesky decomposition of $\mathbf{G}_{ww}$ is needed to estimate the norm, leading to the definition of $\mathbf{L}_x\mathbf{L}_x^\intercal = \mathbf{G}_{ww}$. Using this matrix, we introduce the change of variable $\tilde{\mathbf{w}} = \mathbf{L}_x^\intercal\mathbf{w}$ and the covariance can be approximated as:

   $$\underset{\|w\|_{L^2}=1,\|\mathbf{c}\|=1}{\text{cov}}\left(\int X(t)w(t)dt, \mathbf{c}^\intercal Y\right) = \underset{\|\tilde{\mathbf{w}}\|=1,\|\mathbf{c}\|=1}{\text{cov}}\left(\tilde{\mathbf{w}}^\intercal\mathbf{L}_x^\intercal\mathbf{G}_{xw}X, \mathbf{c}^\intercal Y\right). \tag{2.15}$$

   Looking at the right side of this expression, the problem is reduced to the multivariate problem with the X matrix $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{G}_{xw}\mathbf{L}_x^{-1}$. Note that the normalization included in lines 8 and 9 in Algorithm 2.1 is only needed when regularization is applied (see Section 2.2.2). If both blocks are functional, the analogous $\mathbf{G}_{yc}, \mathbf{G}_{cc}$ and $\mathbf{L}_y$ can be defined. On the contrary, if these matrices are set to the identity matrix, the multivariate behaviour is recovered.

2. **Scores calculation**. The scores are defined as the projection of the data along the weights. When the data are functional, we must use the L^2 inner product. For example, if the X block is functional, its score ($\mathbf{t} = (t_1, \ldots, t_N)^\intercal$) can be calculated as:

   $$t_n = \int X_n(t)w(t)dt = (\mathbf{X}\mathbf{G}_{xw}\mathbf{w})_n \tag{2.16}$$

3. **Choose latent variable for Y**. A different deflation scheme must be used when NIPALS is applied to regression problems. In those problems, the loading calculation of the Y block is performed with respect to the X scores. The reasons behind this are covered in Appendix A.

4. **Loading calculation**. The loadings are calculated by fitting a linear regression. If the X block is functional, its loadings will be a function instead of a vector. The model to fit is

$$\mathbf{X}(t) = \mathbf{t}\mathbf{p}(t)^{\mathsf{T}} + \mathbf{E}, \tag{2.17}$$

where $\mathbf{X}(t) = (X_1(t), \dots X_N(t))^{\mathsf{T}}$ and $\mathbf{p}(t) = (p_1(t), \dots, p_N(t))^{\mathsf{T}}$. Therefore, for each t, we still have to fit the same model as in the multivariate case. This proves that the same formula is valid when the functions are discretized. Using the linearity of the basis expansion, it is easy to prove that this also holds in that case.

5. **Data deflation**. The new data matrices for the next iterations are the residuals of the linear models fitted for the loading calculations.

```
$\mathbf{X}_0 \leftarrow \mathbf{X}, \quad \mathbf{Y}_0 \leftarrow \mathbf{Y}$
$l \leftarrow 1$
while $l < L$ do
    # Weights calculation
    $\tilde{\mathbf{X}}_{l-1} \leftarrow \mathbf{X}_{l-1}\mathbf{G}_{xw}\mathbf{L}_x^{-1}$
    $\tilde{\mathbf{Y}}_{l-1} \leftarrow \mathbf{Y}_{l-1}\mathbf{G}_{yc}\mathbf{L}_y^{-1}$
    $\mathbf{w}_l \leftarrow \mathbf{L}_x^{-1}\Big($normalized dominant eigenvector of $\tilde{\mathbf{X}}_{l-1}^{\mathsf{T}}\tilde{\mathbf{Y}}_{l-1}\tilde{\mathbf{Y}}_{l-1}^{\mathsf{T}}\tilde{\mathbf{X}}_{l-1}\Big)$
    $\mathbf{c}_l \leftarrow \mathbf{L}_y^{-1}\tilde{\mathbf{Y}}_{l-1}\tilde{\mathbf{X}}_{l-1}\mathbf{w}$

    # Weights normalization (only needed with regularization)
    $\mathbf{w}_l \leftarrow \mathbf{w}_l/(\mathbf{w}_l^{\mathsf{T}}\mathbf{G}_{ww}\mathbf{w}_l)$
    $\mathbf{c}_l \leftarrow \mathbf{c}_l/(\mathbf{c}_l^{\mathsf{T}}\mathbf{G}_{cc}\mathbf{c}_l)$

    # Scores calculation
    $\mathbf{t}_l \leftarrow \mathbf{X}_{l-1}\mathbf{G}_{xw}\mathbf{w}_l$
    $\mathbf{u}_l \leftarrow \mathbf{Y}_{l-1}\mathbf{G}_{xw}\mathbf{c}_l$

    # Choose latent variables for Y
    if deflation_mode == regression then
        $\mathbf{d} \leftarrow \mathbf{t}_l$
    else
        $\mathbf{d} \leftarrow \mathbf{u}_l$
    end

    # Loadings calculation
    $\mathbf{p}_l \leftarrow \mathbf{X}_{l-1}^{\mathsf{T}}\mathbf{t}_l/(\mathbf{t}_l^{\mathsf{T}}\mathbf{t}_l)$
    $\mathbf{q}_l \leftarrow \mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{d}/(\mathbf{d}^{\mathsf{T}}\mathbf{d})$

    # Data deflation
    $\mathbf{X}_l \leftarrow \mathbf{X}_{l-1} - \mathbf{t}_l\mathbf{p}_l^{\mathsf{T}}$
    $\mathbf{Y}_l \leftarrow \mathbf{Y}_{l-1} - \mathbf{d}\mathbf{q}_l^{\mathsf{T}}$
    $l \leftarrow l + 1$
end
```

Algorithm 2.1: NIPALS algorithm for FPLS

The rotations of functional blocks are functions as well. For simplicity, if the functional block is using the basis representation, the rotations will also be represented as a basis expansion using the same basis as the weights. Similarly, if the weights are discretized, the rotations will be handled in the same manner. Conveniently, using the definitions of $\mathbf{G}_{xw}$ and $\mathbf{G}_{yc}$, the coefficient or grid discretization of the rotations (using the canonical deflation mode) can be calculated as:

$$\mathbf{R}_x = \mathbf{W}(\mathbf{P}^\mathsf{T}\mathbf{G}_{xw}\mathbf{W})^{-1} \quad \mathbf{R}_y = \mathbf{C}(\mathbf{Q}^\mathsf{T}\mathbf{G}_{yc}\mathbf{C})^{-1} \tag{2.18}$$

2.2.2. FPLS Regularization

In order to obtain smooth rotation functions, a very similar approach as in FPCA is followed. This can be done since smoothing the weights results in smooth rotation functions. Therefore, we need only to modify the weights calculation (first step in the loop of NIPALS). The goal is to change the maximization criterion so that the roughness of the weights is taken into account.

The roughness of the weights is estimated as the norm of a differential linear operator P applied to them. With this definition in mind the penalized covariance can be defined. This is the value that must be maximized during the weight calculation. Its definition depends on whether a single block is functional or both blocks are functional. If only the X block is functional, it is defined as

$$\mathrm{PCOV}_x(w, \mathbf{c}) = \frac{\mathrm{cov}^2\left(\int X(t)w(t)dt, \mathbf{c}^\mathsf{T}Y\right)}{\|w\|^2_{L^2} + \lambda \mathrm{PEN}_P(w)} \tag{2.19}$$

Following the same procedure outlined in Section 2.1.1, the following sample version can be obtained:

$$N \cdot \mathrm{PCOV}_x(w, \mathbf{c}) = \frac{\mathbf{w}^\mathsf{T}\mathbf{G}^\mathsf{T}_{xw}\mathbf{X}^\mathsf{T}\mathbf{Y}\mathbf{c}\mathbf{c}^\mathsf{T}\mathbf{Y}^\mathsf{T}\mathbf{X}\mathbf{G}_{xw}\mathbf{w}}{\mathbf{w}^\mathsf{T}(\mathbf{G}_{ww} + \mathbf{P})\mathbf{w}}, \tag{2.20}$$

where $\mathbf{G}_{xw}$ is defined in the same way as in the previous section and $\mathbf{w}$ is the discretization or basis coefficients of w (depending on the representation considered).

It is possible to transform this problem back into the multivariate version. To do so, it is required to modify the Cholesky decomposition, redefining $\mathbf{L}_x$ as the matrix that fulfills $\mathbf{L}_x\mathbf{L}_x^\mathsf{T} = \mathbf{G}_{ww} + \mathbf{P}$. Then, the variable change $\mathbf{w} = (\mathbf{L}_x^\mathsf{T})^{-1}\tilde{\mathbf{w}}$ can be used. As a result, the penalized covariance can be rewritten as

$$N \cdot \mathrm{PCOV}_x(w, \mathbf{c}) = \frac{\tilde{\mathbf{w}}^\mathsf{T}\mathbf{L}_x^{-1}\mathbf{G}^\mathsf{T}_{xw}\mathbf{X}^\mathsf{T}\mathbf{Y}\mathbf{c}\mathbf{c}^\mathsf{T}\mathbf{Y}^\mathsf{T}\mathbf{X}\mathbf{G}_{xw}(\mathbf{L}_x^\mathsf{T})^{-1}\tilde{\mathbf{w}}}{\tilde{\mathbf{w}}^\mathsf{T}\tilde{\mathbf{w}}}. \tag{2.21}$$

For the same reasons as with the PCA penalized version, this problem is equivalent to maximizing the numerator subject to the denominator being one. That, in turn, makes it possible to solve for $\mathbf{w}$ and $\mathbf{c}$ with the multivariate version. The result of multivariate NIPALS optimization criterion with $\mathbf{X}\mathbf{G}_{xw}(\mathbf{L}_x^\mathsf{T})^{-1}$ as the X data matrix and $\mathbf{Y}$ as the Y data matrix are two vectors $\hat{\mathbf{w}}$ and $\hat{\mathbf{c}}$ that maximize the numerator constrained to their norm being one. Thus, $\mathbf{c} = \hat{\mathbf{c}}$ and $\mathbf{w} = (\mathbf{L}_x^\intercal)^{-1}\hat{\mathbf{w}}$ maximize the penalized covariance.

This method has to be slightly adjusted if the second block is also functional. In that case, the penalized covariance is defined as

$$\mathrm{PCOV}_{x,y}(w,c) = \frac{\mathrm{cov}^2\left(\int X(t)w(t)dt, \int Y(t)c(t)dt\right)}{(\|w\|^2_{L^2} + \lambda\mathrm{PEN}_{P_x}(w))(\|c\|^2_{L^2} + \lambda\mathrm{PEN}_{P_y}(c))}, \tag{2.22}$$

where P_x and P_y are two linear differential operators that measure the roughness of the weights.

The same procedure as before can be carried out in this case. Then, one obtains that this penalized covariance can be calculated in a similar manner:

$$N \cdot \mathrm{PCOV}_{x,y}(w,c) = \frac{\tilde{\mathbf{w}}^\intercal \mathbf{L}_x^{-1}\mathbf{G}_{xw}\mathbf{X}^\intercal\mathbf{Y}\mathbf{G}_{yc}(\mathbf{L}_y^{-1})^\intercal\mathbf{c}\mathbf{c}^\intercal\mathbf{L}_y^{-1}\mathbf{G}_{yc}^\intercal\mathbf{Y}^\intercal\mathbf{X}\mathbf{G}_{xw}(\mathbf{L}_x^\intercal)^{-1}\tilde{\mathbf{w}}}{(\tilde{\mathbf{w}}^\intercal\tilde{\mathbf{w}})(\tilde{\mathbf{c}}^\intercal\tilde{\mathbf{c}})}, \tag{2.23}$$

where two Cholesky decompositions have been carried out: $\mathbf{L}_x\mathbf{L}_x^\intercal = \mathbf{G}_{ww} + \mathbf{P}_x$ and $\mathbf{L}_y\mathbf{L}_y^\intercal = \mathbf{G}_{cc} + \mathbf{P}_y$. Additionally, two variable changes have been made: $\tilde{\mathbf{w}} = (\mathbf{L}_x^\intercal)^{-1}\mathbf{w}$ and $\tilde{\mathbf{c}} = (\mathbf{L}_y^\intercal)^{-1}\mathbf{c}$.

From expression (2.23), following the same steps as before, the problem can be solved using the standard multivariate PLS maximization criterion. Moreover, using the definitions of the inner product matrices $\mathbf{G}_{xw}, \mathbf{G}_{ww}, \mathbf{G}_{yc}, \mathbf{G}_{cc}$, we have reached a procedure that can be applied both to the discretized and basis expansion representations.

2.2.3. FPLS Regression

FPLS regression employs the same techniques as the PLS multivariate regression covered in Appendix A. However, in this case, the response, the explanatory variable or both can be functional data. Even though this would seem to complicate the calculations immensely, this method is quite simple.

The main idea of the multivariate method was to calculate the approximation of the response variable Y in two steps. First, use the rotations ($\mathbf{R}_x$) to obtain $\mathbf{T}$ from $\mathbf{X}$ as $\mathbf{T} = \mathbf{X}\mathbf{R}_x$. Then, use the loadings ($\mathbf{Q}$) to approximate $\mathbf{Y}$ from $\mathbf{T}$ as $\hat{\mathbf{Y}} = \mathbf{T}\mathbf{Q}^\intercal$. Therefore, $\hat{\mathbf{Y}} = \mathbf{X}\mathbf{R}_x\mathbf{Q}^\intercal$

One interesting observation is that, in NIPALS, regardless of whether X, Y or both are functional data, the extracted components (or scores) are still vectors (samples of a scalar variables). As a result, when X is functional data, the only difference is that $\mathbf{R}_x$ is now also functional data and the inner product matrix has to be used: $\mathbf{T} = \mathbf{X}\mathbf{G}_{xw}\mathbf{R}_x$. Thus, if Y is scalar but X is functional, $\hat{\mathbf{Y}} = \mathbf{X}\mathbf{G}_{xw}\mathbf{R}_x\mathbf{Q}^\intercal$

When Y is functional data, the expression for $\hat{\mathbf{Y}}$ does not even change, as the nature of $\mathbf{T}$ is always the same. That is to say, $\mathbf{T}\mathbf{Q}^\intercal$ is not an inner product calculation. Instead, it is the calculation of N linear combination of the loadings expressed as a matrix multiplication. Therefore, in that identity the only change is the meaning of $\mathbf{Q}$, which is now a matrix of coefficients (if the functions are expressed in a basis expansion) or discretized values (if they have been sampled in a grid).

2.3. Computational tools for FDA

The ever-growing interest in functional data analysis has resulted in the development of many FDA computational tools. In particular, the goal of this work is to extend the functionality of one of these tools: *scikit-fda*. However, before describing this Python package in detail, we will cover some of the most prevalent alternatives.

Most of the tools have been implemented in R. That is the case of *fda* [14], one of the main references in this field. This package (which is also available in Matlab), uses a basis representation of the functional data and provides an implementation of the results and methods discussed in [11]. In particular, it includes both standard functional regression and PCA functional regression. However, it does not provide any functionality related to PLS.

Another widespread R package is *fda.usc* [15]. In contrast to *fda*, this library handles data in a discretized manner. That is to say, it works with the values of the functions in some discrete grid. In addition to standard functional regression and PCA functional regression, it also provides a PLS functional regression implementation with multivariate response.

Additionally, there are other packages that focus on specific problems. For example the package *refund* [21] is focused on solving regression problems of different kinds. However, it does not provide any functionality for PLS regression, and its implementation of FPCR is not fully complete as of the writing of this work. Another option is *pls* [22], which provides an implementation of PLS and PCA regression. Even though this implementation is much more complete, some aspects such as regularization and functional response are not covered.

The options in Python are much more limited. Two examples are *FDApy* [23] and *fdasrsf* [24]. Both of these packages include the functionality required to calculate the principal components and apply PCA to regression problems. However, their functionality does not include PLS in any of its versions.

In this context, the package *scikit-fda* [25] seeks to provide a complete implementation of FDA tools in Python. Before starting this bachelor's thesis, *scikit-fda* already provided an implementation of PCA, albeit not fully complete. The goal of this work is to complete that implementation, extend it to regression problems and implement PLS both as a dimensionality reduction technique and as a tool for regression. In the following subsection, we will explore the structure of this package.

2.3.1. The *scikit-fda* package

The goal of the *scikit-fda* [26] package is to provide functional data processing and analysis in Python. Additionally, the package is designed from the ground up to handle functional data expressed both as a basis expansion and as values discretized in a grid. It provides an extensive list of functionality, from preprocessing to classification, clustering or regression.

However, the aim of *scikit-fda* is not only to create a complete and effective tool for FDA. Great care and attention is paid to the documentation and design of the codebase, ensuring that it is easy to read, maintainable, performant and coherent. Both examples and tests are included for all functionality, making sure that the end user has enough information to use this package effectively.

Currently, *scikit-fda* has seven public modules. An overview of the package is provided in Figure 2.3. Complete descriptions for each module can be found in the documentation of the package (accessible at `https://fda.readthedocs.io/en/latest/index.html`). However, in order to provide sufficient context for the following chapter, we provide a short summary of the contents of each module.

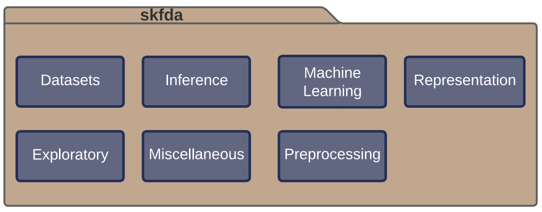


Figure 2.3: Diagram of *scikit-fda* modules

- The Datasets module enables the user to fetch datasets in a format readily compatible with the rest of the modules.
- The Inference module provides methods to derive conclusions from the sampled functional data. It also provides utilities to estimate the reliability of the results obtained.
- The Machine Learning module contains estimators that solve some of the most common machine learning problems, such as classification of data, clustering or regression.
- The Representation module contains the structures and basic functionality required to handle functional data across the entire library. Among others, this module contains the definitions of the different basis.
- The Exploratory module provides functions that can provide descriptive statistics of the functional data. Moreover, utilities for visualization and Interpretation of the data are included.
- The Miscellaneous module was created to contain general functionality that was used by other modules or features that did not fit properly in the other modules. In particular, it contains the logic used to calculate the penalization matrices for linear operators.
- The Preprocessing module provides functionality to be applied before analyzing the data. For example, smoothing, feature construction and dimensionality reduction.

The functionality available in all these modules is provided both for functional data discretized in a grid and functional data expressed as a basis expansion. Across *scikit-fda*, these two representations are associated with `FDataGrid` and `FDataBasis` objects. These two types inherit from the abstract class `FData`. In general, the different public functions and objects take their input data as an `FData`. Then, internally, the implementation may be specialized for each functional data type. This enables the user to work with both representations interchangeably without adapting their code.

3

Software Development Process

The goal of this bachelor's thesis is to expand the functionality of the package *scikit-fda* [25] described in Section 2.3.1. Two brand new features are to be added: functional regression based on principal component analysis and functional regression based on partial least squares. Additionally, the current principal components dimensionality reduction functionality is to be completed.

In order to accomplish these goals, an agile software development methodology has been employed. The following sections contain the final results of the development process. Sections 3.1 and 3.2 include the final requirements and designs. Section 3.3 describes the development process, and Section 3.4 details the testing procedures.

3.1. Analysis

This section contains the analysis of the functionality to be added to *scikit-fda*. This analysis is presented in the form of a catalogue of requirements. The requirements have been divided into functional and non-functional requirements.

3.1.1. Functional Requirements

The functional requirements described in this section have been divided into the groups: basis representation (FR-BASIS), regularization (FR-REG), principal component analysis(FR-FPCA) and partial least squares (FR-FPLS).

Basis representation

As mentioned in Section 2.3.1, functional data in *scikit-fda* is stored either as the values taken by the functions in a grid or as their expansion in some basis. Currently, several generic basis are available such as Fourier o B-Splines. The selection of basis is to be expanded a new types of basis.

FR-BASIS-01 A basis object can be created from a collection of linearly independent functions that can be evaluated in a particular range $\mathcal{T}$.

1.1 The provided functions for the basis can be either discretized or given as a basis expansion.

1.2 The basis created can be evaluated at any point in $\mathcal{T}$.

1.3 It is possible to take derivatives of functional data expressed in this basis.

Regularization

In order to provide regularization in both FPC and FPLS, the calculation of penalty matrices as defined in Appendix B must be available.

FR-REG-01 The penalty matrix can be calculated for any linear differential operator and discretized functions.

FR-REG-02 The penalty matrix can be calculated for any linear differential operator and functions expressed as a basis expansion.

FPCA

The FPCA functionality can be applied in two ways. Firstly, the calculation of the FPC constitutes a dimensionality reduction technique (see Section 2.1). Secondly, this dimensionality reduction technique can be applied to regression problems as described in Section 2.1.2.

FR-FPCA-01 From a set of functional observations in discretized form, it is possible generate a basis with a specified number of principal components weights.

FR-FPCA-02 From a set of functional observations in basis form, it is possible generate a basis with a specified number of principal components weights.

FR-FPCA-03 It is possible to incorporate regularization into the calculation of the FPC basis.

3.1 The regularization corresponds to penalizing the norm of a linear differential operator applied to the calculated components.

3.2 The strength of the regularization is controlled by means of a regularization parameter.

FR-FPCA-04 The user can perform functional principal component regression (FPCR).

4.1 Regularization can be applied both to the weights calculation (FR-FPCA-03) and to the calculation of the regression coefficient.

FPLS

As with FPCA, FPLS can be applied either directly as a dimensionality reduction technique (see Section 2.2) or it can be used to solve regression problems (see Section A.4). As described in Section 2.2, FPLS works with two blocks: X and Y. When it is used as a dimensionality reduction technique, the two blocks are interchangeable. However, when it is used for regression problems, the X block corresponds to the regressor variable and, the Y block, to the response variable.

FR-FPLS-01 The user can use FPLS as a dimensionality reduction technique.

- **1.1** Both the X and Y inputs can be either multivariate or functional. The functional inputs can either be provided as discretized functions or as a basis expansion, without affecting the result.
- **1.2** It is possible to transform data back and forth between the input space and the component space.

FR-FPLS-02 The user can perform FPLSR.

- **2.1** This functionality accepts both multivariate and functional data both for the response and explanatory variables.

FR-FPLS-03 It is possible to incorporate regularization into the calculation of the FPLS weights to obtain smooth rotation functions. The regularization corresponds to penalizing the norm of a linear differential operator applied to the calculated weights.

- **3.1** Regularization can be applied to the X block.
- **3.2** Regularization can be applied to the Y block.
- **3.3** Regularization can only be applied to blocks of functional data.

3.1.2. Non-functional requirements

In this section, the different types of non-functional requirements will be detailed. Most of these requirements are dictated by the developing guidelines of the library.

Environment

NFR-ENV-01 The entirety of the library is developed in Python, with a strong focus on using *NumPy* for vectorized arithmetic to ensure performant implementations.

NFR-ENV-02 The library can be deployed in the main operating systems (MacOS, Windows and Linux-based).

Coding Guidelines

NFR-CG-01 All code must follow the Python Enhancement Proposal (PEP) 8 [27] , which provides general coding style conventions, and PEP 257 [28], which documents the semantics and conventions of docstrings. In particular, *scikit-fda*'s utilizes *wemake-python-styleguide* [29] as its linter. This is a plugin for *flake8* [30] that provides stricter checks, and is configured in the `setup.cfg` file. All additions to the codebase must be free of linter warnings.

NFR-CG-02 All code must be typed according to PEP 484 [31]. This Python Enhancement Proposal defines the semantics of type hinting.

NFR-CG-03 New designs must follow the API conventions in *scikit-learn* [26] to ensure a seamless integration between modules of both libraries. Details of this API are available in [32]. To a lesser extent, the style and naming conventions of *NumPy* [33] and *SciPy* [34] should also be taken into account.

Documentation

NFR-DOC-01 The documentation will be generated from the docstrings of the functions and classes using *sphinx* [35]. Therefore, the reStructuredText(reST) markup language must be used in the docstrings.

NFR-DOC-02 To illustrate the functionality of the classes, doctests will be employed.

NFR-DOC-03 The examples will be generated using *sphinx-gallery* [36] during the documentation build process. Each example must be included in a separate file using the *sphinx-gallery* syntax.

Testing

NFR-TST-01 The testing framework to be used is *pytest*.

NFR-TST-02 Tests for the same functionality must be grouped in a single file. Each test will be enclosed in a function whose name starts by “`test_`”.

3.1.3. Use cases

As part of the analysis process, the most relevant use cases were identified. Two use cases were considered particularly important. They are detailed in the following sections. Additionally, two functionality tests were envisioned to verify that these use cases were present in the final product. These functionality tests correspond to the first two examples of Chapter 4.

FPC Regression

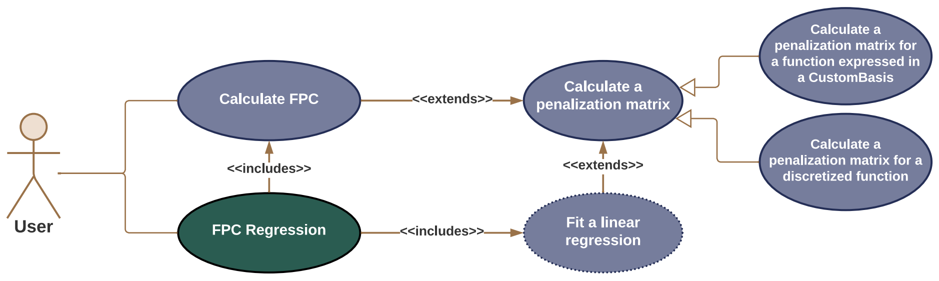

User
Calculate FPC
FPC Regression
<<includes>>
<<extends>>
Calculate a penalization matrix
<<extends>>
<<includes>>
Fit a linear regression
Calculate a penalization matrix for a function expressed in a CustomBasis
Calculate a penalization matrix for a discretized function

Figure 3.1: FPC regression use case diagram

Given a dataset with functional observations and a scalar variable, the user is able to use **FPC** regression to predict the value of the scalar variable from the functional observations. The user can choose the number of components that are used in this prediction. Additionally, the user can incorporate an adjustable level of regularization given by a linear differential operator and a regularization parameter.

FPLS Regression

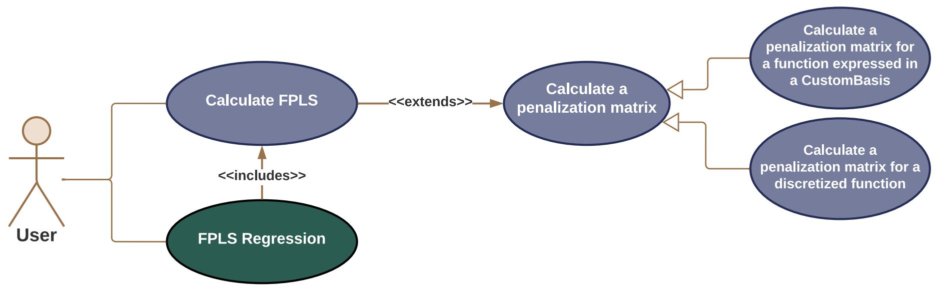

User
Calculate FPLS
FPLS Regression
<<includes>>
<<extends>>
Calculate a penalization matrix
Calculate a penalization matrix for a function expressed in a CustomBasis
Calculate a penalization matrix for a discretized function

Figure 3.2: FPLS regression use case diagram

Given a dataset with functional observations and a scalar variable, the user is able to use **FPLS** regression to predict the value of the scalar variable from the functional observations. The user can choose the number of components that are used in this prediction. Additionally, the user can incorporate an adjustable level of regularization given by a linear differential operator and a regularization parameter.

3.2. Design

An overview of *scikit-fda* was provided in Section 2.3.1. Four of its modules have to be modified to fulfill the new requirements. The internal submodules of each modified module are presented in Figure 3.3, where modified submodules have been highlighted in green.

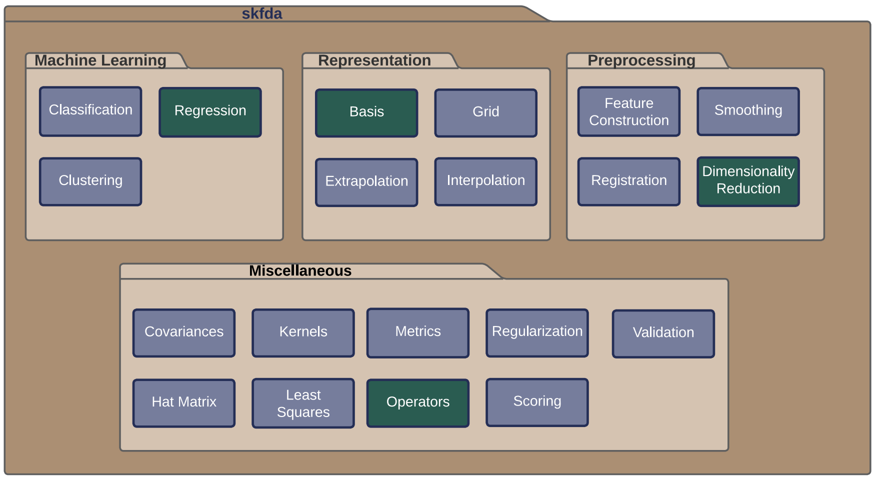


Figure 3.3: Structure of the modified `scikit-fda` modules

The changes made to each module are discussed in the following sections. The next two sections will cover the Representation and Miscellaneous modules. However, for the sake of conciseness, the design of the Machine Learning and Preprocessing modules will be described together (as these are deeply related).

3.2.1. Representation module

The Representation module contains the data structures and functionality needed to represent and store functional data within the library. Two new classes have been added to it, defining two new types of basis. As it can be seen in the diagram in Figure 3.4, both of them inherit from *`Basis`*. Note that in Figure 3.4, not all methods are included, only the most representative ones for the new *`Basis`* subtypes. A full diagram can be consulted in Appendix D (see Figure D.1). We will now describe both objects separately. Note, however, that methods that are overloading a method in *`Basis`* are not described. Their documentation can be checked at `https://fda.readthedocs.io/en/latest/modules/autosummary/skfda.representation.basis.Basis.html`.

- `CustomBasis` is a basis where the basis functions can be arbitrary `FData` objects as long as they are linearly independent. This basis definition seeks to fulfill FR-BASIS-01. The brand new methods are:
  - The method `_check_linearly_independent` checks that the input functions are li-

  nearly independent. This method is called during the construction of the object. If the provided functions are not linearly independent, a `ValueError` exception is raised. This method has specializations both for `FDataGrid` and `FDataBasis` arguments.

  - The method `_create_subspace_basis_coef` creates a new basis spanning subspace generated by the derivatives of the basis functions. The second argument contains the coefficients of some functions in the original basis. The function returns the coefficients of the derivatives in the newly created basis. This method has specializations both for `FDataGrid` and `FDataBasis` arguments.

- `GridBasis` is a basis that enables the representation of an `FDataGrid` object as an `FDataBasis`. . An `FDataBasis` with a `GridBasis` as its basis and a series of coefficients in this basis corresponds to an `FDataGrid` discretized in the `GridBasis grid_points` attribute and with the `FDataBasis` coefficients as the `data_matrix`.

  This different representation for `FDataGrid` object has enabled simplifications in the penalty calculation (see Section 3.2.2). However, this basis does not support evaluation. Therefore the `_evaluate` method raises an exception. To evaluate an `FDataBasis` with this underlying basis, it must be converted to an `FDataGrid` first.

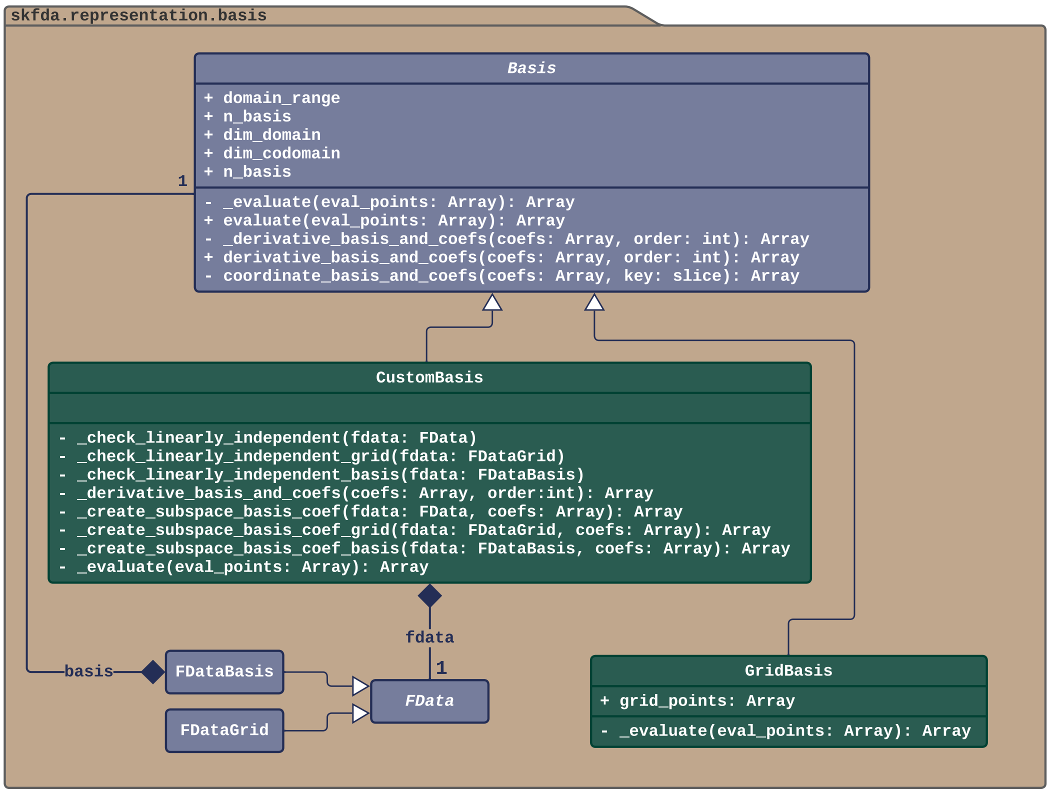
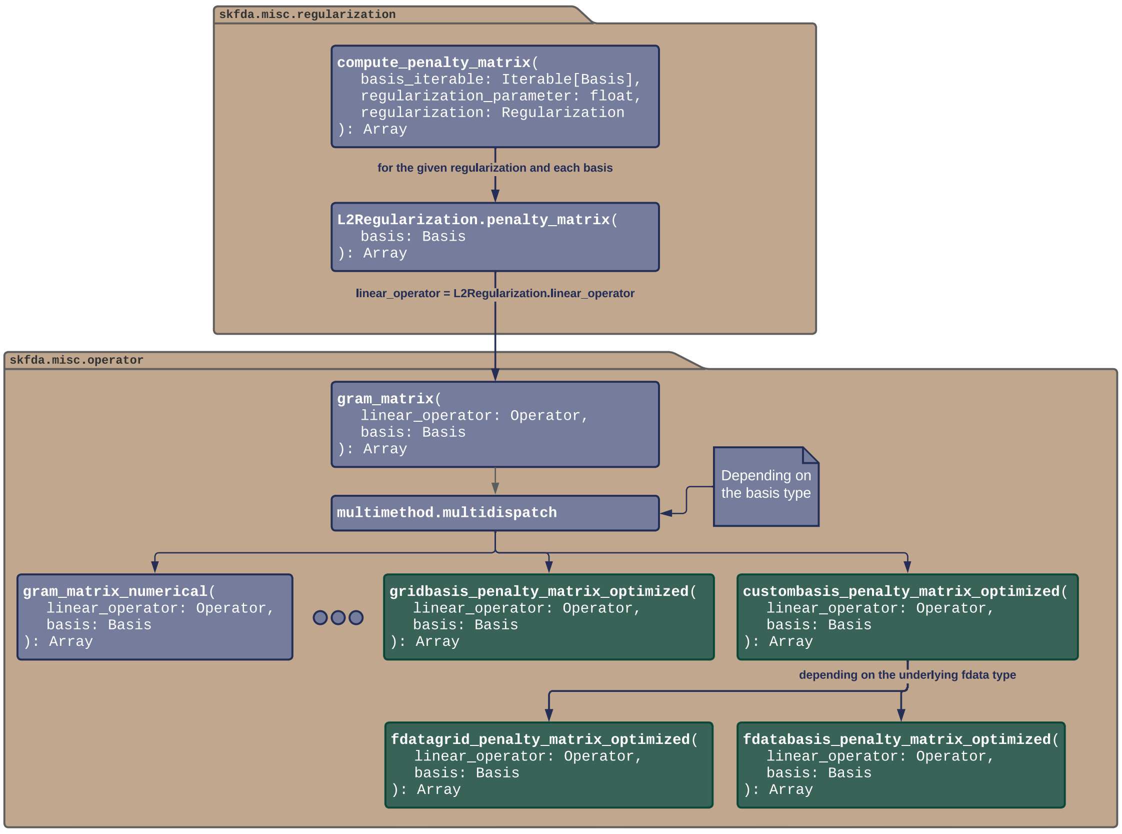

skfda.representation.basis
Basis
+ domain_range
+ n_basis
+ dim_domain
+ dim_codomain
+ n_basis
- _evaluate(eval_points: Array): Array
+ evaluate(eval_points: Array): Array
- _derivative_basis_and_coefs(coefs: Array, order: int): Array
+ derivative_basis_and_coefs(coefs: Array, order: int): Array
- coordinate_basis_and_coefs(coefs: Array, key: slice): Array
1
CustomBasis
- _check_linearly_independent(fdata: FData)
- _check_linearly_independent_grid(fdata: FDataGrid)
- _check_linearly_independent_basis(fdata: FDataBasis)
- _derivative_basis_and_coefs(coefs: Array, order:int): Array
- _create_subspace_basis_coef(fdata: FData, coefs: Array)): Array
- _create_subspace_basis_coef_grid(fdata: FDataGrid, coefs: Array)): Array
- _create_subspace_basis_coef_basis(fdata: FDataBasis, coefs: Array)): Array
- _evaluate(eval_points: Array): Array
fdata
1
basis
FDataBasis
FDataGrid
FData
GridBasis
+ grid_points: Array
- _evaluate(eval_points: Array): Array

Figure 3.4: Design for the changes to the Representation module

3.2.2. Miscellaneous module

The changes needed in this module were minor. The only goal was to extend the already present penalty matrix calculation functionality to the new basis types. This part of the library relies heavily on global functions instead of object instances. Therefore, to describe its design, a call graph has been used instead of a class diagram. This can be seen in Figure 3.5, with the new functions highlighted in green.


skfda.misc.regularization
for the given regularization and each basis
linear_operator = L2Regularization.linear_operator
skfda.misc.operator
Depending on the basis type
depending on the underlying fdata type

```
compute_penalty_matrix(
   basis_iterable: Iterable[Basis],
   regularization_parameter: float,
   regularization: Regularization
): Array
```

```
L2Regularization.penalty_matrix(
   basis: Basis
): Array
```

```
gram_matrix(
   linear_operator: Operator,
   basis: Basis
): Array
```

```
multimethod.multidispatch
```

```
gram_matrix_numerical(
   linear_operator: Operator,
   basis: Basis
): Array
```

```
gridbasis_penalty_matrix_optimized(
   linear_operator: Operator,
   basis: Basis
): Array
```

```
custombasis_penalty_matrix_optimized(
   linear_operator: Operator,
   basis: Basis
): Array
```

```
fdatagrid_penalty_matrix_optimized(
   linear_operator: Operator,
   basis: Basis
): Array
```

```
fdatabasis_penalty_matrix_optimized(
   linear_operator: Operator,
   basis: Basis
): Array
```

Figure 3.5: Call graph of the penalty matrix calculation

The central function in the penalty matrix calculation is `gram_matrix`. This function, in turn, uses dynamic dispatch to call a specialized function depending on the type of basis given. If there are no optimized versions available for that particular basis, `gram_matrix_numerical` provides a numerical fallback. We now describe the rest of the functions:

- `gridbasis_penalty_matrix_optimized`. This function uses the procedures described in Appendix B to calculate the penalty matrix of the given operator for discretized functions. The discretization grid is defined by the `grid_points` in the given `GridBasis`.
- `custombasis_penalty_matrix_optimized`. This function calculates the penalization matrix in a `CustomBasis` basis. Depending on whether the basis functions are discretized on a grid

or expressed as a linear combination of an underlying basis, the calculation is delegated to one of the following methods:

- `fdatagrid_penalty_matrix_optimized`. This function is called when the basis functions are discretized. It calculates the penalization matrix by applying the differential operator to each basis function and estimating the L^2 products with Simpson's integration weights [16, p.257].
- `fdatabasis_penalty_matrix_optimized`. This function is called when the basis functions are provided as an expansion on an underlying basis. In this case, the penalty matrix is calculated using the penalty matrix of the underlying basis.

3.2.3. Machine Learning and Preprocessing modules

Most of the functionality developed in this project is integrated in these modules. An overview of the design can be seen in Figure 3.6. The details of the classes contained in both these submodules can be found in Figures D.2, D.3 and D.4 in the Appendix D. Note, however, that the `LinearRegression` and `L2Regularization` classes were not part of this work and they are only included in the diagram since it the new classes contain instances of them. The overview in Figure 3.6 shows how the Regression and Dimensionality Reduction submodules are interrelated. Both `FPCARegression` and `FPLSRegression` depend on their counterparts in the Dimensionality Reduction submodule.

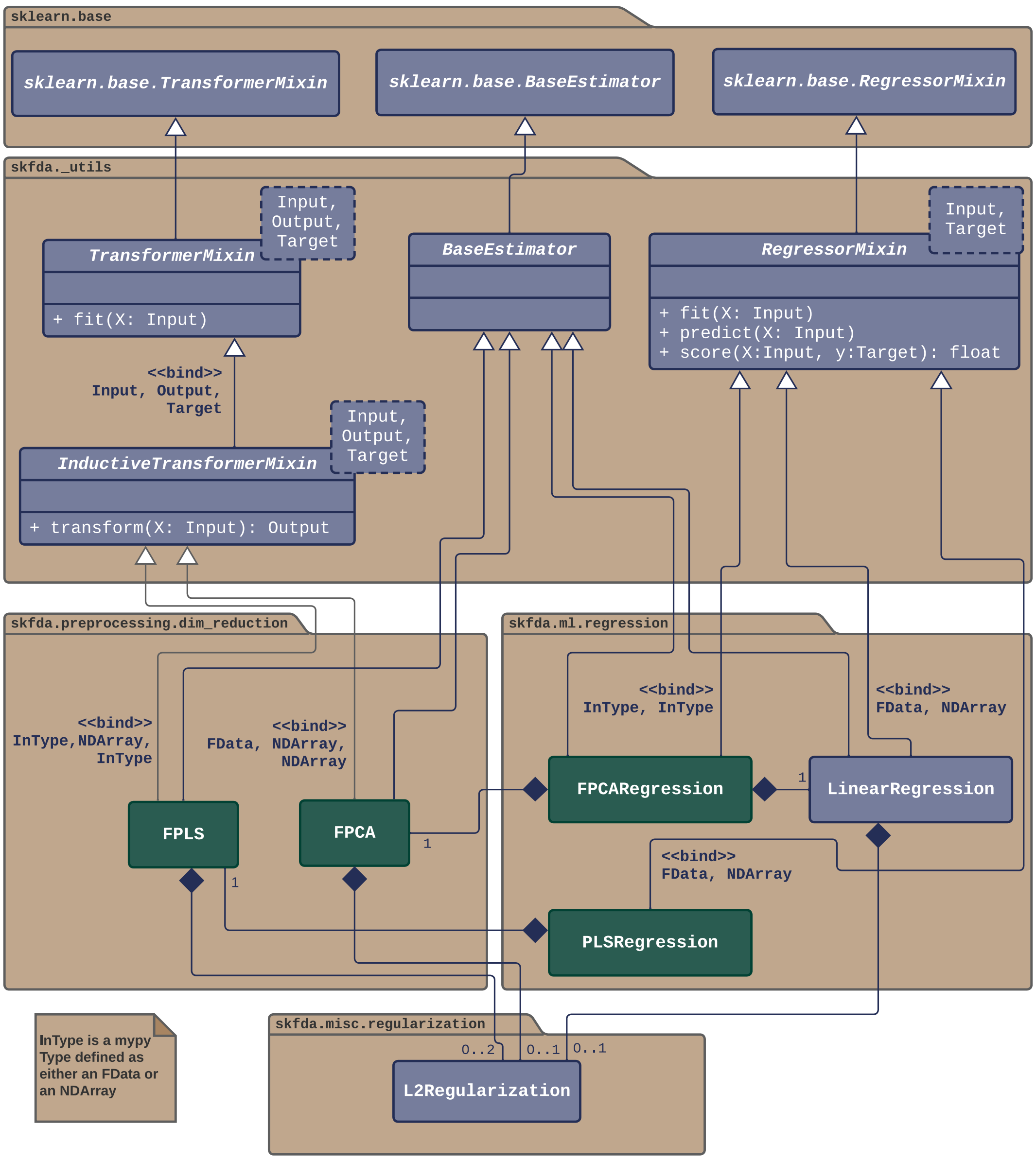
As depicted in Figure 3.6, all the estimators have the base classes of *scikit-learn* as an ancestor. Moreover, their interfaces follow the *scikit-learn* API guidelines [32]. Four methods are of particular importance: `fit`, `predict`, `transform` and `inverse_transform`. All estimators have a `fit` method to process the training data. The regressors (`FPCARegression` and `FPLSRegression`) predict the response to new data with the `predict` method, while the transformers (`FPCA` and `FPLS`) project the data onto the component space using the `transform` method, with `inverse_transform` performing the opposite operation.

The `FPCA` class calculates the Functional Principal Components as described in Section 2.1. In order to provide regularization capabilities, it can optionally contain a `L2Regularization` object. This class is defined in the `skfda.misc.regularization` module and represents the regularization for a given linear differential operator.

Following the procedure described in Section 2.1.2, `FPCARegression` relies on both `FPCA` and `LinearRegression` to carry out the functional principal component regression. Both the fit and predict methods delegate in these two components. This class can contain up to two regularization objects, one of them describes the regularization for the FPC calculation, while the other describes the regularization for the linear regression.

The design of the PLS funcionality follows a similar approach. Firstly, the `FPLS` class implements

PLS as a dimensionality reduction technique as described in Section 2.2. This class has a similar interface as `FPCA`. However, the inputs to both `fit` and `transform` are doubled as PLS works with two blocks of variables (X and Y), instead of one. For the same reasons, two regularization objects can be contained in a single `FPLS` transformer.



sklearn.base
sklearn.base.TransformerMixin
sklearn.base.BaseEstimator
sklearn.base.RegressorMixin
skfda._utils
TransformerMixin
Input, Output, Target
+ fit(X: Input)
BaseEstimator
RegressorMixin
Input, Target
+ fit(X: Input)
+ predict(X: Input)
+ score(X:Input, y:Target): float
<<bind>> Input, Output, Target
InductiveTransformerMixin
Input, Output, Target
+ transform(X: Input): Output
skfda.preprocessing.dim_reduction
<<bind>> InType,NDArray, InType
<<bind>> FData, NDArray, NDArray
FPLS
FPCA
skfda.ml.regression
<<bind>> InType, InType
<<bind>> FData, NDArray
FPCARegression
LinearRegression
<<bind>> FData, NDArray
PLSRegression
1
skfda.misc.regularization
0..2
0..1
0..1
L2Regularization
InType is a mypy Type defined as either an FData or an NDArray

Figure 3.6: Design overview of the Machine Learning and Preprocessing modules

Secondly, `FPLSRegression` utilizes `FPLS` to carry out functional PLS regression as described in Section A.4. In order to be able to reuse `FPLS` for regression, the transformer must implement the two deflation modes of PLS (canonical and regression). A private parameter controls which mode is active in `FPLS`, selecting the canonical mode by default. When this object is initialized during the fitting of a `FPLSRegression` object, the regression mode is selected.

3.3. Development methodology

This section seeks to describe the steps, procedures and methodologies used in this project. Firstly, the general project structure is introduced. Then, the tools and methods used to ensure code quality during the development process are enumerated. Finally, the documentation methodology for the new code is described.

3.3.1. Project organization

One of the first decisions in a software project is which kind of methodology should be used. This was considered at the very beginning of this bachelor's thesis. The decision was made to use an agile methodology. The goal of the project is to implement computational tools that enable functional data processing in Python. However, especially when it comes to PLS, there are several ways to approach the problem, and their advantages and drawbacks are not readily apparent at first sight. In this situation, an agile methodology enables us to iterate on the different possible approaches and find the best solution. The development cycle is portrayed in Figure 3.7.

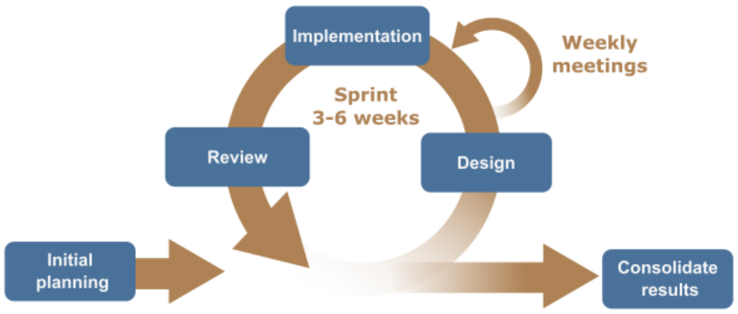


Figure 3.7: Development cycle

At the beginning, an initial planning phase took place. During this phase, the broad goals of the project were defined. It was decided that both FPCA and FPLS regression had to be added to the *scikit-fda* library.

With those goals in mind, the development process started. A series of approximately weekly sprints took place. In each sprint, new techniques were implemented and different functionality was tested. At the end of the sprint, during the review phase, the results obtained were evaluated. If the results were considered sufficient, the focus of the next sprint was set to a new part of the new functionality. Otherwise, the next sprint was dedicated to overcoming the encountered obstacle and reaching a better solution. The beginning of a new sprint was also used to look for new references, read different articles and attain a better understanding of the problem to solve.

During the sprints, weekly progress meetings were held. In these meetings, the members of the *scikit-fda* development team shared their progress, blocking points and foresight on how the development should move forward. This approach has two main benefits. Firstly, the entire development team is aware of how other features are being implemented. This can prevent integration problems when the functionality is merged and improve the consistency across modules. Secondly, blocking points are identified quickly and everyone can help to push the development forward.

When all the desired functionality was achieved, the sprint phase was concluded. From this point onwards, the effort was spent consolidating the results. This included preparing the examples for the already finished functionality and revising the code, tests and documentation to identify possible issues and correct them. A Gantt diagram of the entire project is included in Appendix C.

Aside from the development cycle, it is also critical to define how the development team of *scikit-fda* should collaborate. Creating a library is a complex task, but one that can be broken down into parts with little dependencies. As a result, there have been several people working on different features for `scikit-fda` at the same time. In order for this collaboration to be successful, it is essential to use some kind of version control system. Even though it is not the only option, `git` is the de-facto standard for many open-source projects. In particular, a repository in GitHub is used as the remote. Aside from storing the source code, GitHub provides the issue tracking and pull request functionality that is at heart of effective software development collaboration.

The most basic collaborative tools in `git` are branches. In the case of `scikit-fda`, these branches are organized following the gitflow model. An example of the branch structure in this workflow is included in Figure 3.8 [1] . Five kinds of branches are defined in this model.

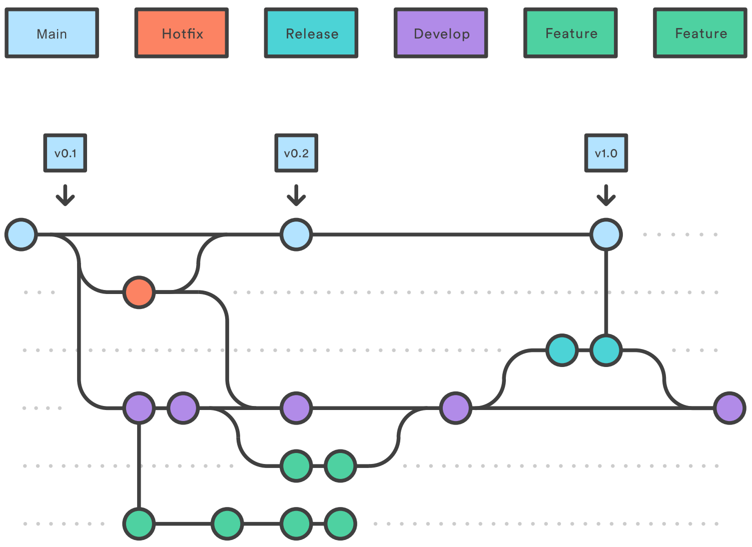


Figure 3.8: Example of gitflow branching structure

The main branch only contains the official releases of the library, while develop is the branch to

[1] Image licensed under Creative Commons Attribution 2.5 from `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow` (fetched on the 6th of May 2023)

which new features are integrated. Once the develop branch is deemed to be stable enough and all features have been integrated, it is merged to the master branch. Usually, the merge commits to the master are tagged since they correspond to different versions of the library.

Most of the work is performed in feature branches. Each one of these branches is created to implement a particular functionality. Once the functionality has been coded and tested, they are merged into the develop branch. Finally, hotfix branches are created to quickly patch production releases. This is very beneficial when it comes to bug fixing. It allows us to quickly push out a change both to the develop and master branch without interrupting the current development efforts in other branches.

Before merging a feature branch into develop, a code review is carried out. That is to say, another developer reviews all changes in the pull request. This has two main benefits. Firstly, it greatly reduces the number of bugs introduced into development. Secondly, it helps identify code sections that are not easy to understand but might seem trivial to the developer who implemented them. Upon identification, these sections can be refactored or documented properly so that they are as straightforward as possible.

3.3.2. Code quality assurance

One of the key development principles in *scikit-fda* is the focus on maintainability, code conventions and complete documentation. A coherent style must be followed across the entire codebase to make all modules approachable to new contributors. Coding guidelines (see requirements NFR-CG-01, NFR-CG-02 and NFR-CG-03) have been established to accomplish this goal. In particular, three standards are employed. On the one hand, PEP 8 regulates indentation, white lines, trailing commas, line width and other factors that could hinder the readability of the code. On the other hand PEP 257 provides a standard format for docstrings. That is to say, the build-in documentation strings in Python. Finally, PEP 484 defines the semantics of the type hits in Python. In order to ensure the quality of the resulting code several tools and processes have been used. They have been separated into four groups: linting, formatting, type checking, continuous integration and testing and profiling

Linting

In order to ensure that the written code meets the PEP 8 guidelines, it is imperative to have automatic checkers that point out violations. That role is primarily fulfilled by the following linting tools:

- *flake8* [30] is a linter that scans Python code and indicates the lines whose style deviates from the PEP 8 standard.
- *wemake-python-styleguide* [29] is a plugin for *flake8* that incorporates stricter rules. It ensures consistency across the code and helps identify complex sections that should be refactored to ensure readability.

Formatting

In order to automate some of the process of producing PEP 8 compliant code, automatic code formatters have been used. These tools help ensure consistency and readability across the library:

- *isort* [37] is a tool that reorganizes the Python imports at the beginning of a file. This reduces merge conflicts and provides a more consistent ordering of the import statements.
- *black* [38] is a heavily opinionated Python formatter. *black* is PEP8 compliant but it is not fully compatible with *wemake-python-styleguide*. It was very effective at fixing most of the linter warnings quickly, even if some manual fixes were needed.
- *autopep8* [39] is another Python formatter. This one has the advantage of being able to process file fragments. Therefore it could be used in files where only a portion of them was modified.

Type checking

Even though Python ascribes to the duck typing paradigm, a type checking tool can identify potential bugs before the code is even tested. By detecting situation where the expected types do not match the types of the objects being used, buggy code can be detected and refactored. To do so, *mypy* is used:

- *mypy* [40] is a static type checker for Python. Using the type hints defined in PEP 484, mypy can check if the type definitions are consistent with the function calls and class instantiations.

Continuous integration and testing

Another step taken to increase code quality is the use of GitHub Actions. GitHub Actions is a CI (Continuous integration) platform that provides automatic testing and linting pipelines. These pipelines are then triggered by any new commit added to a pull requests, which can only be merged if the pipelines are successfully executed. For example, there are pipelines that run all tests in all target operating systems (see NFR-ENV-02) and only finish successfully if all tests pass. Additionally, other pipelines are used to enforce PEP 8 and *mypy* compliance.

Regarding the tests themselves, up until recently, *unittest* was the framework used in *scikit-fda*. However, currently, new tests are being created following *pytest*'s format, as it constitutes a simpler and more effective framework.

- *pytest* [41] is a Python testing framework that simplifies the definition of test files. Its definition of parametrized testing functions avoids unnecessary repetition while writing similar test scenarios.

Profiling

In order to identify bottlenecks, sections that were found to be computationally intensive were analyzed using profiling tools. These tools are capable of summarizing where the most CPU time was spent, identifying the hot path and possible areas of improvement. Depending on the granularity required, different tools were employed:

- *cProfile* is a deterministic profiler for Python. It is part of the Python Standard Library and is capable of tracing function calls and calculate the percentage of time spent in different functions.
- *gprof2dot* [42] is a Python utility that generates function call graphs from the output generated by many different profilers. In particular, it supports the creation of function call diagrams from the output of *cProfile*. Figure 3.9 contains a section of a call graph generated during the optimization process of the penalty matrix calculation.
- *line_profiler* [43] is a line by line Python profiler. This profiler can calculate the CPU time spent in each line of some previously selected functions. It was used to provide finer measurements than *cProfile* when needed.

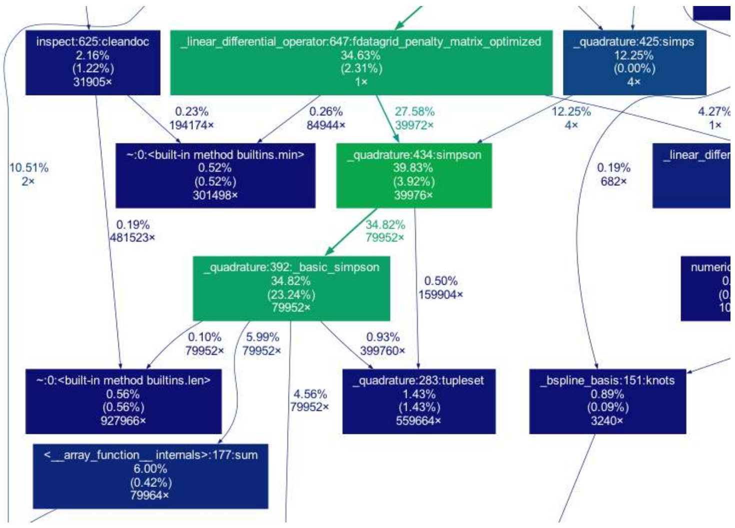


Figure 3.9: Function call graph generated as part of the optimization process for the discretized penalty matrix calculation

3.3.3. Documentation

The main tool used to document Python codes are docstrings. A docstring is defined as a string literal inserted in the first line of the definition of a module function, class or method. This string contains a description of the defined element as well as some additional details that could help understand its interactions with other elements. In some cases such as class methods, all needed information can be summarized in a single line.

On the contrary, docstrings for class definitions tend to be longer in order to provide a detailed explanation of the class. An example of this is provided in Code 3.1. These docstrings usually also include a list of the arguments required to build an instance of this class and the attributes of the class. Additionally, in most cases an examples section is added. These examples are known as doctests since they are executed when the documentation is built, and their results are compared to the ones indicated in the doctest itself. Therefore, they fulfill two purposes. Firstly, they provide extra validation of some basic functionalities. Secondly, they provide an example of how the class should be used.

```python
class FPCARegression(
    BaseEstimator,
    RegressorMixin,
):
    r"""Regression using Functional Principal Components Analysis.

    It performs Functional Principal Components Analysis to reduce the
    dimension of the functional data, and then uses a linear regression model
    to relate the transformed data to a scalar value.

    Args:
        n_components: Number of principal components to keep. Defaults to 5.
        fit\_intercept: If True, the linear model is calculated with an
            intercept.  Defaults to ``True``.
        pca_regularization: Regularization parameter for the principal
            component extraction. If None then no regularization is applied.
            Defaults to ``None``.
        regression_regularization: Regularization parameter for the linear
            regression. If None then no regularization is applied.
            Defaults to ``None``.
        components_basis: Basis used for the principal components. If None
            then the basis of the input data is used. Defaults to None.
            It is only used if the input data is a FDataBasis object.

    Attributes:
        n\_components\_: Number of principal components used.
        components\_: Principal components.
        coef\_: Coefficients of the linear regression model.
        explained\_variance\_: Amount of variance explained by
            each of the selected components.
        explained\_variance\_ratio\_: Percentage of variance
            explained by each of the selected components.

    Examples:
        Using the Berkeley Growth Study dataset, we can fit the model.

        >>> import skfda
        >>> dataset = skfda.datasets.fetch_growth()
        >>> fd = dataset["data"]
        >>> y = dataset["target"]
        >>> reg = skfda.ml.regression.FPCARegression(n_components=2)
        >>> reg.fit(fd, y)
        FPCARegression(n_components=2)

        Then, we can predict the target values and calculate the
        score.

        >>> score = reg.score(fd, y)
        >>> reg.predict(fd) # doctest:+ELLIPSIS
        array([...])

    """
```

Code 3.1: `FPCARegression` docstring

However, these doctests are not limited to class definitions. In fact, they have proven to be very useful when documenting complex functions. Adding a doctest to their docstring provides simple example of the inputs and expected outputs. Moreover, since some functions are not part of a class, including a simple test in the docstring is more convenient that creating a dedicated unittest file if only very basic validation is needed.

Nevertheless, these short examples are not sufficient to fully showcase the functionality of each class or module. To do that, a series of examples are cleated along with each new major feature added to the library. These have a similar structure to a notebook and are rendered into html by *sphinx*. All the examples of the library are accessible at `https://fda.readthedocs.io/en/latest/auto_examples/index.html`.

To conclude this section, it is important to mention that the online documentation is generated automatically from the docstrings in the source code by *sphinx*. This simplifies the documentation process immensely. If the source code has been documented appropriately, it suffices to add the source file to the list of files processed by *sphinx* to have the new functionality added to the online documentation.

3.4. Testing

To ensure that the developed code is correct and free of bugs, several different kinds of tests have been applied. In this section, the use of doctests, unit tests and functionality tests is described, and their integration with the continuous integration system is explained. The type checking validation process using *mypy* is not included in this section, as it has already been covered in sufficient detail in Section 3.3.2.

3.4.1. Doctests

Doctests are small code fragments included within the documentation of a class or a function. These fragments are executed during the build process for the documentation and the results are compared against the expected values specified in the doctest itself. The validation power of these tests is rather limited, but executing them during the documentation build ensures that the documentation stays up to date with the implementation

3.4.2. Unit tests

The different classes in *scikit-fda* have associated unit tests that verify their correctness. Each class to be tested has an associated test class with all its tests. In order to validate the requirements, three different testing approaches have been followed. First of all, some small tests are included with data for which the output is already known. This includes both tests of very simple scenarios that can be solved by hand and tests of the behaviour of the class. For example, tests that verify that exceptions are raised for certain inputs.

Secondly, in some cases, the same functionality has been implemented in some R packages such as *fda* or *fda.usc*. In those cases, tests are created to compare our results with the ones obtained in R with the same inputs. However, in some cases, the results from one of the libraries has to be adapted to be comparable. For example, the notion of a penalization matrix in *fda.usc* is slightly different and its values differ from the one obtained in our code by a certain scalar that depends on the discretization grid.

Thirdly, some problems can be solved in more than one way. For instance, there are two ways to calculate the norm of a differential operator applied to a function. On the one hand, using the regularization matrix framework, a penalization matrix can be used to calculate this norm. On the other hand, one can calculate the function that results from applying the linear operator to the original function and, then, calculate its norm. Since both methods are equivalent, their results can be compared to validate the new implementation.

Additionally, in some cases, there are other options available. For example the discretized version of FPCA with unit weights and no regularization must be equivalent to the multivariate PCA. These kind of tests have also been included since they are very useful to ensure consistency with the estimators in *scikit-learn*. This is particularly important since the libraries should have a seamless integration (see NFR-CG-03).

3.4.3. Functionality tests

The functionality tests are comprised of a series of examples of use. These intend to mimic the behaviour of an end user and ensure that the desired functionality is present. In contrast with the unit tests, no extreme scenarios or edge cases are considered in these test. As with the doctests, these tests are executed when the documentation is built by *sphinx_gallery*, and they are a great addition to the documentation of the library.

3.4.4. Continuous integration and testing

A combination of all these testing approaches is used to provide adequate coverage and requirement validation. The coverage of the entire codebase is tracked using *codecov*, which is integrated into GitHub Actions. Therefore, only pull requests whose tests are successful and have a sufficient coverage are accepted onto the develop branch. In particular, the average coverage of the files added for the new functionality is around 91 %.

4

Results

The goal of this chapter is to showcase the application of the regression techniques developed during this project. Three different experiments have been selected. The first one shows the application of FPCR with regularization to a noisy dataset. The second one seeks to display the performance of FPLSR when solving the same problem. Finally, the third scenario compares the two methods applied to a less noisy dataset, in which no regularization is needed.

4.1. FPCR applied to the Canadian Weather dataset

The Canadian Weather dataset from [14] contains the daily average temperature of 35 Canadian weather stations along with their geographical information. In this example, the goal is to predict the latitude of the weather station given the evolution of its daily average temperatures.

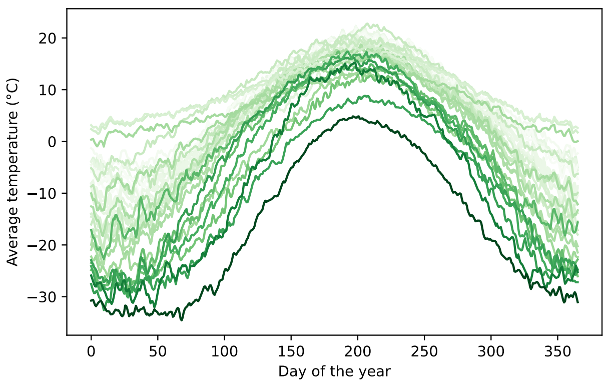
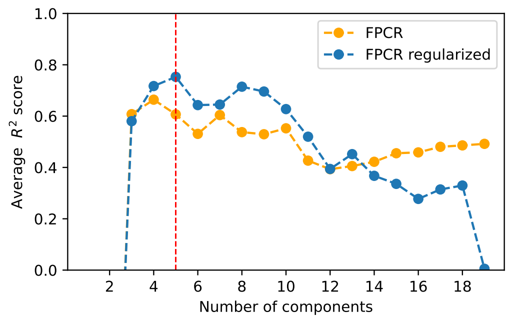


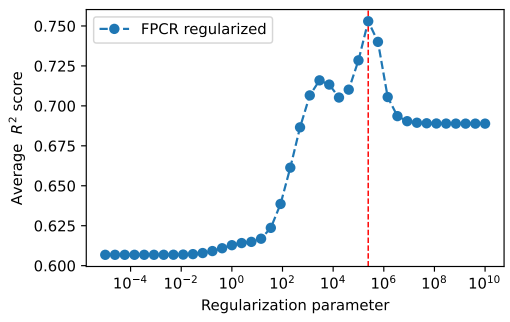
Figure 4.1: Canadian Weather dataset. Each curve represents the average daily temperature. Their color corresponds to latitude of the corresponding stations. The darker the color, the higher the latitude.

Figure 4.1 portrays the daily temperatures for each weather station. The daily measurements have been averaged across every year from 1960 to 1994. In Figure 4.1, the curves corresponding to northern stations are colored in a darker shade of green, while the curves of southern stations are lighter. Naturally, the northern stations tend to be colder.

Given the noisy nature of the data, it was decided to apply regularization to the calculation of the principal components. In order to do so, a regularization parameter must be selected. With that goal in mind, a grid search with cross validation was used. Since the `FPCARegression` predictor has been design to be compatible with *scikit-learn*'s interface, the `GridSearchCV` function in said library can be used. The best regularization parameter might change depending on the number of components. Therefore, the grid search considered both the regularization parameter and the number of components. The best 5-fold cross validation scores were obtained with 5 components and a regularization parameter of 242446.



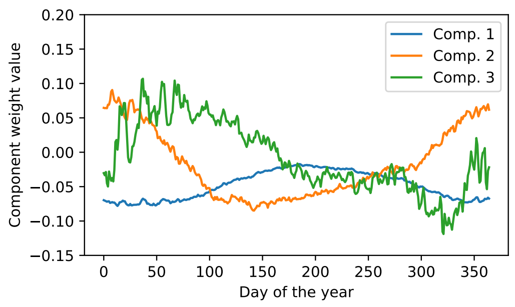
(a) Depending on the number of components



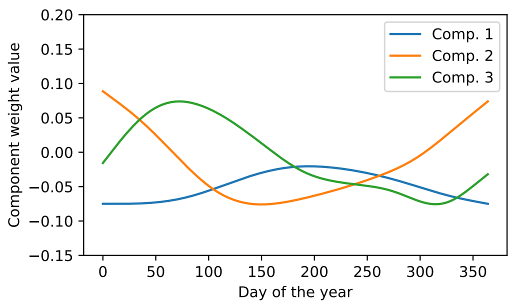
(b) Depending on the regularization parameter

Figure 4.2: Average R^2 score obtained in 5-fold cross validation varying either the number of components or the regularization parameter, with the other one kept at the optimum

Figure 4.2 shows the cross validation scores obtained depending on the number of components and regularization parameter with and without regularization. Without regularization, the highest cross validation score is 0.664 ± 0.17. With regularization, the score raises to 0.752 ± 0.11.



(a) Without regularization



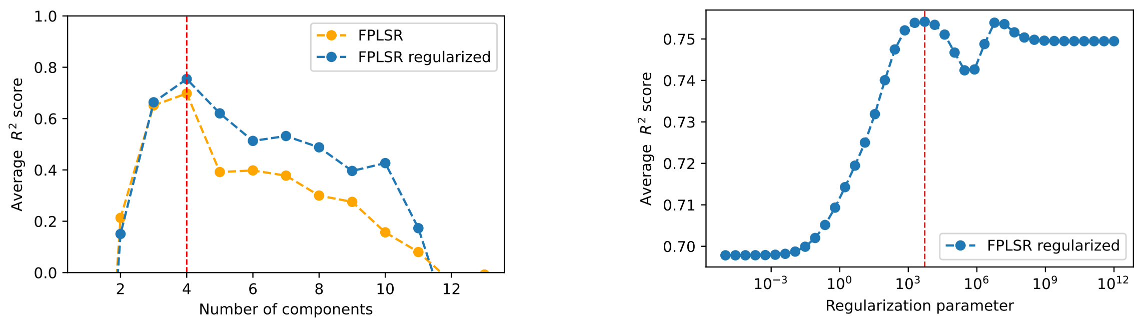
(b) With regularization

Figure 4.3: Principal components weights for the first three components

Finally, in Figure 4.3, the weights for the first three principal component are shown. It can be easily seen that the regularization was able to smooth the weights, removing the noise from the principal components and leading to better results in the regression.

4.2. FPLSR applied to the Canadian Weather dataset

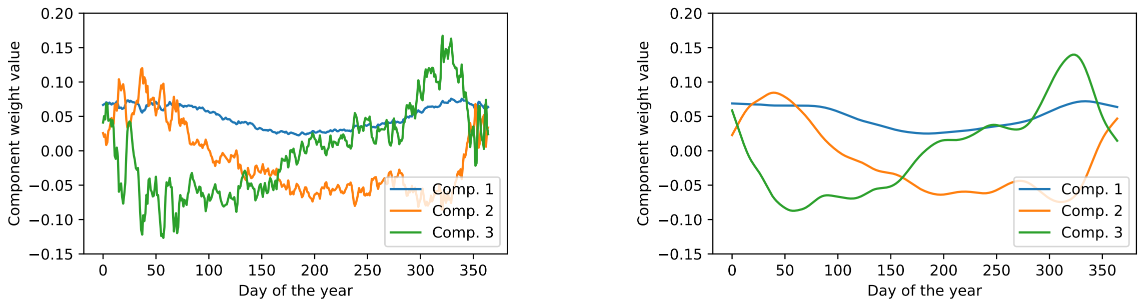
The regression problem that FLPSR was tasked with solving in this scenario is the same as the one in the previous section. Once again, due to the noise in the regressor (X) variable, regularization has to be applied to the weights in the X block. As with FPLSR, both the number of components and the regularization parameter were decided based on the results of a cross validation grid search. The optimum parameters were found to be 4 components and a regularization parameter of 5223.



(a) Depending on the numer of components (b) Depending on the regularization parameter

Figure 4.4: Average R^2 score obtained in 5-fold cross validation varying either the number of components or the regularization parameter, with the other one kept at the optimum

In Figure 4.4, the performance of the model with different number of components and regularization parameters is shown. As it can be seen in Figure 4.4(a), applying regularization leads to better results. In particular, with 4 components, the cross validation score jumps from 0.698 ± 0.14 to 0.754 ± 0.12.



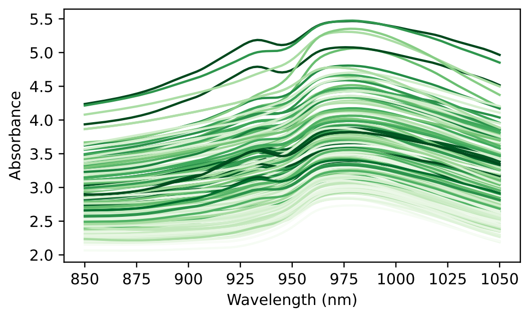
(a) Without regularization (b) With regularization

Figure 4.5: X block FPLS weights for the first three components

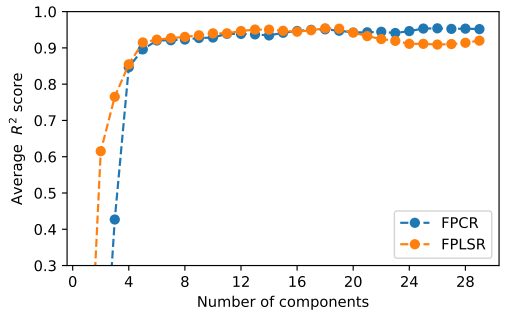
Finally, in Figure 4.5, the first three X block FPLS weights are plotted. It can be easily seen that the regularization was able to remove the noise in the weights, which in turn led to better results. It can also be noted that the weights are different to the ones extracted by FPCR (Figure 4.3) even though both methods obtained very similar cross validation scores.

4.3. Comparison between FPLSR and FPCR in the Tecator dataset

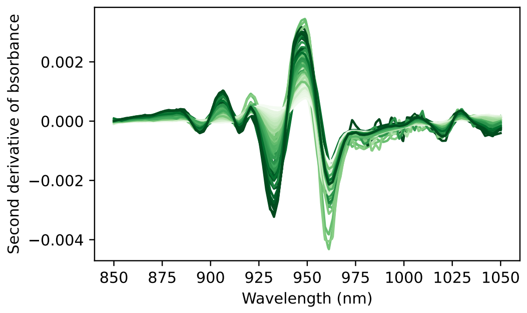
This last scenario seeks to directly compare FPLSR and FPCR using one of the most popular datasets for functional regression with scalar response: the Tecator dataset (from `http://lib.stat.cmu.edu/datasets/tecator`). This dataset contains the absorbance spectra for 215 meat samples, as well as their concentration of fat, protein and water. Both regression methods are tasked with predicting the fat contents from the absorbance curves. The input data is shown in Figure 4.6(a), where darker shades of green correspond to samples with higher contents of fat.



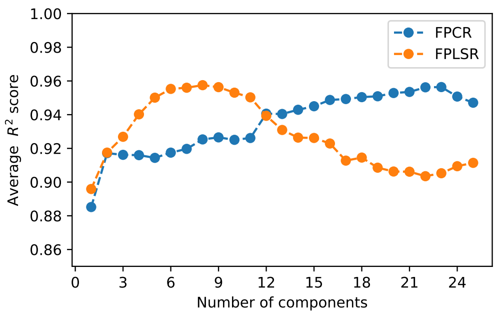
(a) Absorbance curves



(b) Average CV scores with the absorbance curves



(c) Second derivatives



(d) Average CV scores with the second derivatives

Figure 4.6: FPLSR and FPCR applied to the absorbance curves

Figure 4.6(b) shows the evolution of the average R^2 score obtained with 5-fold cross validation depending on the number of components considered. There is very little difference between the maximum score obtained by both methods, with FPLSR obtaining slightly better results.

Frequently, when analyzing the Tecator dataset, the estimators are fitted using the second derivatives of the absorbance curves instead, which provide a better separation between high and low fat content (see Figure 4.6(c)). Fitting the estimators using the second derivatives leads to slightly better results. However, the main difference is the number of components required to obtain the best results. In particular, FPLSR obtain its best result with only 8 components. This shows the better capabilities of FPLSR when it comes to summarizing the information needed for the regression in the first components.

5 Conclusions and future work

The main objective of this bachelor's thesis was to implement regression methods for functional data based on dimensionality reduction. Specifically, functional partial least squares regression (FPLSR) and functional principal components regression (FPCR). Both techniques have been implemented and integrated in *scikit-fda*, a Python package for statistical analysis and machine learning with functional data. The goal of these dimensionality reduction methods is to identify a low-dimensional basis of components that are linear combinations of the original functional observations. In FPCR, these components are computed by maximizing the explained variance of the functional covariates. In FPLSR, the components maximize the covariance between these covariates and the predicted variable. Once this basis is obtained, the functional observations are mapped to their coefficients in this basis. In this manner, the original functions, which are, in principle, infinite-dimensional objects, are represented in a finite-dimensional space in which standard multivariate regression can be applied.

The basis derived by FPCA and FPLS are expected to capture relevant information in the data better than generic basis such as Fourier or B-Splines since they are computed from the data themselves. In particular, FPLS is expected to yield a representation that is better adapted to the regression problem because it takes into account not only the regressor variable, but also the response.

Moreover, to identify basis that are smooth, roughness penalties have also been implemented for both FPCA and FPLS. These penalties are defined in terms of linear differential operators. For instance, the L^2 norm of the second derivative can be used to penalize the curvature of the components. Besides being smooth, these components are usually more interpretable, and the regression models built on them are more robust.

The effectiveness of these methods has been illustrated by means of two examples, where they were employed to analyze meteorological and spectroscopic measurements respectively with the goal of predicting related quantities. The first one shows the benefits of applying regularization when the input data is noisy. In the second one, both methods are compared in a situation where no regularization is needed, but in which fitting the models with the derivatives of the original data leads to more accurate predictions. The better performance of FPLSR and FPCR in these examples illustrates the usefulness of these methods. In future developments, these methods will be extended to accept multiple functional covariates.

Another possible extension is the implementation of canonical correlation analysis (CCA). This dimensionality reduction technique is similar to PLS. However, where PLS maximizes the variance between the components, CCA maximizes the correlation. It is even possible that part of the PLS current design could be reused to implement CCA, as the two methods are very similar (see [19, p. 26]). However, the application of this dimensionality reduction technique to regression problems has not been explored as much as PLS in the FDA literature. As we briefly mentioned in Chapter 2, there are different alternative methods of PLS. Therefore, even within the PLS family, methods such as SIMPLS [18] and PLS-SVD [19] could be candidates for a future expansion of *scikit-fda*'s functionality.

However, the NIPALS version of FPLSR and the FPCR methods were considered to be the most applicable to many problems. Both of them are general tools that can be applied to a wide range of functional regression problems. In order to be able to implement them within the library, it was first required to understand both the functional data analysis literature in general, and dimensionality reduction techniques in particular. Once these topics were understood, a set of requirements was derived. From there, a design that fulfilled them was created and, when it was considered to be stable enough, its implementation was started. Finally, the functionality was tested using both unit tests and examples that validate each use case. All these steps were encompassed in an iterative cycle, following an agile methodology. This iterative approach led to the refinement of the requirements and design when the methods were understood in greater detail as the project progressed.

To conclude, the development process of the new functionality added to the library proved very insightful. *Scikit-fda* is a large project resulting from a collaborative effort, with contributions from a diverse team of developers over an extended period of time. As a result, I gained valuable experience in collaborating with other developers, working with an extensive codebase and using continuous integration and testing tools. Moreover, the statistical and machine learning methods implemented in this project are at the forefront of the FDA research (specially PLS). To successfully codify these methods, extensive investigation and review of the literature was needed, which greatly improved my research capabilities. Additionally, the development of the functionality required acquiring in-depth knowledge of the *NumPy*, *SciPy* and *scikit-learn* models since *scikit-fda* is integrated in the scientific Python ecosystem and conforms to the *scikit-learn* API. Finally, as described in Chapter 3, I used many different tools and utilities to help generate high-quality, self-documented, maintainable and performant code.

Bibliography

[1] H.-G. Müller, S. Wu, A. Diamantidis, N. Papadopoulos, and J. Carey, “Reproduction is adapted to survival characteristics across geographically isolated medfly populations,” *Proceedings. Biological sciences / The Royal Society*, vol. 276, pp. 4409–16, 09 2009.

[2] X. Leng and H.-G. Müller, “Classification using functional data analysis for temporal gene expression data,” *Bioinformatics*, vol. 22, pp. 68–76, 10 2005.

[3] A. Harrison, W. Ryan, and K. Hayes, “Functional data analysis of joint coordination in the development of vertical jump performance,” *Sports Biomechanics*, vol. 6, no. 2, pp. 199–214, 2007. PMID: 17892096.

[4] S. Ratcliffe, L. Leader, and G. Heller, “Functional data analysis with application to periodically stimulated foetal heart rate data. i: Functional regression,” *Statistics in medicine*, vol. 21, pp. 1103–14, 04 2002.

[5] M. Hermanussen, “Auxology: An update,” *Hormone research in pædiatrics*, vol. 74, pp. 153–64, 07 2010.

[6] A. Laukaitis, “Functional data analysis for cash flow and transactions intensity continuous-time prediction using hilbert-valued autoregressive processes,” *European Journal of Operational Research*, vol. 185, no. 3, pp. 1607–1614, 2008.

[7] R. Bapna, W. Jank, and G. Shmueli, “Price formation and its dynamics in online auctions,” *Decision Support Systems*, vol. 44, no. 3, pp. 641–656, 2008.

[8] L. L. Koenig, J. C. Lucero, and E. Perlman, “Speech production variability in fricatives of children and adults: results of functional data analysis.,” *The Journal of the Acoustical Society of America*, vol. 124 5, pp. 3158–70, 2008.

[9] S. Lee, D. Byrd, and J. Krivokapic, “Functional data analysis of prosodic effects on articulatory timing,” *The Journal of the Acoustical Society of America*, vol. 119, pp. 1666–71, 04 2006.

[10] P. Kokoszka and M. Reimherr, *Introduction to Functional Data Analysis*. Chapman & Hall/CRC Texts in Statistical Science, CRC Press, 2017.

[11] J. Ramsay and B. Silverman, *Functional Data Analysis*. Springer Series in Statistics, Springer New York, 2013.

[12] C. Preda and G. Saporta, “Pls regression on a stochastic process,” *Computational Statistics and Data Analysis*, vol. 48, no. 1, pp. 149–158, 2005. Partial Least Squares.

[13] M. Febrero-Bande, P. Galeano, and W. González-Manteiga, “Functional principal component regression and functional partial least-squares regression: An overview and a comparative study,” *International Statistical Review*, vol. 85, no. 1, pp. 61–83, 2017.

[14] J. O. Ramsay, S. Graves, and G. Hooker, “fda: Functional data analysis,” 2022. `https://CRAN.R-project.org/package=fda`.

[15] M. Febrero-Bande and M. Oviedo de la Fuente, “Statistical computing in functional data analysis: The R package fda.usc,” *Journal of Statistical Software*, vol. 51, no. 4, pp. 1–28, 2012.

[16] T. Sauer, *Numerical Analysis*. Pearson, 2018.

[17] R. Noonan and H. Wold, “Nipals path modelling with latent variables,” *Scandinavian Journal of Educational Research - SCAND J EDUC RES*, vol. 21, pp. 33–61, 01 1977.

[18] S. De Jong, “Simpls: an alternative approach to partial least squares regression,” *Chemometrics and intelligent laboratory systems*, vol. 18, no. 3, pp. 251–263, 1993.

[19] J. Wegelin, “A survey of partial least squares (pls) methods, with emphasis on the two-block case,” *Technical report*, 04 2000.

[20] A. Delaigle and P. Hall, “Methodology and theory for partial least squares applied to functional data,” *The Annals of Statistics*, vol. 40, no. 1, pp. 322 – 352, 2012.

[21] J. Goldsmith, F. Scheipl, L. Huang, J. Wrobel, C. Di, J. Gellar, J. Harezlak, M. W. McLean, B. Swihart, L. Xiao, C. Crainiceanu, and P. T. Reiss, “refund: Regression with functional data,” 2023. https://CRAN.R-project.org/package=refund.

[22] K. H. Liland, B.-H. Mevik, and R. Wehrens, “pls: Partial least squares and principal component regression,” 2023. https://CRAN.R-project.org/package=pls.

[23] S. Golovkine, “Fdapy: a Python package for functional data,” *CoRR*, vol. abs/2101.11003, 2021.

[24] J. D. Tucker, “fdasrsf: Functional data analysis using the square root slope framework,” 2020. https://pypi.org/project/fdasrsf.

[25] C. Ramos-Carreño, A. Suárez, J. L. Torrecilla, M. Carbajo Berrocal, P. Marcos Manchón, P. Pérez Manso, A. Hernando Bernabé, D. García Fernández, Y. Hong, P. M. Rodríguez-Ponga Eyriès, A. Sánchez Romero, E. Petrunina, A. Castillo, D. Serna, and R. Hidalgo, “GAA-UAM/scikit-fda: Functional Data Analysis in Python,” Oct. 2019. https://github.com/GAA-UAM/scikit-fda.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] G. van Rossum, B. Warsaw, and N. Coghlan, “Style guide for Python code,” PEP 8, 2001–2013. https://www.python.org/dev/peps/pep-0008/.

[28] D. Goodger and G. van Rossum, “Docstring conventions,” PEP 257, 2001. https://www.python.org/dev/peps/pep-0257/.

[29] N. Sobolev, “wemake-python-styleguide: The strictest and most opinionated python linter ever,” 2023. https://pypi.org/project/wemake-python-styleguide.

[30] T. Ziade, “fake8: The modular source code checker, pep8 pyflakes and co,” 2022. https://pypi.org/project/flake8.

[31] G. van Rossum, J. Lehtosalo, and L. Langa, “Type hints,” PEP 484, 2014–2015. https://www.python.org/dev/peps/pep-0484/.

[32] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux, “API design for machine learning software: experiences from the scikit-learn project,” 2013.

[33] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[34] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020.

[35] G. Brandl, “Sphinx: Python documentation generator,” 2023. `https://pypi.org/project/sphinx`.

[36] O. Nájera, E. Larson, L. Estève, G. Varoquaux, L. Liu, J. Grobler, E. S. de Andrade, C. Holdgraf, A. Gramfort, M. Jas, J. Nothman, O. Grisel, N. Varoquaux, E. Gouillart, M. Luessi, A. Lee, J. Vanderplas, T. Hoffmann, T. A. Caswell, B. Sullivan, A. Batula, jaeilepp, T. Robitaille, S. Appelhoff, P. Kunzmann, M. Geier, Lars, K. Sunden, D. Stańczak, and A. Y. Shih, “sphinx-gallery: v0.12.2,” Mar. 2023. `https://doi.org/10.5281/zenodo.3838216`.

[37] T. Crosley, “Python utility to sort imports,” 2023. `https://github.com/pycqa/isort`.

[38] L. Langa, “Black: the uncompromising Python code formatter,” 2023. `https://github.com/psf/black`.

[39] H. H. et all, “Pep 8 Python formatter,” 2023. `https://github.com/hhatto/autopep8`.

[40] J. Lehtosalo, “mypy: Optional static typing for Python,” 2023. `https://pypi.org/project/mypy`.

[41] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, “pytest,” 2004. `https://github.com/pytest-dev/pytest`.

[42] J. Fonseca, “Conversor from profiling output to a dot graph,” 2023. `https://github.com/jrfonseca/gprof2dot`.

[43] R. K. et al, “Line-by-line profiling for Python,” 2023. `https://github.com/pyutils/line_profiler`.

[44] A. Phatak and F. Hoog, “Exploiting the connection between pls, lanczos methods and conjugate gradients: Alternative proofs of some properties of pls,” *Journal of Chemometrics*, vol. 16, pp. 361 – 367, 07 2002.

[45] A. Höskuldsson, “Pls regression methods,” *Journal of Chemometrics*, vol. 2, no. 3, pp. 211–228,

1988.

[46] H. Wold, “Model construction and evaluation when theoretical knowledge is scarce,” pp. 47–74, 1980.

[47] H. Wold, “Soft modelling by latent variables: The non-linear iterative partial least squares (nipals) approach,” *Journal of Applied Probability*, vol. 12, no. S1, p. 117–142, 1975.

[48] L. Eldén, “Partial least-squares vs. lanczos bidiagonalization—i: analysis of a projection method for multiple regression,” *Computational Statistics and Data Analysis*, vol. 46, no. 1, pp. 11–31, 2004.

Appendices

A

Multivariate PLS

The multivariate NIPALS algorithm solves the multivariate PLS problem stated in Section 2.2. The first iteration of the algorithm looks for a pair of directions $(\mathbf{w}_1, \mathbf{c}_1)$ such that the covariance between the components $\operatorname{cov}^2(\mathbf{t}_1, \mathbf{u}_1) = \operatorname{cov}^2(\mathbf{X}\mathbf{w}_1, \mathbf{Y}\mathbf{c}_1)$ is maximized. Then, to calculates subsequent components, the data matrices **X** and **Y** are modified so that solving the same problem again leads to incorrelated components. That is to say, when one has all the components, they must fulfill $\operatorname{corr}(\mathbf{t}_i, \mathbf{t}_j) = 0$ and $\operatorname{corr}(\mathbf{u}_i, \mathbf{u}_j) = 0$ if $i \neq j$.

The deflation scheme is described in the following section, with the rest of the algorithm. A proof that the algorithm fulfills the condition stated in the previous paragraph can be found in [19]. Alternatively, a simpler proof, but only applicable to scalar Y blocks can be consulted in [44].

A.1. The steps of multivariate NIPALS

The NIPALS algorithm is an iterative method that extracts a pair of PLS components in each iteration. This algorithm can be broken down into four blocks. Understanding the work being done in each of these blocks is key to being able to generalize this algorithm for the functional case. In order to analyze this method, the entire algorithm has been provided in Algorithm A.1. We will now describe each one of the blocks in the algorithm:

1. **Weights calculation**. Consider $\mathbf{X}_{l-1}$ and $\mathbf{Y}_{l-1}$ (the data matrices at that iteration) as the inputs to this block, and $\mathbf{w}_l$, $\mathbf{c}_l$ as the outputs. Then the calculated $\mathbf{w}_l$ and $\mathbf{c}_l$ must maximize the covariance of the projections. That is to say,

$$(\mathbf{w}_l, \mathbf{c}_l) = \max_{\mathbf{w},\mathbf{c}} \arg \operatorname{cov}^2(\mathbf{X}_{l-1}\mathbf{w}, \mathbf{Y}_{l-1}\mathbf{c}). \tag{A.1}$$

   Moreover, these vectors must have norm one. It is not hard to see that the vector that fulfill said condition must also fulfill:

$$\mathbf{X}_{l-1}^{\mathsf{T}}\mathbf{Y}_{l-1}\mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{X}_{l-1}\mathbf{w}_l = \lambda_w \mathbf{w}_l \qquad \mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{X}_{l-1}\mathbf{X}_{l-1}^{\mathsf{T}}\mathbf{Y}_{l-1}\mathbf{c}_l = \lambda_c \mathbf{c}_l, \tag{A.2}$$

with the highest possible eigenvalues λ_w and λ_c. Proofs of these claims can be found in [45] and [19].

The norm one constraint plus the eigenvector characterization is enough to arrive at the calculation steps of Algorithm A.1.

2. **Scores calculation**. Score is the name that has been traditionally given to the PLS components. This has been done in the PLS literature to clearly differentiate the components from the weights. The scores are defined as the projection of the data along the extracted weights. In the multivariate setting, this translates into two matrix multiplications.

   The extracted vectors can be thought of as realizations of latent random variables that drive the observed random variables. This interpretation is explained in more detail in Section A.2.

3. **Choose latent variables for Y**. This step has been included as the deflation must be carried out in a slightly different manner when using PLS in a regression problem. The logic behind this change will be explained in Section A.4. The intuitive interpretation of this change is that to perform the regression, we assume that there is only one set of latent variables that drive both the **X** and **Y** blocks.

   It is important to notice that, when the deflation mode is not regression (we will name this setting "canonical"), the algorithm is symmetric for **X** and **Y**. That is to say, if the data matrices for the X and Y block were interchanged, the results would be the same but for the opposite block.

4. **Loadings calculation**. The loadings are defined as the result of regressing the data matrices on the scores. That is to say, they are the result of adjusting two linear regression models by least squares. In the canonical case:

$$\mathbf{X}_{l-1} = \mathbf{t}_l \mathbf{p}_l^{\intercal} + \mathbf{E}_l \quad \text{and} \quad \mathbf{Y}_{l-1} = \mathbf{u}_l \mathbf{q}_l^{\intercal} + \mathbf{F}_l. \tag{A.3}$$

5. **Data deflation**. At the end of each iteration, the data matrices are changed. The goal of this update is to remove the "information" that can already been explained by the extracted components. In particular, the data matrices for the next iteration are the residuals left after fitting the models in (A.3).

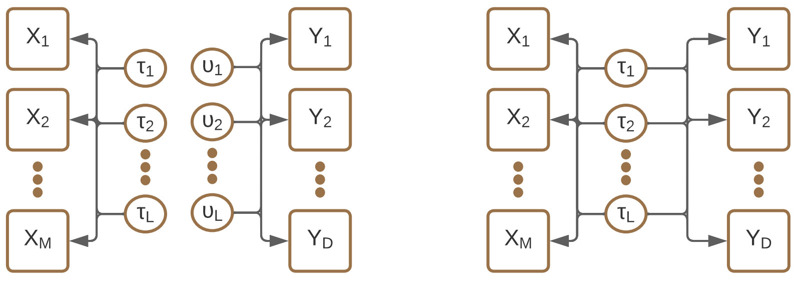
A.2. Latent Variables Interpretation

One of the most intuitive ways to understand the NIPALS method is to assume a particular model for the random variables. In this section, we will denote by $X = (X_1, \dots, X_M)^{\intercal}$ the random vector (with M features) that is sampled to obtain the data for the **X** matrix. Similarly, $Y = (Y_1, \dots, Y_D)^{\intercal}$ (with D features) will represent the underlying random variable for the other block.

The technical details of these models can be consulted in [46] and [47]. For the arguments that

```
$\mathbf{X}_0 \leftarrow \mathbf{X}, \quad \mathbf{Y}_0 \leftarrow \mathbf{Y}$
$l \leftarrow 1$
while $l < L$ do
    # Weights calculation
    $\mathbf{w}_l \leftarrow$ normalized dominant eigenvector of $\mathbf{X}_{l-1}^{\mathsf{T}}\mathbf{Y}_{l-1}\mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{X}_{l-1}$
    $\mathbf{c}_l \leftarrow \mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{X}_{l-1}\,\mathbf{w}_l / \|\mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{X}_{l-1}\,\mathbf{w}_l\|$

    # Scores calculation
    $\mathbf{t}_l \leftarrow \mathbf{X}_{l-1}\mathbf{w}_l$
    $\mathbf{u}_l \leftarrow \mathbf{Y}_{l-1}\mathbf{c}_l$

    # Choose latent variables for Y
    if deflation_mode == regression then
        $\mathbf{d} \leftarrow \mathbf{t}_l$
    else
        $\mathbf{d} \leftarrow \mathbf{u}_l$
    end

    # Loadings calculation
    $\mathbf{p}_l \leftarrow \mathbf{X}_{l-1}^{\mathsf{T}}\mathbf{t}_l / (\mathbf{t}_l^{\mathsf{T}}\mathbf{t}_l)$
    $\mathbf{q}_l \leftarrow \mathbf{Y}_{l-1}^{\mathsf{T}}\mathbf{d} / (\mathbf{d}^{\mathsf{T}}\mathbf{d})$

    # Data deflation
    $\mathbf{X}_l \leftarrow \mathbf{X}_{l-1} - \mathbf{t}_l\mathbf{p}_l^{\mathsf{T}}$
    $\mathbf{Y}_l \leftarrow \mathbf{Y}_{l-1} - \mathbf{d}\mathbf{q}_l^{\mathsf{T}}$
    $l \leftarrow l + 1$
end
```

Algorithm A.1: NIPALS algorithm for PLS.



(a) Canonical (b) Regression

Figure A.1: Latent variables models depending on the latent variable chosen for **Y** (step 3 of NIPALS).

we need in this work we do not need a precise characterization of all relations between the random variables in the model. However, it is very useful to visualize this model. With that goal in mind, two diagrams have been included in Figure A.1.

We shall focus on the canonical case first. In this case, pairs of components are extracted. That is to say, in the first iteration of NIPALS, the two random variables τ_1 and υ_1 will be extracted. The result is the vectors $\mathbf{t}_1$ and $\mathbf{u}_1$, which contain the N samples of these random variables. In particular, these variables are obtained as linear combination of the observed ones in each block. The weight vectors ($\mathbf{w}_l$ and $\mathbf{c}_l$) contain these coefficients.

The regression case is slightly different. In that scenario, only one latent variable is extracted in each iteration. This will prove to be very useful when this method is applied to a regression problem. We will cover the details in Section A.4. However, the intuitive idea is the following. By having only one set of latent variables, we can use the weights to obtain the latent variables from **X** and, then, we can use the loadings of the **Y** block to estimate **Y** from the latent variables.

A.3. Multivariate NIPALS properties

In order to analyze this algorithm, the vectors obtained in each iteration are arranged into matrices.

$$\begin{aligned}
\mathbf{W} &= (\mathbf{w}_1, \ldots, \mathbf{w}_L) \in \mathbb{R}^{M\times L} & \mathbf{C} &= (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \mathbb{R}^{D\times L} \\
\mathbf{T} &= (\mathbf{t}_1, \ldots, \mathbf{t}_L) \in \mathbb{R}^{N\times M} & \mathbf{U} &= (\mathbf{u}_1, \ldots, \mathbf{u}_L) \in \mathbb{R}^{N\times D} \\
\mathbf{P} &= (\mathbf{p}_1, \ldots, \mathbf{p}_L) \in \mathbb{R}^{M\times L} & \mathbf{Q} &= (\mathbf{q}_1, \ldots, \mathbf{q}_L) \in \mathbb{R}^{D\times L}
\end{aligned} \tag{A.4}$$

There are many interesting relationships between these matrices. However, in the interest of conciseness, we will only cover now the ones that are essential for applying the FPLS method. For a more complete list of properties, refer to [19]. We will now discuss the properties of the matrices obtained with the canonical deflation mode. The different properties that result form the regression deflation mode will be covered in the next section.

Firstly, due to the successive deflations, the following expression can be obtained for the data matrices:

$$\mathbf{X} = \mathbf{T}\mathbf{P}^{\mathsf{T}} + \mathbf{E} \qquad \mathbf{Y} = \mathbf{U}\mathbf{Q}^{\mathsf{T}} + \mathbf{F}, \tag{A.5}$$

where **E** and **F** are matrices whose entries get smaller as the number of components extracted increases.

Additionally, two rotation matrices can be defined. If one analyzes the weights extracted during the algorithm, it is easy to see that $\mathbf{w}_l$ and $\mathbf{c}_l$ are chosen using the data matrix in that iteration, not the original data matrices. As a consequence, $\mathbf{T} \neq \mathbf{XW}$ and $\mathbf{U} \neq \mathbf{YC}$ in general. That is to say, $\mathbf{W}$ are not the directions that we were looking for. Those directions are the rotations. Their definition is

$$\mathbf{R}_x = \mathbf{W}(\mathbf{P}^\mathsf{T}\mathbf{W})^{-1} \quad \text{and} \quad \mathbf{R}_y = \mathbf{C}(\mathbf{Q}^\mathsf{T}\mathbf{C})^{-1}. \tag{A.6}$$

These matrices do fulfill that

$$\mathbf{T} = \mathbf{XR}_x \quad \text{and} \quad \mathbf{U} = \mathbf{YR}_y. \tag{A.7}$$

Using equations (A.5) and (A.7), one can get $\mathbf{T}$ and $\mathbf{U}$ from the data matrices (using the rotations). And, conversely, it is possible to approximate the data matrices if $\mathbf{T}$ and $\mathbf{U}$ are known (using $\mathbf{P}$ and $\mathbf{Q}$). Looking back at the diagram in Figure A.1(a), this means that we can obtain the latent variables from the observed ones and vice-versa (approximately). Furthermore, the matrices used for this (rotations and loadings) are not realizations of random variables.

A.4. Multivariate PLS regression

Having understood the properties of PLS as a dimensionality reduction method, we can now apply them to a regression problem. One possibility would be to use the rotation matrices (see (A.6)) as a basis and then perform a standard linear regression with the components. That is to say, predict $\mathbf{U}$ using $\mathbf{T}$. Following this approach would mimic the steps that we took to define the application of FPCA to regression problems.

However, there is a more efficient and simple approach in this case. Moreover, this approach can easily be extended to the functional case rather simply. In particular, we can use the regression deflation mode defined in Algorithm A.1. By using this version of NIPALS, one has the latent variable model in Figure A.1(b). As we already anticipated in Section A.2, this is key to obtaining this regression method. Before providing the exact steps of this method, it is needed to analyze how the properties of the algorithm change with this deflation mode. An exhaustive list of these properties can be found in [45] and [19].

In our case, it suffices to provide the counterparts of equations (A.5), (A.6) and (A.7). Even though the matrices are different, in order to avoid complicating the notation, we will denote the matrices in the same manner as in the canonical mode. Then, one has:

$$\mathbf{X} = \mathbf{TP}^\mathsf{T} + \mathbf{E} \qquad \mathbf{Y} = \mathbf{TQ}^\mathsf{T} + \mathbf{F} \tag{A.8a}$$

$$\mathbf{R}_x = \mathbf{W}(\mathbf{P}^\mathsf{T}\mathbf{W})^{-1} \quad \text{and} \quad \mathbf{T} = \mathbf{XR}_x \tag{A.8b}$$

Now, from (A.8a) and using the X rotations, the following expression can be obtained:

$$\mathbf{Y} = \mathbf{T}\mathbf{Q}^{\mathsf{T}} + \mathbf{E} = \mathbf{X}\mathbf{W}(\mathbf{P}^{\mathsf{T}}\mathbf{W})^{-1}\mathbf{Q}^{\mathsf{T}} + \mathbf{F} = \mathbf{X}\mathbf{B} + \mathbf{F}, \tag{A.9}$$

where **B** is the regression matrix. It is immediate that this matrix has the expression

$$\mathbf{B} = \mathbf{W}(\mathbf{P}^{\mathsf{T}}\mathbf{W})^{-1}\mathbf{Q}^{\mathsf{T}}. \tag{A.10}$$

This regression method is known as PLS2 when the data matrix **Y** has more than 1 feature (column). If the response is univariate, it is known as PLS1. In the present work, we will not exploit any of the properties that are unique to PLS1. However, note that PLS1 has many interesting properties that do not hold for PLS2, making it a method worth studying in its own right. Some of these properties are covered in [48] and [44].

B
Roughness penalty

When modeling quantities that depend on a continuous parameter, one typically assumes underlying smoothness. To incorporate this assumption into multivariate techniques, one can incorporate a roughness penalization in the optimization criteria. In order to do so, the roughness of a function must be measured in some way. A natural way to quantify the roughness of a given function f is the norm of its derivatives. In particular, the L^2 norm of the second derivative is a commonly used penalty (see [11, p.84]). This penalization is defined as:

$$\mathrm{PEN}_2(f) = \int_I (D^2 f(s))^2 ds, \tag{B.1}$$

where D^2 denotes the second derivative operator and I is the domain of f.

For example a straight line will have a penalty of 0. On the contrary, a very noisy function will have a high curvature in many points, leading to a high absolute value of its second derivative. However, this notion of penalization proves insufficient in some cases. For example, consider a situation where f represents the position of a travelling object. It is not unreasonable to seek a smooth acceleration of said object, which would lead to penalizing the second derivative of the acceleration. Since the acceleration is, in turn, the second derivative of f, the goal would seek to minimize the norm of the fourth derivative of f.

Similar examples show that there might be reasonable reasons to penalize different derivatives, or even combinations of them. To accommodate these needs, we define the penalization for a linear differential operator with constant weights. Such an operator can be defined as

$$P = \sum_{k=0}^{M} w_k D^k. \tag{B.2}$$

That is to say, the operator P is a linear combination of the first M derivatives. Usually, M will be a small number since higher derivatives usually do not have a straightforward interpretation. The penalization for P can be defined as

$$\mathrm{PEN}_P(f) = \int_I (Pf(s))^2 ds, \tag{B.3}$$

With this definition in hand, the task now is to find how this penalization can be easily computed. As it has already been mentioned, *scikit-fda* provides two different ways to represent functional data: discretized and as a basis expansion. Therefore, this penalty must be possible to calculate in both cases.

If f is represented as a basis expansion in a basis $\{\phi_k\}_{k=1}^K$, we will denote $\mathbf{f} = (f_1, \dots, f_K)^\intercal$ its coefficients in said basis. Plugging the basis expansion into (B.3) and using both the linearity of the operator and the integral, one obtains:

$$\begin{aligned}\mathrm{PEN}_P(f) &= \int_I \left(P\left(\sum_{k=1}^K \phi_k f_k\right)(s)\right)^2 ds = \int_I \left(\left(\sum_{k=1}^K f_k P(\phi_k)\right)(s)\right)^2 ds = \\ &= \int_I \left(\sum_{k=1}^K f_k P(\phi_k)(s)\right)^2 ds = \int_I \left(\sum_{1\leq i,j\leq K} f_i f_j P(\phi_i)(s)P(\phi_j)(s)\right) ds = \\ &= \sum_{1\leq i,j\leq K} f_i f_j \int_I P(\phi_i)(s)P(\phi_j)(s) ds = \mathbf{f}^\intercal \mathbf{P}\mathbf{f},\end{aligned} \tag{B.4}$$

where $\mathbf{P}$ is the penalty matrix for the operator P in the basis ϕ. This matrix is defined as

$$\mathbf{P}_{ij} = \int_I P(\phi_i)(s)P(\phi_j)(s). \tag{B.5}$$

The usual approach to calculating these values is to first calculate the result of applying the linear operator to each basis function. Then the L^2 inner products can be calculated. Usually the number of basis functions is rather small. Therefore the $K \times K$ products required to populate $\mathbf{P}$ can be calculated rather quickly. Moreover, once $\mathbf{P}$ has been calculated, it can be reused to calculate penalties for any function expressed in the same basis.

To ensure that the computational complexity of this calculation is acceptable, a complexity analysis has been carried out (see Table B.1). Note that it has been assumed that the derivatives can be easily calculated from the basis. This holds true for most of the widely use basis such as Fourier or B-Splines. If that is not the case, the complexity of the first step would increase. However, as it can be seen, the bottleneck in terms of performance is the last step.

Additionally, for the additions and integrals, it has been assumed that all operations are performed using the basis expansion of the functions. Therefore, adding functions corresponds to adding their coefficients in the basis and calculating integrals corresponds to: $\int b(t)c(t)dt = \mathbf{b}^\intercal \mathbf{G}\mathbf{c}$. Here, $\mathbf{b}$ and $\mathbf{c}$ are the coefficient vectors of $b(t)$ and $c(t)$, and $\mathbf{G}$ is the Gram matrix of the basis. Therefore, addition is linear in K and, integration, quadratic.

The final complexity is $\mathcal{O}(K^4)$. This is usually manageable since K is small in most cases. Moreover, in many situations, additional properties of the basis functions can be applied to speed up the calculations. For example, the derivatives of the basis functions in a Fourier basis are multiples of the

Step	Time complexity analysis	Result
$d_i^m \leftarrow D^m(\phi_i)$	$M \times K$ derivations. Each of them is usually at most $\mathcal{O}(K)$.	$\mathcal{O}(M \times K^2)$
$s_i \leftarrow \sum_{m=1}^{M} w_m d_i^m$	$M \times K$ sums. Each of them is $\mathcal{O}(K)$.	$\mathcal{O}(M \times K^2)$
$\mathbf{P}_{i,j} \leftarrow \int_I s_i(s)s_j(s)ds$	$K \times K$ integrals. Each of them is $\mathcal{O}(K^2)$.	$\mathcal{O}(K^4)$

Table B.1: Complexity analysis of penalty matrix (**P**)

elements of the basis. Additionally, if the domain range of the basis is a multiple of the period, the basis functions are orthogonal. Using these two properties together, it can be deduced that the penalty matrix is diagonal and that there is a closed formula for the elements of its diagonal.

The process is similar when a function has been discretized. However, it is important to note that, in this case, we can only estimate the derivatives of the function. Since its values are not known in every point, the result can vary slightly depending on the method used. In this representation, the values of the function f are known in some given points $\{t_k\}_{k=1}^{K}$. In order to use a similar notation as before, we consider $\mathbf{f} = (f_1, \dots, f_K)^\intercal$, where $f_k = f(t_k)$

We can now consider a certain basis $\{\psi_k(t)\}_{k=1}^{K}$ such that $\psi_k(t_k) = 1$ but $\psi_k(t_j) = 0$ if $j \neq k$. As long as this property holds, $f = \sum_{k=1}^{K} f_k\psi_k$. This is true because when functions are discretized, they are equal if and only if they take the same value in each point of the grid. Since we have no information outside these points, we can ignore the values of $\sum_{k=1}^{K} f_k\psi_k$ in between grid points.

Using this basis expansion, the same procedure as before can be carried out and one obtains (B.4) and (B.5), Nevertheless, there is a crucial difference. While the number of basis is usually rather small, the number of discretization points can be considerably larger. One might only need 20 of 50 basis functions to express a dataset but dense discretization grids can have upwards of 1000 points. Therefore, it can be easily seen that it is impossible to use the same procedure described as before since a time complexity of K^4 is not acceptable.

However, $\{\psi_k\}_{k=1}^{K}$ is not an arbitrary basis. We known the values of the basis functions in the grid points, and we can take advantage of this insight to significantly accelerate the process. Firstly, it is imperative to decide how to calculate the derivatives of these functions. A reasonable approach is to use the finite difference formulas [16, p. 174]. These fit our problem particularly well since they only use the values of the functions in a grid. For example, assuming that the points are equally spaced, the second derivative of f in t_k $(1 < k < K)$ can be estimated as

$$f''(x_k) \approx \frac{1}{h^2}\left(f(x_{k-1}) - 2f(x_k) + f(x_{k+1})\right) = \frac{1}{h^2}(f_{k-1} - 2f_k + f_{k+1}) = (\mathbf{d}_k^2)^\intercal \mathbf{f}, \tag{B.6}$$

where h is the distance between the grid points and $\mathbf{d}_k^2$ is a vector that can be defined as:

$$\mathbf{d}_k^2 = \left(0, \dots 0, \frac{1}{h^2}, \frac{-2}{h^2}, \frac{1}{h^2}, 0, \dots, 0\right)^\intercal . \tag{B.7}$$

Note that we have, so far, ignored how the derivatives can be calculated in the borders of the grid. This is not a problem, however, since there are formulas that provide an approximation of the derivative in such cases. With this in mind, it is possible to calculate the second derivative of f in all points by multiplying **f** by a matrix $\mathbf{D}_2$ (second derivative) whose k-th row is $\mathbf{d}_k^2$. That is to say

$$\big(f''(t_1), \dots, f''(t_K)\big) = \mathbf{D}_2\mathbf{f}. \tag{B.8}$$

Using this last formula, we can calculate the derivative of each function of the basis $\{\psi_k\}_{k=1}^K$. The coefficient vector for the k-th basis function is a vector with a 1 in the k-th position. Therefore, by stacking all these vectors vertically, the $K \times K$ identity matrix is obtained and one has the following equation

$$\begin{pmatrix} \psi_1''(t_1) & \dots & \psi_1''(t_K) \\ \vdots & \ddots & \vdots \\ \psi_K''(t_1) & \dots & \psi_K''(t_K) \end{pmatrix} = \mathbf{D}_2\mathbf{I}_K = \mathbf{D}_2. \tag{B.9}$$

Since it is possible to consider approximations of this kind for derivatives of any order (see [16, ch. 4]), we can now accomplish the first step described in Table B.1. Moreover, as the coefficients that form the differentiation weights vectors ($\mathbf{d}_k^2$) have a simple closed formula, the complexity of calculating the derivatives of all the basis functions is $\mathcal{O}(M \times K^2)$, where M is the number of derivatives required.

With these results, calculating the result of applying the operator P to all $\{\psi_k\}_{k=1}^K$ can be performed by adding the matrices $\mathbf{D}_m$ multiplied by the operator weights. These matrices are defined in a similar fashion to $\mathbf{D}_2$. There are closed formulas for the differential weights of any order [16]. The result of this step is a matrix **S** such that

$$\mathbf{S} = \sum_{m=1}^{M} w_m \mathbf{D}_m. \tag{B.10}$$

The complexity of this operation is $\mathcal{O}(M \times K^2)$.

Both these first steps are not faster than before. However, knowing the structure of the matrix **S** will be highly beneficial in the third step. Let us consider $\{\mathbf{s}_k\}_{k=1}^K$ the rows of **S**. Then, one has that $\mathbf{s}_k = \sum_{m=1}^M w_m \mathbf{d}_k^m$. Since the derivatives of a point t_k are calculated using a small number of grid points around t_k, most of the entries of $\mathbf{s}_k$ will be zeroes. Note that now many elements around t_k are used in this calculation is a configurable parameter. There are different differentiation rules that use a different amount of points around t_k.

We will refer to how many points to each side are considered as the spread of the differentiation weights. For example, the integration weight in (B.7) has a spread of 1 since it uses one point before t_k and another point after t_k. Therefore, we can write $\operatorname{spread}(\mathbf{d}_k^2) = 1$. It can be easily seen that $\operatorname{spread}(\mathbf{s}_k) \leq \operatorname{máx}_{1 \leq m \leq M} \operatorname{spread}(\mathbf{d}_k^m)$. The spread of the differentiation weights depends on the particular formulas used to calculate them (It is possible to use more points to obtain smoother estimations). However, for the sake of simplicity, we can assume that the spread is bounded by $2M$. Note that, outside of the borders, the spread has an upper bound of M. Nevertheless, for convenience we will apply the $2M$ bound to all points

This result has a huge effect on the time complexity of the calculation of $\mathbf{P}_{i,j}$. In order to understand the reason behind this, it is crucial to define how $\mathbf{P}_{i,j}$ should be computed. In this case, the integral is discretized in the same grid as the functions using some integration weights $\{a_k\}_{k=1}^{K}$. That is to say:

$$\mathbf{P}_{i,j} = \int P(\psi_i)(s) \int P(\psi_j)(s)ds = \sum_{k=1}^{K} a_k \psi_i(t_k)\psi_j(t_k) = \sum_{k=1}^{K} \nu_k (\mathbf{s}_i)_k (\mathbf{s}_j)_k \tag{B.11}$$

Using the spread, it can be easily seen that if $|i-j| > \text{spread}(s_i) + \text{spread}(s_j)$, then $\mathbf{P}_{i,j} = 0$. Using the upper bound defined before,

$$|i-j| > 4M \implies \mathbf{P}_{i,j} = 0. \tag{B.12}$$

In most cases, M is rather small, Therefore, this shows that when the grid has many points, we only need to calculate the values of $\mathbf{P}_{i,j}$ close to the diagonal. This alone reduces the number of integrals needed to calculate to $\mathcal{O}(M \times K)$. However, this optimization alone is not enough. If each integral was calculated naïvely, it would have a time complexity of $\mathcal{O}(K^2)$, bringing the overall complexity to $\mathcal{O}(M \times K^3)$. As soon as K increases past 100 points, this is too slow.

However, we can reuse the idea used to reduce the number of integrals to speedup their calculation. In particular, from (B.11), one can derive

$$\mathbf{P}_{i,j} = \sum_{k=1}^{K} a_k (\mathbf{s}_i)_k (\mathbf{s}_j)_k = \sum_{k \in \Delta_{i,j}} a_k (\mathbf{s}_i)_k (\mathbf{s}_j)_k, \tag{B.13}$$

where $\Delta_{i,j} = \{k : i - \text{spread}(\mathbf{s}_i) \leq k \leq j + \text{spread}(\mathbf{s}_j) \text{ and } 1 \leq k \leq K\}$ if $i < j$ and $\Delta_{i,j} = \Delta_{j,i}$ otherwise.

Since the spreads are bounded by $2M$, when $|i-j| < 4M$, $|\Delta_{i,j}| \leq 6M$. Therefore, for the pairs of points where we have to do the integral, we need only consider $6M$ points. This optimization brings the time complexity of the integral down to $\mathcal{O}(M^2)$. Using this bound, the total complexity of this step is brought down to $\mathcal{O}(K \times M^3)$. As a summary, we present the counterpart of Table B.1 in Table B.2. Since $M << K$, the overall time complexity of the entire calculation of the $\mathbf{P}$ matrix is $\mathcal{O}(M \times K^2)$.

To conclude, in this section we have explored how to estimate the roughness of a function f , whether it is given as a basis expansion or in a discretized form. Following the same notation as in this section, $\mathbf{f}$ represents the basis coefficients in the former and the values in the grid in the latter. Additionally, P represents the differential operator whose norm is defined as the roughness of the function.

Step	Grid operation	Time complexity
$d_k^m \leftarrow D^m(\psi_k)$	$\mathbf{d}_k^m \leftarrow$ derivation weights for the m-th derivative in the k-th point.	$\mathcal{O}(M \times K^2)$
$s_k \leftarrow \sum_{m=1}^{M} w_m d_k^m$	$\mathbf{s}_k \leftarrow \sum_{m=1}^{M} w_m \mathbf{d}_k^m$	$\mathcal{O}(M \times K^2)$
$\mathbf{P}_{i,j} \leftarrow \int_I s_i(s) s_j(s) ds$	$\mathbf{P}_{i,j} \leftarrow \sum_{k \in \Delta_{i,j}} \nu_k (\mathbf{s}_i)_k (\mathbf{s}_j)_k$	$\mathcal{O}(K \times M^3)$

Table B.2: Complexity analysis of penalty matrix (**P**) for discretized function

With these definitions in mind, we have derived a method that computes in a reasonable time a matrix **P** that depends on the basis selected (if f is given in a basis expansion) or the discretization grid (if f is discretized) and fulfills

$$\text{PEN}_P(f) = \mathbf{f}^\mathsf{T}\mathbf{P}\mathbf{f}. \tag{B.14}$$

C

Gantt diagram

In Figure C.1, a Gantt diagram of project is included. This diagram shows the organization of the different tasks that have been carried out and their relationships with each other. The tasks have been separated into three groups: the research of the theoretical framework, the implementation of the functionality into *scikit-fda* and the writing of this document.

The first step in each topic was researching the scientific literature on it. As a result, several tasks have been created for the different theoretical aspects applied in this work. Once a certain method, algorithm or approach had been understood, the process of incorporating it into *scikit-fda* could start. Within the *scikit-fda* task group, milestones have been associated with the completion of each feature.

The last part of this project corresponds to the elaboration of this document. In this case, the subtasks have been associated with each version of the document. It has been decided not to show the dependencies between these subtasks, as most of them depend on each other to achieve a coherent document. Finally, a milestone has been associated with the final deliverable. This milestone depends on all tasks in this task group. However, to prevent clutter in the diagram, those relationships have not been drawn.

On the other hand, in section 3.3.1, three phases for the project were identified. The initial planning phase corresponds to the first 5 weeks of the Gantt diagram. The sprints took place from week 6 to week 34 and, on week 34, the effort was shifted to consolidating the already achieved results. On the library development side, this corresponds to the creation of examples and bugfixes and improvements tasks. Regarding the document creation, during this last phase, all tasks were reopened and effort was spent on finalizing each section and revising them.

The diagram in Figure C.1 shows the time allocated for each task during each week of the project. Week one corresponds to the week starting on the 11 of September 2022.

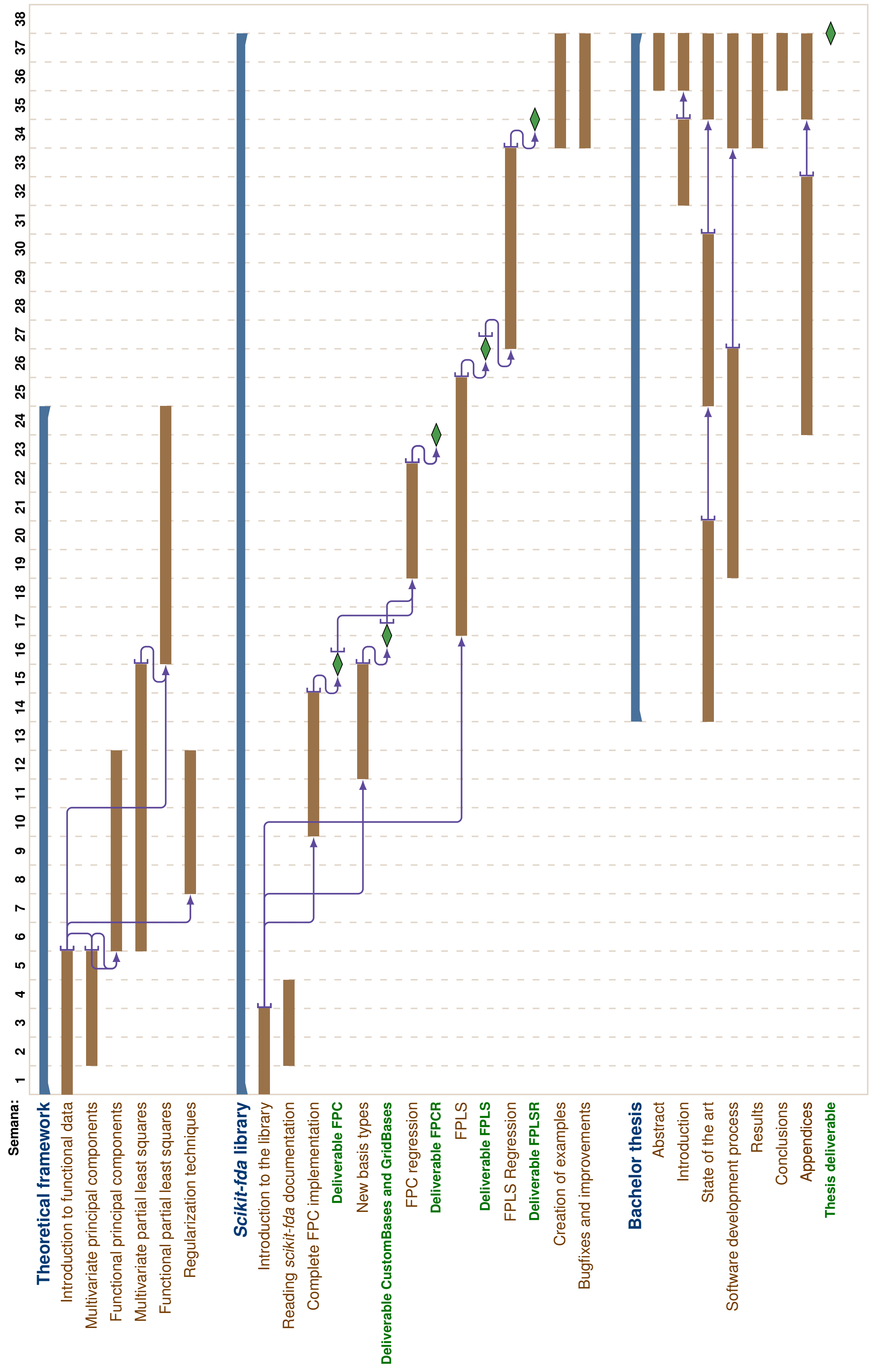


Figure C.1: Gantt diagram of the project

D

Additional Design Diagrams

In this appendix, detailed design diagrams are presented. Unlike hte ones included in Section 3.2, these diagrams include all fields and operations of each depicted class. Abstract classes have been labeled with an “A” next to their name and their name is typeset in italics. Also note that, in the FPLS diagram, `InputType` is a type alias for the input type of this class, which is equivalent to `FData | NDArrayFloat`.

The only diagram that includes significantly more details than the corresponding diagram in Section 3.2 is the FPLS diagram (Figure D.3). The class *`FPLSBlock`* and its specializations for multivariate, grid and basis data were not covered in the design section. These classes are only used within the `FPLS` class to simplify the implementation and avoid code repetition. In fact, they are private and not accessible from other modules. As it was covered in the description of FPLS as a dimensionality reduction technique (see Section 2.2.1), this method is symmetric for the two blocks X and Y. As a result, many operations have to be performed for both blocks in the same manner. These operations have been delegated to the *`FPLSBlock`*. Therefore, by having one instance of this class for each block, these task can be performed by calling the corresponding instance. Additionally, since these operations must be performed differently depending on the nature of the data, there are three implementations of this class, overloading the abstract methods.

The UML class diagrams are included in the following pages.

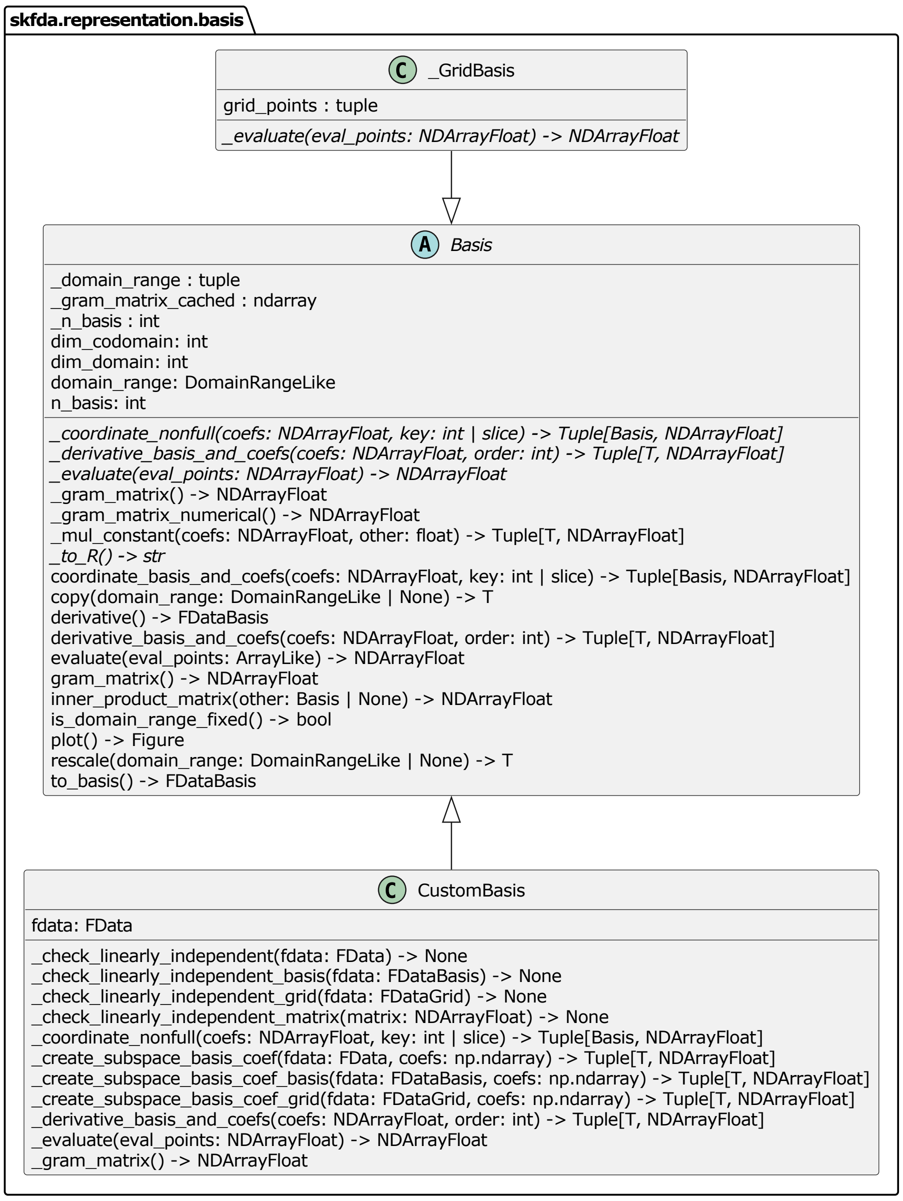

skfda.representation.basis
C _GridBasis
grid_points : tuple
_evaluate(eval_points: NDArrayFloat) -> NDArrayFloat
A Basis
_domain_range : tuple
_gram_matrix_cached : ndarray
_n_basis : int
dim_codomain: int
dim_domain: int
domain_range: DomainRangeLike
n_basis: int
_coordinate_nonfull(coefs: NDArrayFloat, key: int | slice) -> Tuple[Basis, NDArrayFloat]
_derivative_basis_and_coefs(coefs: NDArrayFloat, order: int) -> Tuple[T, NDArrayFloat]
_evaluate(eval_points: NDArrayFloat) -> NDArrayFloat
_gram_matrix() -> NDArrayFloat
_gram_matrix_numerical() -> NDArrayFloat
_mul_constant(coefs: NDArrayFloat, other: float) -> Tuple[T, NDArrayFloat]
_to_R() -> str
coordinate_basis_and_coefs(coefs: NDArrayFloat, key: int | slice) -> Tuple[Basis, NDArrayFloat]
copy(domain_range: DomainRangeLike | None) -> T
derivative() -> FDataBasis
derivative_basis_and_coefs(coefs: NDArrayFloat, order: int) -> Tuple[T, NDArrayFloat]
evaluate(eval_points: ArrayLike) -> NDArrayFloat
gram_matrix() -> NDArrayFloat
inner_product_matrix(other: Basis | None) -> NDArrayFloat
is_domain_range_fixed() -> bool
plot() -> Figure
rescale(domain_range: DomainRangeLike | None) -> T
to_basis() -> FDataBasis
C CustomBasis
fdata: FData
_check_linearly_independent(fdata: FData) -> None
_check_linearly_independent_basis(fdata: FDataBasis) -> None
_check_linearly_independent_grid(fdata: FDataGrid) -> None
_check_linearly_independent_matrix(matrix: NDArrayFloat) -> None
_coordinate_nonfull(coefs: NDArrayFloat, key: int | slice) -> Tuple[Basis, NDArrayFloat]
_create_subspace_basis_coef(fdata: FData, coefs: np.ndarray) -> Tuple[T, NDArrayFloat]
_create_subspace_basis_coef_basis(fdata: FDataBasis, coefs: np.ndarray) -> Tuple[T, NDArrayFloat]
_create_subspace_basis_coef_grid(fdata: FDataGrid, coefs: np.ndarray) -> Tuple[T, NDArrayFloat]
_derivative_basis_and_coefs(coefs: NDArrayFloat, order: int) -> Tuple[T, NDArrayFloat]
_evaluate(eval_points: NDArrayFloat) -> NDArrayFloat
_gram_matrix() -> NDArrayFloat

Figure D.1: Detailed design of the new `Basis` subclasses

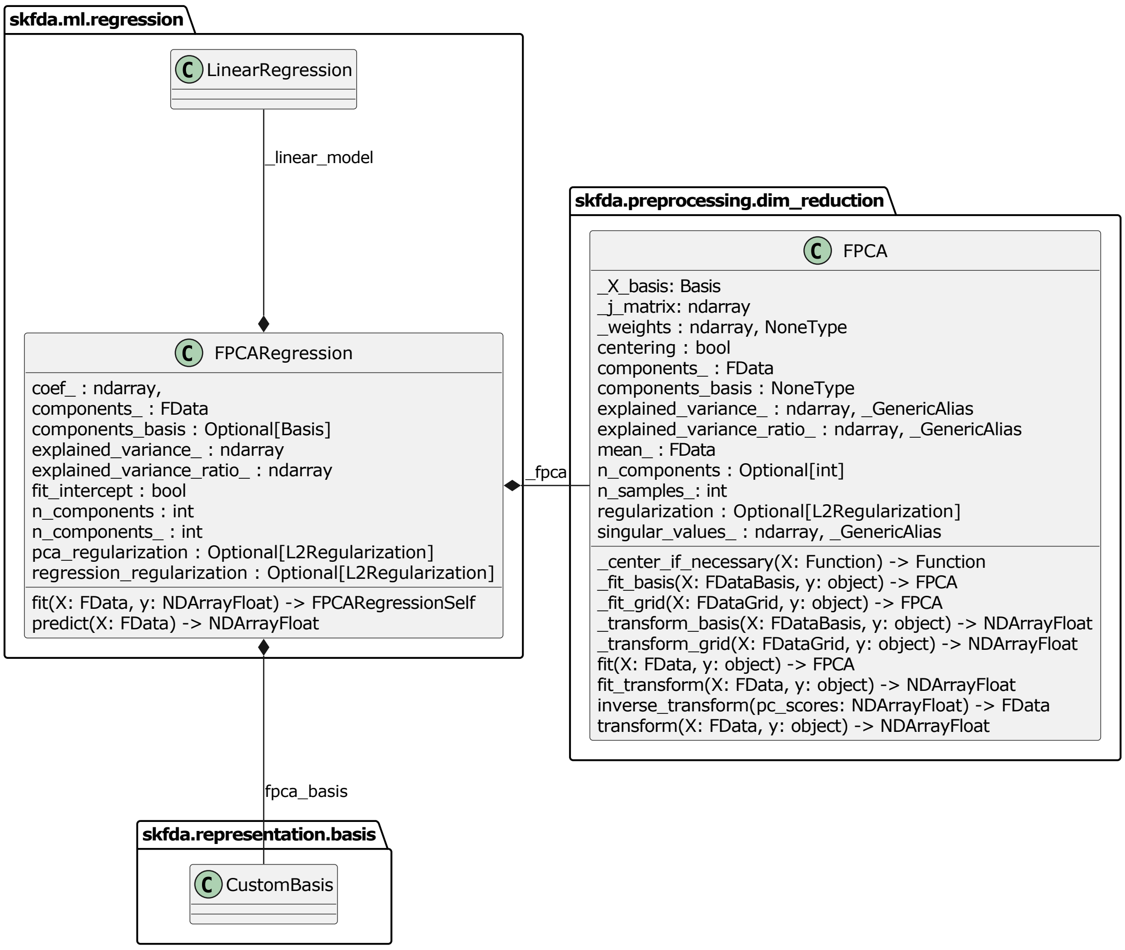

skfda.ml.regression
LinearRegression
_linear_model
FPCARegression
coef_ : ndarray,
components_ : FData
components_basis : Optional[Basis]
explained_variance_ : ndarray
explained_variance_ratio_ : ndarray
fit_intercept : bool
n_components : int
n_components_ : int
pca_regularization : Optional[L2Regularization]
regression_regularization : Optional[L2Regularization]
fit(X: FData, y: NDArrayFloat) -> FPCARegressionSelf
predict(X: FData) -> NDArrayFloat
_fpca
skfda.preprocessing.dim_reduction
FPCA
_X_basis: Basis
_j_matrix: ndarray
_weights : ndarray, NoneType
centering : bool
components_ : FData
components_basis : NoneType
explained_variance_ : ndarray, _GenericAlias
explained_variance_ratio_ : ndarray, _GenericAlias
mean_ : FData
n_components : Optional[int]
n_samples_: int
regularization : Optional[L2Regularization]
singular_values_ : ndarray, _GenericAlias
_center_if_necessary(X: Function) -> Function
_fit_basis(X: FDataBasis, y: object) -> FPCA
_fit_grid(X: FDataGrid, y: object) -> FPCA
_transform_basis(X: FDataBasis, y: object) -> NDArrayFloat
_transform_grid(X: FDataGrid, y: object) -> NDArrayFloat
fit(X: FData, y: object) -> FPCA
fit_transform(X: FData, y: object) -> NDArrayFloat
inverse_transform(pc_scores: NDArrayFloat) -> FData
transform(X: FData, y: object) -> NDArrayFloat
fpca_basis
skfda.representation.basis
CustomBasis

Figure D.2: Detailed design of the FPCA and FPCR functionality

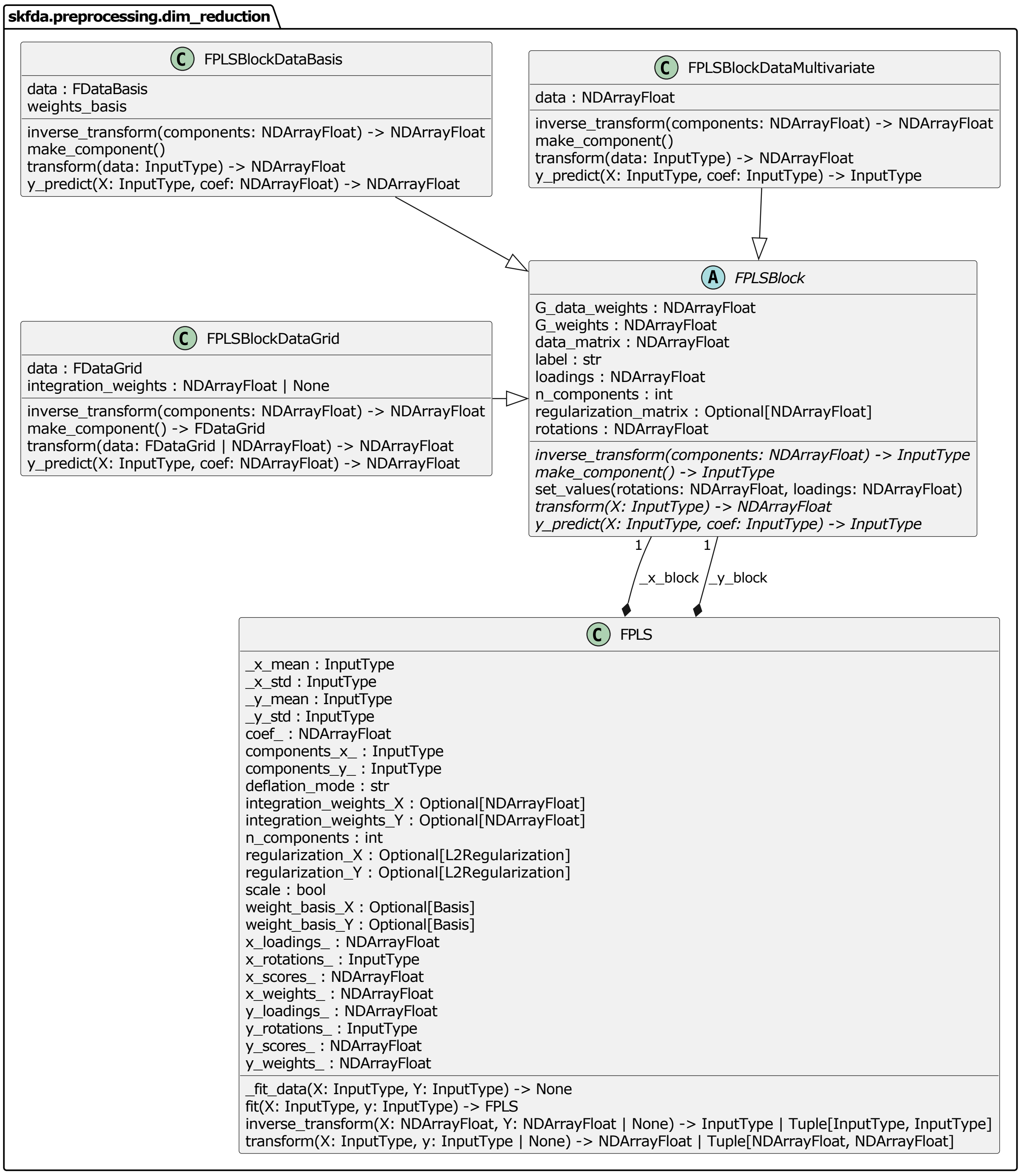

skfda.preprocessing.dim_reduction
C FPLSBlockDataBasis
data : FDataBasis
weights_basis
inverse_transform(components: NDArrayFloat) -> NDArrayFloat
make_component()
transform(data: InputType) -> NDArrayFloat
y_predict(X: InputType, coef: NDArrayFloat) -> NDArrayFloat
C FPLSBlockDataMultivariate
data : NDArrayFloat
inverse_transform(components: NDArrayFloat) -> NDArrayFloat
make_component()
transform(data: InputType) -> NDArrayFloat
y_predict(X: InputType, coef: InputType) -> InputType
A FPLSBlock
G_data_weights : NDArrayFloat
G_weights : NDArrayFloat
data_matrix : NDArrayFloat
label : str
loadings : NDArrayFloat
n_components : int
regularization_matrix : Optional[NDArrayFloat]
rotations : NDArrayFloat
inverse_transform(components: NDArrayFloat) -> InputType
make_component() -> InputType
set_values(rotations: NDArrayFloat, loadings: NDArrayFloat)
transform(X: InputType) -> NDArrayFloat
y_predict(X: InputType, coef: InputType) -> InputType
C FPLSBlockDataGrid
data : FDataGrid
integration_weights : NDArrayFloat | None
inverse_transform(components: NDArrayFloat) -> NDArrayFloat
make_component() -> FDataGrid
transform(data: FDataGrid | NDArrayFloat) -> NDArrayFloat
y_predict(X: InputType, coef: NDArrayFloat) -> NDArrayFloat
1
1
_x_block
_y_block
C FPLS
_x_mean : InputType
_x_std : InputType
_y_mean : InputType
_y_std : InputType
coef_ : NDArrayFloat
components_x_ : InputType
components_y_ : InputType
deflation_mode : str
integration_weights_X : Optional[NDArrayFloat]
integration_weights_Y : Optional[NDArrayFloat]
n_components : int
regularization_X : Optional[L2Regularization]
regularization_Y : Optional[L2Regularization]
scale : bool
weight_basis_X : Optional[Basis]
weight_basis_Y : Optional[Basis]
x_loadings_ : NDArrayFloat
x_rotations_ : InputType
x_scores_ : NDArrayFloat
x_weights_ : NDArrayFloat
y_loadings_ : NDArrayFloat
y_rotations_ : InputType
y_scores_ : NDArrayFloat
y_weights_ : NDArrayFloat
_fit_data(X: InputType, Y: InputType) -> None
fit(X: InputType, y: InputType) -> FPLS
inverse_transform(X: NDArrayFloat, Y: NDArrayFloat | None) -> InputType | Tuple[InputType, InputType]
transform(X: InputType, y: InputType | None) -> NDArrayFloat | Tuple[NDArrayFloat, NDArrayFloat]

Figure D.3: Detailed design of the FPLS functionality

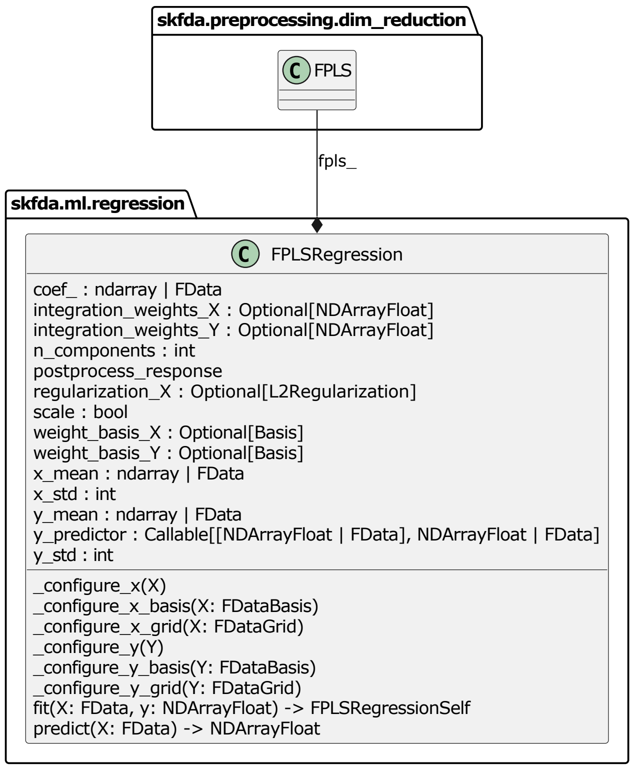

skfda.preprocessing.dim_reduction
C FPLS
fpls_
skfda.ml.regression
C FPLSRegression
coef_ : ndarray | FData
integration_weights_X : Optional[NDArrayFloat]
integration_weights_Y : Optional[NDArrayFloat]
n_components : int
postprocess_response
regularization_X : Optional[L2Regularization]
scale : bool
weight_basis_X : Optional[Basis]
weight_basis_Y : Optional[Basis]
x_mean : ndarray | FData
x_std : int
y_mean : ndarray | FData
y_predictor : Callable[[NDArrayFloat | FData], NDArrayFloat | FData]
y_std : int
_configure_x(X)
_configure_x_basis(X: FDataBasis)
_configure_x_grid(X: FDataGrid)
_configure_y(Y)
_configure_y_basis(Y: FDataBasis)
_configure_y_grid(Y: FDataGrid)
fit(X: FData, y: NDArrayFloat) -> FPLSRegressionSelf
predict(X: FData) -> NDArrayFloat

Figure D.4: Detailed design of the FPLSR functionality

UAM

Universidad Autónoma
de Madrid