Trabajo de Fin de Grado

Grado en Ingeniería de Tecnologías Industriales

Predicciones Intervalares Utilizando Optimización Convexa

Autor: José Torres Morón

Tutor: Teodoro Álamo Cantarero

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo de Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Predicciones Intervalares Utilizando Optimización Convexa

Autor:
José Torres Morón

Tutor:
Teodoro Álamo Cantarero
Catedrático de Universidad

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo de Fin de Grado: Predicciones Intervalares Utilizando Optimización Convexa

Autor: José Torres Morón

Tutor: Teodoro Álamo Cantarero

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2021

A mis padres y hermano

A todos lo que me han enseñado

Agradecimientos

No hubiera sido capaz de escribir un solo párrafo si no fuera por el apoyo incondicional que siempre me han brindado mis padres y hermano. Siempre han tratado de ayudarme en todo lo que estuviera en su mano no importa qué, incluso cuando mi trabajo eran jeroglíficos para ellos. Cada uno ha formado una parte imprescindible en mi vida y nada de lo que es este trabajo, ni nada de lo que soy, hubiera sido posible sin ellos.

A mis amigos que me han ayudado en todo momento a sobrellevar un verano encerrado en casa y me han sacado del pozo en los momentos más inesperados.

Y por supuesto, a Teodoro. Al profesor que hizo que me interesara por la ciencia de datos y que nunca dudó en perder el tiempo que fuera necesario para que aprendiera.

Resumen

El fin último de este trabajo consiste en desarrollar una metodología que provea de predicciones intervalares útiles. Es decir, para un regresor x_k, proveer del intervalo más pequeño posible que contenga a la salida y_k con una probabilidad especificada. Esta meta se apoya en dos pilares muy bien diferenciados.

En primer lugar, se propone una familia de funciones de disimilitud con la que, estimando la función de densidad empírica condicionada de una salida y_k, se obtiene la función de distribución en la que la noción de cuantil está altamente relacionada con la de intervalos de confianza. Estas funciones de disimilitud miden el grado de semejanza entre un vector z y un set de datos D, donde pequeños valores indican un alto grado de similitud y grandes valores implican un alto grado de disimilitud. En concreto, la familia de funciones de disimilitud propuesta consiste en un problema de optimización convexa invariante respecto a transformaciones afines. También se ha previsto la inclusión de información local al algoritmo mejorando, en la mayoría de los casos, considerablemente las predicciones.

El segundo pilar, (sin el cual el primero no sustentaría el fin último) aborda, con el método FISTA, el problema de optimización que se plantea en la definición de la función de disimilitud. Este algoritmo ha probado tener una ratio de convergencia mucho mayor que los tradicionales *gradient descent* básicos. No obstante, puede plantear algunos problemas prácticos los cuales se resuelven con el método *Restart*.

En definitiva, estos dos pilares sustentan una serie de funciones en Python que computan los algoritmos necesarios para las predicciones intervalares. Se plantean principalmente dos algoritmos: el primero obtiene una predicción intervalar dependiente de dos parámetros (γ y c). El segundo (basado en el primero) permite, dada γ, encontrar el c óptimo que reduce al máximo el intervalo. Este algoritmo asegura que no se viole la restricción de que la salida y_k pertenezca a dicho intervalo para una probabilidad dada.

Se ha buscado en todo momento la eficiencia tanto en los algoritmos como en la implementación en el código Python. En el apéndice se incluyen instrucciones que permiten usar con comodidad la librería de funciones.

Abstract

The ultimate goal of this work is to develop a methodology that provides useful interval predictions. That is, for a regressor x_k, to find the smallest possible interval containing the output value y_k with a given probability. This goal rests on two very distinct pillars.

Firstly, a family of dissimilarity functions is proposed with which to obtain conditional empirical distribution functions where the notion of quantile is highly related to that of confidence intervals. These dissimilarity functions measure the degree of similarity between a vector z and a data set D where small values indicate a high degree of similarity and large values imply a high degree of dissimilarity. Specifically, the proposed family of dissimilarity functions consists of a convex optimization problem invariant with respect to affine transformations. Provision has also been made for the inclusion of local information to the algorithm improving considerably the predictions.

The second pillar, (without which the first one would not support the ultimate goal) addresses with the FISTA method the optimization problem posed in the definition of the dissimilarity function. This algorithm has proven to have a much higher convergence rate than the traditional basic gradient descent. However, it can pose some practical problems which are solved with the Restart method.

In short, these two pillars support a series of Python functions that compute the algorithms needed for interval predictions. Two algorithms are mainly posed: the first one obtains an interval prediction dependent on two parameters (γ and c). The second one (based on the first one) allows, given γ, to find the optimal c that minimizes the interval. This algorithm ensures that the constraint that the output y_k belongs to the interval for a given probability is not violated.

Efficiency has been sought throughout, both in the algorithms and in the implementation in Python code. Instructions for convenient use of the function library have been included in the appendix.

Índice

Agradecimientos **ix**
Resumen **xi**
Abstract **xiii**
Índice **xv**
ÍNDICE DE TABLAS **xvii**
ÍNDICE DE FIGURAS **xviii**
1 Introducción **1**
1.1 Motivación *2*
1.2 Objetivos *2*
1.3 Organización de la memoria *2*
2 Cálculo de la Similitud y Disimilitud **5**
2.1 Introducción a las funciones de disimilitud *5*
2.2 Familia de funciones de disimilitud propuesta *6*
2.2.1 Definición formal 6
2.2.2 Ponderaciones en la función de coste 6
2.2.3 Función de disimilitud para matriz de pesos 7
2.3 Regresión con $J_y(z, D)$ *7*
2.3.1 Demanda eléctrica española 8
2.3.2 Predicción de la evolución del COVID-19 11
3 Optimización **15**
3.1 Problema a minimizar *15*
3.2 Restricciones de igualdad y enfoque dual *16*
3.3 Algoritmo Gradencial *19*
3.3.1 Gradiente del funcional dual 19
3.3.2 ISTA 20
3.3.3 Implementación de ISTA en Código Python 20
3.4 FISTA *20*
3.4.1 Implementación de FISTA en Código Python 21
3.4.2 Verificación de la implementación en Python 21
3.5 Optimización de la Optimización *22*
3.5.1 ISTA frente a FISTA 22
3.5.2 Eficiencia del código 24
3.5.3 Problema práctico *Rippling* 25

4 Predicciones Intervalares **29**
4.1 Función de Densidad Empírica *29*
4.2 Función de Densidad Empírica Condicionada *30*
4.2.1 Discretización del conjunto $\mathcal{Y}$ 30
4.2.2 Función de distribución empírica condicionada discreta de $fdec_{\gamma,c}$ 31
4.2.3 Cuantiles empíricos condicionados superior e inferior 31
4.3 Algoritmo 1. Estimación Intervalar *31*
4.3.1 Explicación del Algoritmo 1 32
4.3.2 Implementación del Algoritmo 1 en Python 34
4.3.3 Eficiencia del Algoritmo 1 35
4.4 Algoritmo 2. Valor óptimo de c *38*
4.4.1 Justificación del Algoritmo 2 38
4.4.2 Explicación del Algoritmo 2 39
4.4.3 Implementación del Algoritmo 2 en Python 40
4.5 Alcance del Fin Último *41*
4.5.1 Predicción intervalar de la demanda eléctrica española 41
4.5.2 Función de coste Q_{γ} 46

5 Conclusiones **47**

6 Librería de Funciones Python **49**
6.1 OPTIONS.py *50*
6.2 Interval_Estimation.py *51*
6.2.1 Importaciones necesarias 51
6.2.2 set_Yhat 51
6.2.3 *ec_pdf* 52
6.2.4 *algorithm1* 53
6.2.5 *algorithm2* 54
6.3 disimilarity_regression.py *55*
6.3.1 Importaciones Necesarias 56
6.3.2 *regression* 56
6.3.3 *training_validation_test_regression* 57
6.4 Estimacion_Intervalar_Demanda.py *59*
6.4.1 Importaciones Necesarias 59
6.4.2 Cuerpo del Código 59
6.5 Regresion_Demanda.py *62*
6.5.1 Importaciones Necesarias 62
6.5.2 Cuerpo del Código 62
6.6 Regresion_Covid.py *64*
6.6.1 Importaciones Necesarias 64
6.6.2 Cuerpo del Código 64
6.7 Fista.py *65*
6.7.1 Importaciones Necesarias 65
6.7.2 *lambda_opt* 65
6.7.3 *algorithm* 65
6.8 Fista_Based_Restart.py *66*
6.8.1 Importaciones necesarias 66
6.8.2 *lambda_opt* 66
6.8.3 *MaxCotInf* 67
6.8.4 *algorithm* 67
6.9 Momentum.py *68*
6.9.1 Importaciones Necesarias 68
6.9.2 Cuerpo del Código 68

Referencias **71**

ÍNDICE DE TABLAS

Tabla 1. Resultados de predicciones de la demanda eléctrica 11
Tabla 2. Resultados predicciones COVID-19 14
Tabla 3. Resultados predicciones intervalares de la demanda eléctrica 46

ÍNDICE DE FIGURAS

Figura 2-1 Demanda Eléctrica estimada para $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0$ 9

Figura 2-2. Demanda Eléctrica estimada para $\gamma = 1,\ \alpha_h = 0.71,\ \alpha_\gamma = 0$ 10

Figura 2-3 Demanda Eléctrica estimada para $\gamma = 1,\ \alpha_h = 4.6,\ \alpha_\gamma = 5$ 10

Figura 2-4. Predicción de hospitalizados con $\gamma = 5,\ \alpha_h = 0,\ \alpha_\gamma = 0$ con BDF 20-80 11

Figura 2-5. Predicción de hospitalizados con $\gamma = 5,\ \alpha_h = 0,\ \alpha_\gamma = 0$ con BDF 70%-30% 12

Figura 2-6. Predicción de hospitalizados con $\gamma = 5,\ \alpha_h = 1,\ \alpha_\gamma = 1$ con BDF 70%-30% 12

Figura 2-7 Predicción de hospitalizados con $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0$ con BDD 25%-75% 13

Figura 2-8. Predicción de hospitalizados con $\gamma = 10,\ \alpha_h = 3,\ \alpha_\gamma = 0.5$ con BDD 25%-75% 13

Figura 3-1. Aproximación de $\varphi(\beta)$ a J^* 16

Figura 3-2. Caso $\lambda_i > 0$ 18

Figura 3-3. Caso $\lambda_i = 0$ 18

Figura 3-4. Caso $\lambda_i < 0$ 18

Figura 3-5. Aproximación de ISTA 20

Figura 3-6. Punto extrapolado 20

Figura 3-7. Convergencia ISTA 23

Figura 3-8. Convergencia FISTA 23

Figura 3-9. Convergencia ISTA - FISTA 24

Figura 3-10. Tiempo de cómputo frente a iteraciones de ISTA y FISTA 24

Figura 3-11. Mejora del coste computacional 25

Figura 3-12. Evolución del coeficiente $\frac{t_{k-1}-1}{t_k}$ frente a iteraciones 25

Figura 3-13. Variación del momentum en torno al pseudo-óptimo 26

Figura 3-14. Convergencia FISTA – FISTA restart 28

Figura 3-15. Tiempo de cómputo por iteración. FISTA - FISTA restart 28

Figura 4-1. Función cuadrática por estimar 32
Figura 4-2. Función de densidad empírica condicionada 33
Figura 4-3. Predicción intervalar de x^2 para $x = 6$ 33
Figura 4-4. Predicción intervalar de x^2 para $x = 6$ con c aumentada 34
Figura 4-5. Evolución de la variable dual 36
Figura 4-6. Iteraciones FISTA para cada una de las soluciones de y_l 37
Figura 4-7. Predicción Intervalar 37
Figura 4-8. FISTA para distintas condiciones iniciales. 37
Figura 4-9. Predicción intervalar con c exagerada 38
Figura 4-10. Predicción intervalar con c muy disminuida 38
Figura 4-11. Predicción intervalar con c plausible 39
Figura 4-12. Predicción intervalar para $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0\ , c = 1.797$ 42
Figura 4-13. Predicción intervalar para $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0,\ c = 1.797$ 42
Figura 4-14. Predicción intervalar para $\gamma = 3,\ \alpha_h = 3,\ \alpha_\gamma = 3,\ c = 0.1875$ 44
Figura 4-15. Predicción intervalar para $\gamma = 1,\ \alpha_h = 1.4,\ \alpha_\gamma = 1.4,\ c = 0.875$ 44
Figura 4-16. Predicción intervalar para $\gamma = 0.2,\ \alpha_h = 0.5,\ \alpha_\gamma = 0.23,\ c = 4.76$ 45
Figura 4-17. Predicción intervalar para $\gamma = 0.2,\ \alpha_h = 0.5,\ \alpha_\gamma = 0.23,\ c = 4.76$ 45

1 INTRODUCCIÓN

Busca la sencillez y la eficacia

Úrsula Morón Pachón

Desde el inicio de su desarrollo, la predicción y la ciencia de datos en general han sido herramientas fundamentales en cualquier ciencia y/o aplicación tecnológica como la meteorología, el control automático, ciencias biológicas… Es fácil comprender que obtener una estimación fiable de un dato desconocido es una ventaja. Estas herramientas han provisto, actualmente, gracias a los recursos de los que se disponen hoy en día, de avances esenciales en dichas materias, formando intrínsecamente parte de la búsqueda de soluciones de una gran generalidad de problemas. Además, gracias al aumento de la capacidad computacional, han permitido el tratamiento y uso de una ingente cantidad de datos para la obtención de predicciones y resolución de problemas de una manera eficaz. En general, la predicción consiste en, partiendo de unos datos pasados o de entrada, la obtención de unos datos futuros o de salida, esperables.

En este trabajo se pretende abordar las "predicciones intervalares" que consisten en determinar el intervalo de confianza más pequeño dentro del cual, para una probabilidad preestablecida, están contenidas las salidas futuras. Es decir, dada una entrada x_k, se tratará de obtener el intervalo de menor tamaño que contiene a la salida y_k con una probabilidad especificada. La consecución de este objetivo se aborda con un problema de optimización convexa definido por una familia de funciones de disimilitud propuesta por A. D. Carnerero, D. R. Ramírez y Teodoro Álamo en [1].

Para finalizar, se han desarrollado una serie de librerías de funciones en lenguaje Python con las que se obtienen predicciones intervalares y predicciones basadas en la regresión con funciones de disimilitud. Estas librerías proveen de una gran cantidad de opciones que facilitan su manejabilidad sin mermar su robustez.

1.1 Motivación

Las predicciones intervalares juegan un papel muy importante en el control de sistemas inciertos. En otros ámbitos como *Zonotopes* y *DC Programming* se obtienen intervalos de estimadores de estado ([2] y [3] respectivamente). Observadores intervalares para sistemas lineales variantes en el tiempo son propuestos en [4] y [5]. La caracterización estadística del ruido y las perturbaciones son usadas para mejorar el desempeño del método de la estimación intervalar en [6] y [7].

Como ya se ha mencionado, para la obtención de estas predicciones intervalares hará falta resolver un problema de optimización convexa. Tradicionalmente, métodos basados en *gradient descent* han sido muy utilizados para minimizar problemas con una dimensión elevada de variables, incluso siendo muy densos matricialmente. No obstante, es conocida la lentitud de estos algoritmos a la hora de converger [8]. Por esta razón, en este Trabajo de Fin de Grado (en adelante TFG) se plantea la resolución de los problemas de optimización convexa usando el método FISTA [8] (el cual es una mejora de su predecesor ISTA). También se incorpora una modificación al algoritmo original descrita en [9] que permite solventar un problema práctico de este método.

1.2 Objetivos

El objetivo principal de este TFG es obtener predicciones intervalares minimizando la amplitud de un intervalo que contenga una salida y_k, asociada al regresor x_k, para una probabilidad especificada.

El segundo objetivo, derivado del anterior, sería conseguir estas predicciones intervalares de la forma más fiable y rápida posible. Para ello, puesto que las predicciones intervalares de este TFG se basan en la resolución de un problema de optimización convexa, se ha escogido un método con una ratio global de convergencia superior a los métodos de *gradient descent* tradicionales.

Para implementar los dos objetivos anteriores se desarrollará una serie de librerías en Python que computen los algoritmos necesarios. Consecuentemente con el segundo objetivo, la piedra angular del código será la reducción del coste computacional sin merma de su eficiencia y fiabilidad. A su vez, este código ofrecerá opciones que facilitan su uso sin menoscabo de su claridad y robustez.

1.3 Organización de la memoria

En primer lugar, en el capítulo **2 Cálculo de la Similitud y Disimilitud**, se explican las funciones de similitud y disimilitud, así como su utilidad. Una vez comprendidas y definidas formalmente, se define una familia de funciones de disimilitud usadas para las predicciones intervalares y la posibilidad de introducir información local en los algoritmos que se usarán más adelante. Finalmente, se muestra cómo aplicar esta familia de funciones en el ámbito de la regresión para obtener predicciones.

En el capítulo **3 Optimización**, se resuelve el problema de optimización convexa que permite obtener una medida de disimilitud entre un punto y un set de datos. Se presenta el problema a minimizar asociado a la familia de funciones propuesta en el capítulo anterior. Igualmente, se desarrollan las características y el planteamiento utilizado para su determinación. Se introducen los algoritmos gradenciales tradicionales en los que está basado el método FISTA utilizado en este TFG. Además, se presentará la implementación en Python del mismo. Para finalizar, se mostrará como el método FISTA supera con creces la ratio de convergencia de su predecesor ISTA y se explicarán una serie de inconvenientes y como solucionarlos. Todo el capítulo contendrá secciones de cara a mejorar la eficiencia del código implementado.

En el siguiente capítulo **4 Predicciones Intervalares**, convergen los dos capítulos anteriores permitiendo definir la metodología para la obtención de las predicciones intervalares. Las funciones de disimilitud se usarán para calcular la función de densidad empírica de la salida y_k condicionada a x_k. Partiendo de esta función de densidad empírica condicionada, se introduce el Algoritmo 1 con el que se llega a una predicción intervalar en función de dos escalares, γ y c . Finalmente, para ser consecuente con el objetivo principal del trabajo, el Algoritmo 2 es utilizado para optimizar el escalar c de modo y forma que los intervalos calculados sean lo más reducidos posible y contengan a la salida y_k para una probabilidad especificada previamente. Como en el capítulo anterior, todos los algoritmos tienen secciones específicas para optimizar la eficiencia y

reducir el coste computacional.

En el último capítulo **5 Conclusiones**, se resumen todos los resultados alcanzados para las metodologías propuestas exponiendo las ventajas de las mismas, así como las dificultades encontradas.

Para terminar, en el punto **6 Librería de Funciones Python** se expone el código en Python y se explica el uso de las librerías de funciones desarrolladas íntegramente por el autor. Se trata de proveer de un manual de instrucciones para que cualquier interesado pueda hacer uso de ellas.

2 CÁLCULO DE LA SIMILITUD Y DISIMILITUD

Ser derrotada es, a menudo, una condición temporal.

Rendirse es lo que lo hace permanente.

Marilyn vos Savant

Durante todo el TFG, las funciones de similitud y disimilitud tendrán un papel fundamental. Estas son la base para sustentar todo el desarrollo de las predicciones intervalares y regresión con funciones de disimilitud. Por ello, en este capítulo, se definirá y se expondrá la familia de funciones de disimilitud que será usada. Una vez conocida esta familia de funciones, se mostrará como pueden usarse para la regresión asemejándose a otros métodos de combinaciones lineales de salidas observadas [6].

2.1 Introducción a las funciones de disimilitud

La idea de las funciones de similitud y disimilitud, como su nombre indica, es la de medir el grado de similaridad o disimilaridad entre escalares, vectores, y sets de datos. En este TFG se usarán para medir la disimilitud entre un vector z dado con un set de datos D. En concreto, las funciones de disimilitud devuelven grandes valores cuando z y el vector del set de datos D difieren en gran medida. En contraposición, devolverán pequeños valores cuando haya un alto grado de similaridad. Un ejemplo simple de función de disimilitud podría ser la norma euclídea.

La función de disimilitud será definida de ahora en adelante como:

$$J_d(\cdot,\cdot):\ \mathbb{R}^{\mathrm{n}}\ x\ D \rightarrow [0, \infty],$$

donde D será el set de datos tal que: $D = \{z_i : i = 1, \dots, N\} \subset \mathbb{R}^N$.

Es muy importante señalar que partiendo de una función de disimilitud se puede obtener una función de similitud $J_s(x, D)$. En concreto, se usará la siguiente relación. Dado $c > 0$:

$$J_s(x, D) = e^{-c \cdot J_d(z,D)}. \qquad (2\text{-}1)$$

Un ejemplo de uso de funciones de similitud podría ser el *clustering*, pues permiten agrupar conjuntos de datos según su similitud o disimilitud en el contexto del aprendizaje no supervisado.

En este trabajo se usarán en el contexto de la regresión y para el cálculo de funciones de densidad empíricas condicionadas. Es decir, partiendo de los pares $\{z_i, y_i\}$ y teniendo en cuenta la no similitud entre un punto z_i y su conjunto D se tratará de estimar y_i como se explica en **2.3 Regresión con** $\mathrm{Jy}\ (\mathrm{z}, \mathrm{D})$**.**

2.2 Familia de funciones de disimilitud propuesta

La familia de funciones que será usada a lo largo de todo el TFG fue propuesta en [1] y proporciona una serie de características mucho más interesantes que la simple distancia mínima a un conjunto:

$$J_d(z, D) = \min_{\hat{z} \in D} ||z - \hat{z}||_1.$$

Además, la familia propuesta tiene una serie de propiedades demostradas en [1] (propiedad 1) como la invarianza respecto a transformaciones afines. Esta propiedad de invarianza implica que el análisis que se realice basándose en esta familia de funciones no se verá afectado por la elección particular de un sistema de coordenadas. De esta forma, ni el punto z ni el conjunto D deberán tener un tratamiento de datos previos de normalización y aunque lo tuviera, el resultado del análisis sería exactamente el mismo. Esta propiedad no es común en funciones de disimilitud. Como ejemplo, una función de disimilitud sobre la distancia mínima entre dos puntos depende directamente del sistema de coordenadas elegido.

2.2.1 Definición formal

En primera instancia, se definirá formalmente la función de disimilitud y después se explicará una variante que será usada en este trabajo.

Dado el conjunto de datos $D = \{z_i : i = 1, \dots, N\} \subset \mathbb{R}^N$ y el escalar $\gamma \geq 0$ la función de disimilitud $J_\gamma: \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}$ está definida como:

$$\begin{aligned} J_\gamma(z, D) &= \min_{\lambda_1,\dots,\lambda_N} \sum_{i=1}^{N} \lambda_i^2 + \gamma \sum_{i=1}^{N} |\lambda_i| \\ s.t. \qquad z &= \sum_{i=1}^{N} \lambda_i z_i \\ 1 &= \sum_{i=1}^{N} \lambda_i. \end{aligned} \qquad (2\text{-}2)$$

Como se puede observar, el resultado de la función $J_\gamma(z, D)$ necesita de la resolución de un problema de optimización convexa de una función de coste con dos restricciones de igualdad. Para resolver este problema es necesario encontrar $\lambda = \{\lambda_i : i = 1, \dots, N\}$ que minimice la función de coste cumpliendo las restricciones de igualdad. Se busca por tanto λ^*. El problema de optimización será tratado en detalle en el capítulo **3 Optimización**. Si se considera al parámetro γ un parámetro a sintonizar, se obtiene una familia de funciones de disimilitud.

2.2.2 Ponderaciones en la función de coste

Una vez expuesta la función de coste que se quiere minimizar, se propone una variante a la que se han añadido pesos que ponderan la disimilitud de z con los datos de D. Especialmente útil para sistemas no lineales.

$$J_\gamma(z, D) = \min_{\lambda_1,\dots,\lambda_N} \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i| . \qquad (2\text{-}3)$$

El escalar $\gamma \geq 0$, que se encontraba en la definición formal, se sustituye por un vector ponderado multiplicado por γ pues se debe permitir la elección de pesos tanto en el término cuadrático como en el término del valor absoluto. Así pues, se define la matriz diagonal definida positiva:

$$\Gamma_w \in \mathbb{R}^{\mathrm{NxN}}, \qquad \Gamma_w = \gamma \cdot diag\left(w_1^\gamma, \dots, w_N^\gamma\right), \qquad w_i^\gamma \geq 0 \; \forall \left\{w_1^\gamma : i = 1, \dots, N\right\},$$

junto a otra matriz diagonal definida positiva para el término cuadrático:

$$H_w \in \mathbb{R}^{\mathrm{NxN}}, \qquad H_w = diag\left(w_1^h, \dots, w_N^h\right), \qquad w_i^h \geq 0 \; \forall \left\{w_i^h : i = 1, \dots, N\right\}.$$

2.2.3 Función de disimilitud para matriz de pesos

A la hora de introducir información local en la función de coste para ponderar de distinta manera los datos de D se pueden proponer distintas soluciones. Una posibilidad es ponderar con mayor importancia un dato reciente que uno menos actual, principalmente para series temporales. Otra será usar una función de disimilitud entre el dato i-ésimo de D con el punto z que se está evaluando. En concreto, se tomará esta última opción incluyendo la siguiente función de disimilitud para cada elemento de la diagonal de H_w y Γ_w :

$$w_i = J_d(z, z_i) = e^{\alpha \cdot \|z - z_i\|_2^2}, \quad (2\text{-}4)$$

definiéndose α_h y α_γ como los parámetros mayores o iguales a cero necesarios para controlar la información local añadida al funcional. En este caso, la función de disimilitud no es invariante respecto a cambios en los ejes de coordenadas y es sensible a la posible no normalización de los datos de D. En esencia, si un vector z_i del set de datos D se aleja de z, la disimilitud w_i será grande. Esto provocará que la λ_i^* asociada a z_i deba disminuir considerablemente su valor para paliar el mayor valor de w_i. Por consiguiente, la salida asociada a z_i será tenida mucho menos en cuenta en la combinación lineal de salidas para la solución de $\hat{y}$. Esto se verá más a fondo en la siguiente sección.

2.3 Regresión con $J_\gamma(z, D)$

Sea D un conjunto de entradas y salidas de un sistema tal que: $D = \left\{ z_i = \begin{bmatrix} y_i \\ x_i \end{bmatrix} : i = 1, \dots, N \right\} \subset \Upsilon \times \mathrm{X}$ con $[y_i, x_i]^T$ los pares de datos de salida y entrada respectivamente. Dado un punto de entrada x_k se puede obtener una estimación de $\hat{y}_k$ de y_k minimizando la función de disimilitud $J_\gamma\left(\begin{bmatrix} y_k \\ x_k \end{bmatrix}, D\right)$. Es decir:

$$\hat{y}_k = \arg\min_y J_\gamma\left(\begin{bmatrix} y \\ x_k \end{bmatrix}, D\right), \quad (2\text{-}5)$$

por lo que, teniendo el set de datos D, la entrada x_k, el parámetro γ y las matrices de peso H_w y Γ_w, la estimación de $\hat{y}_k$ se obtiene de la minimización de:

$$\begin{aligned} \min_{y, \lambda_1, \dots, \lambda_N} \; & \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i| \\ s.t. \quad & \begin{bmatrix} y_k \\ x_k \end{bmatrix} = \sum_{i=1}^{N} \lambda_i \begin{bmatrix} y_i \\ x_i \end{bmatrix} \\ & 1 = \sum_{i=1}^{N} \lambda_i. \end{aligned} \quad (2\text{-}6)$$

De la restricción de igualdad (2-6) se puede aislar la variable de decisión buscada y_k puesto que no aparece ni en la función de coste ni en la restricción de igualdad de la normalización a la unidad de los escalares λ_i:

$$y_k = \sum_{i=1}^{N} \lambda_i y_i.$$

Es decir, el problema de minimización puede reducirse a:

$$\min_{\lambda_1,\dots,\lambda_N} \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i|$$
$$s.t. \qquad x_k = \sum_{i=1}^{N} \lambda_i x_i \tag{2-7}$$
$$1 = \sum_{i=1}^{N} \lambda_i.$$

Como se puede observar, el problema de optimización ignora la restricción de igualdad en la que se encontraba y_k de forma que cuando se obtengan los escalares λ_i^* la estimación óptima de y_k será:

$$\hat{y}_k = y_k^* = \sum_{i=1}^{N} \lambda_i^* y_i.$$

Por tanto, la estimación de y_k será una combinación lineal de las salidas anteriores observadas $y_i\,:\,i = 1,\dots N$ y con los escalares de la solución óptima del problema $\lambda_i^*\,:\,i = 1,\dots,N$. Esto coincide con los resultados obtenidos en [6] donde la solución óptima de un problema de estimación, para unos supuestos, está dada únicamente por combinaciones lineales de las salidas observadas.

El problema de optimización y su implementación en Python se verán en detalle en el siguiente capítulo **3 Optimización**.

2.3.1 Demanda eléctrica española

En esta subsección se aplica lo explicado sobre regresión con funciones de disimilitud a la predicción de la demanda eléctrica española a siete días vista (aunque es fácilmente generalizable como se ve en el siguiente apartado). Los datos se refieren a los años 2014 – finales de 2019 y se han descargado de [10]. En total se ha tomado la demanda de 2139 días.

A la hora de predecir series temporales es necesario plantear una base de datos dinámica que olvide datos pasados. También, debido a la gran cantidad de días es necesario que el set de datos D se nutra de una base de datos dinámica que se definirá como $BD = x_i : i = 1,\dots,dbd$ (con dbd: días base de datos). Es decir, para cada predicción se debe olvidar el dato de demanda x_1, que al ser antiguo no aporta utilidad real, e incluir el más actual a x_{dbd} después de trasladar todos los elementos a una posición anterior. Además, sería muy costoso computar, para cada predicción, toda la base de datos acumulada. Por consiguiente, el último elemento de BD (el más actual) sería el día desde el que se quiere predecir (suponiendo que el dato de demanda ya esté disponible). El día siguiente, se incluirá como el dato más actual y se trasladarán todos los elementos de BD a una posición anterior (olvidando el primer elemento, que es el más antiguo).

Dado $D = \left\{ \begin{bmatrix} X_i \\ y_i \end{bmatrix} : i = 8,\dots,dbd - 7 \right\}$ y la base de datos: se ha montado el regresor $X_i \in \mathbb{R}^4$, referido a la demanda correspondiente a siete días más tarde y_i:

- El primer componente de X_i será la demanda del día i-ésimo de la base de datos: x_i.
- El segundo componente de X_i será la demanda del día anterior al día i-ésimo de la base de datos: x_{i-1}.
- El tercer componente de X_i será la demanda de hace siete días desde el día i-ésimo de la base de datos: x_{i-7}.
- El cuarto componente será el día de la semana de x_i codificado como 1, 2…7 para lunes, martes… domingo respectivamente definiéndose $ds_i\,:\,i = 1,\dots,7$.

De esta forma, la demanda y_i será en todo momento la demanda a siete días vista desde x_i.Es decir, se cumple: $\forall y_i \mid y_i = x_{i+7}$.

Por lo que la dimensión de D será: $D \in \mathbb{R}^{5\text{x}(\text{dbd}-14)}$. Es importante señalar que en la medida en la que la base de datos BD se actualice así lo hará D.

En definitiva, para $BD = x_i : i = 1, \dots, dbd$:

$$D = \left\{ \begin{bmatrix} x_i \\ x_{i-1} \\ x_{i-7} \\ ds_i \\ x_{i+7} \end{bmatrix} : i = 8, \dots, dbd - 7 \right\}. \qquad (2\text{-}8)$$

Como ya se ha mencionado, el último dato de la base de datos será el más actual, es decir, la demanda del día desde el cual se quiere predecir. Por lo que, si hoy se quiere predecir la demanda a siete días vista, hoy sería el día dbd-ésimo de la base de datos.

No se tendrán en cuenta las estaciones del año, ni la meteorología (si hace mucho frío o calor la demanda sube) ni fiestas más allá de los fines de semana. En resumen, el regresor del día desde donde se quiere predecir será:

$$X = \left\{ \begin{bmatrix} x_{dbd} \\ x_{dbd-1} \\ x_{dbd-7} \\ ds_{dbd} \end{bmatrix} \right\}.$$

2.3.1.1 Predicción sin información local en la función de coste

Para empezar, se usará la familia de funciones de disimilitud sin ponderaciones en el término cuadrático y valor absoluto de la función de coste. La base de datos usada es de ciento veinte días. A la izquierda de la gráfica superior se puede apreciar el tamaño de la misma respeto al resto de días:

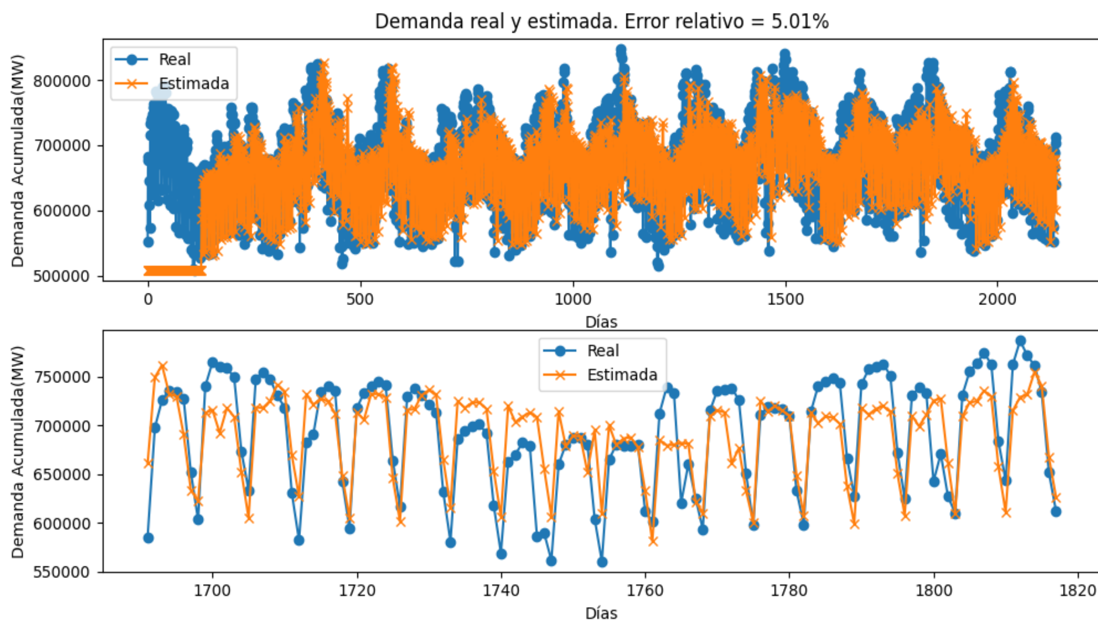

Figura 2-1 Demanda Eléctrica estimada para $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0$

Se consigue una predicción estable e invariante respecto a picos inesperados.

2.3.1.2 Predicción con información local en la función de coste

En este caso, se sintonizan los tres parámetros fundamentales de ponderación: γ, α_γ y α_h. Para ello, habrá que tener en cuenta las prioridades de la predicción. Mientras que valores como $\gamma = 0,\ \alpha_\gamma = 0,\ \alpha_h = 0.71$ minimizan el valor relativo medio de la predicción, no consiguen amoldarse del todo a la curva real ni a cambios más bruscos. Valores que dan más importancia a días pasados similares, como $\gamma = 1, \alpha_\gamma = 5, \alpha_h = 5$, siguen mejor la curva real pero cometen más fallos cuando hay un pico completamente inesperado.

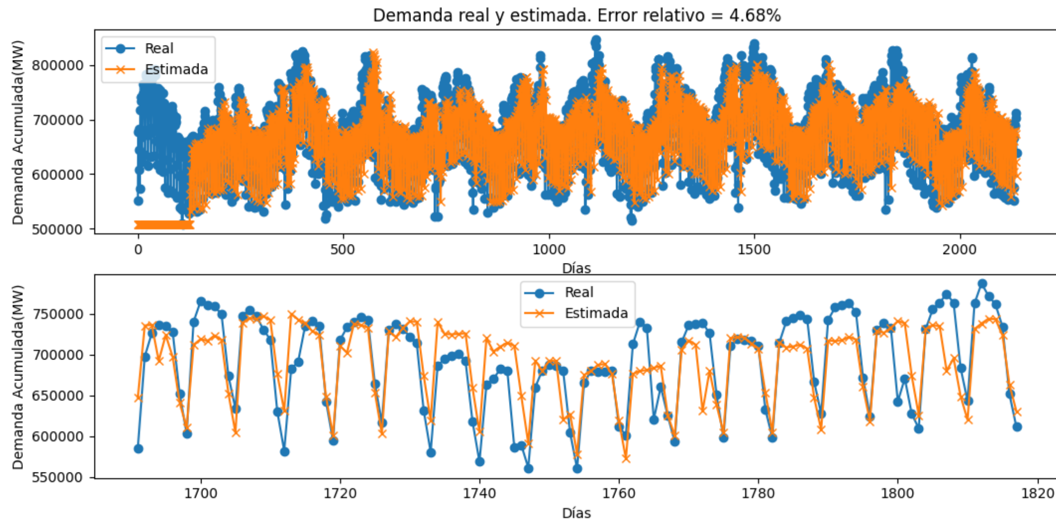

Figura 2-2. Demanda Eléctrica estimada para $\gamma = 0,\ \alpha_h = 0.71,\ \alpha_\gamma = 0$

Al incluir información local única y exclusivamente en el término cuadrático de la función de coste y reducir el método a mínimos cuadrados ($\gamma = 0$), se consigue un buen error relativo general pero la predicción parece un reflejo de la semana anterior en la mayoría de los casos. Es decir, esta predicción se basa mucho en lo que ocurrió la semana anterior.

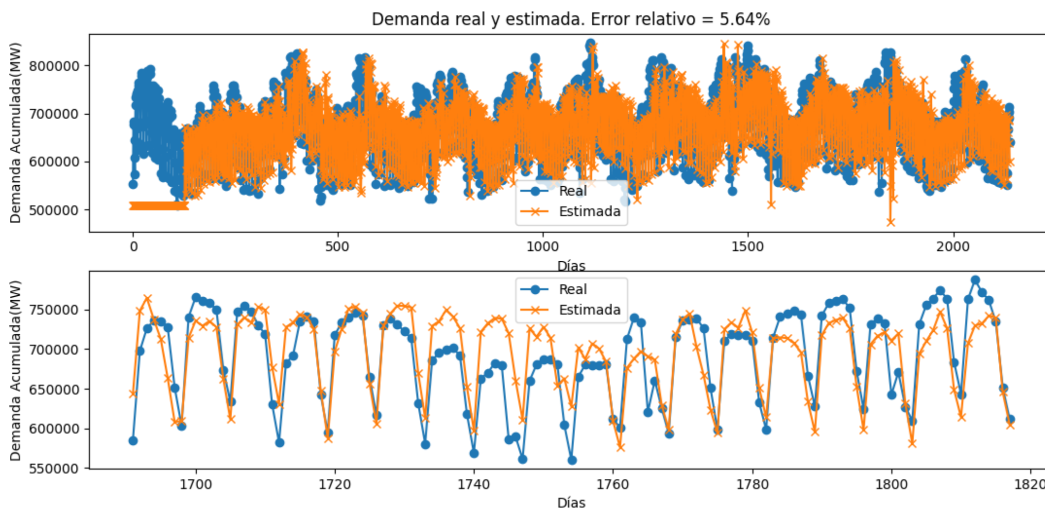

Figura 2-3 Demanda Eléctrica estimada para $\gamma = 1,\ \alpha_h = 4.6,\ \alpha_\gamma = 5$

Como ya se ha comentado, en este caso la predicción tiene la capacidad de asemejarse a la curva real. El mejor ejemplo sucede en la penúltima semana entre los días 1803 y 1810. En la Figura 2-2 la predicción de esa semana es un reflejo de la anterior. En cambio en la Figura 2-3 queda plasmado que la curva de la predicción se parece mucho más a la real. No obstante, cuando la curva real es inesperada el error cometido es superior. Se podría decir que este segundo caso es más agresivo. Estos picos podrían ser tratados a fondo con un mejor estudio de eventos como días festivos, temperatura, estaciones, etc.

2.3.1.3 Tabla de resultados

En la siguiente tabla se muestran los resultados más significativos de las 3 predicciones anteriores. En ella se puede observar la relación entre los parámetros: $\gamma, \alpha_\gamma, \alpha_h$, y el comportamiento de la predicción. Los resultados que se han considerado son:

- ERM: el error relativo medio.
- EAM: el error absoluto medio.
- σ: desviación estándar.

Tabla 1. Resultados de predicciones de la demanda eléctrica

γ	α_h	α_γ	ERM	EAM	σ
1	0	0	5.0%	34138	46592
0	0.71	0	4.99%	34069	46504
1	4.6	5	5.54	38054	52843

2.3.2 Predicción de la evolución del COVID-19

En esta subsección se aplica lo explicado sobre regresión con funciones de disimilitud en un tipo de caso distinto al de la demanda eléctrica: se predice la evolución de hospitalizados por SARS-CoV-2 en España para lo que se han descargado los datos de [11].

Como se mencionó anteriormente, a la hora de predecir series temporales es importante plantear, la mayoría de las veces, una base de datos dinámica. Esto es especialmente relevante para la regresión con funciones de disimilitud ya que esta no funciona adecuadamente cuando los datos a predecir se alejan mucho del dominio de D. Puesto que el número de hospitalizados por COVID-19 tiene una evolución oscilante, el set de datos D queda rápidamente desfasado si no se actualiza. No obstante, como se ejemplifica en los siguientes apartados, para este método de predicción y para este ejemplo, es preferible un set de datos D grande con numerosas muestras a una base de datos dinámica.

Se han tomado 267 días desde el 2020-04-12 hasta el 2021-01-4 y se quiere predecir para diez días vista aunque es fácilmente generalizable definiendo dv: días vista. En primer lugar, se tratará de predecir con una base de datos fija del 20% de los días totales: $BD = x_i : i = 1, \dots, dbd$ (con dbd: días base de datos) donde x_i contiene el número de hospitalizados para el día i-ésimo. Es un buen ejemplo de la debilidad de la regresión planteada.

El set D introducido se define como $D = \left\{ \begin{bmatrix} y_i \\ x_i \end{bmatrix} : i = 1 + dv, \dots, dbs - dv \right\}$ donde y_i será el número de hospitalizados diez días más tarde del número de hospitalizados correspondientes a x_i. Es decir, se cumple:

$$\forall y_i \mid y_i = x_{i+dv}.$$

2.3.2.1 Predicción con base de datos fija

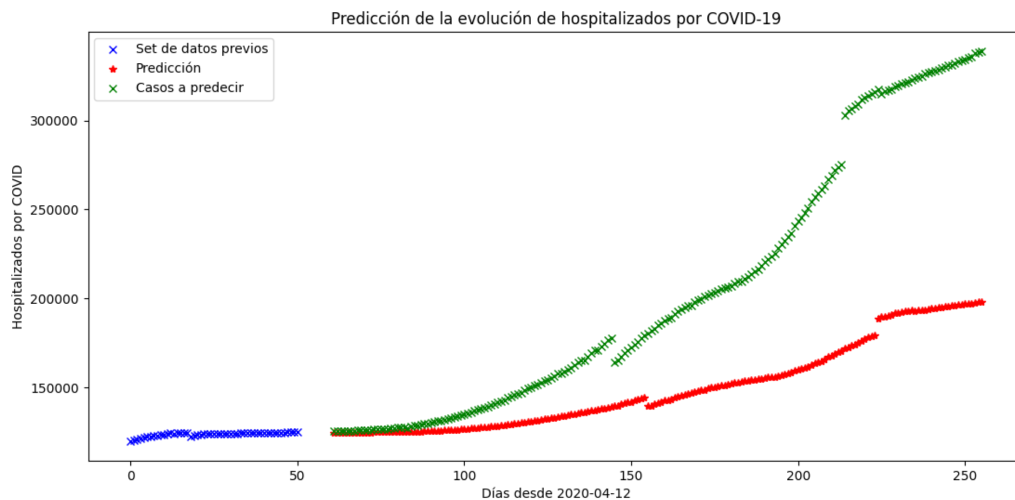

Figura 2-4. Predicción de hospitalizados con $\gamma = 5$, $\alpha_h = 0$, $\alpha_\gamma = 0$ con BDF 20-80

Al dejar fija BD y tener tan pocas muestras, la predicción no consigue seguir la naturaleza ascendente de la evolución de los hospitalizados. El error relativo es de 26.56%. No importa si se trata de mejorar la predicción con pesos $\alpha_h > 0$ $\alpha_\gamma > 0$ pues el problema real reside en que al ser una base de datos fija hacen falta muchísimas más muestras. Se prueba a continuación con una división entre los días de BD y el set a predecir de 70% y 30% respectivamente.

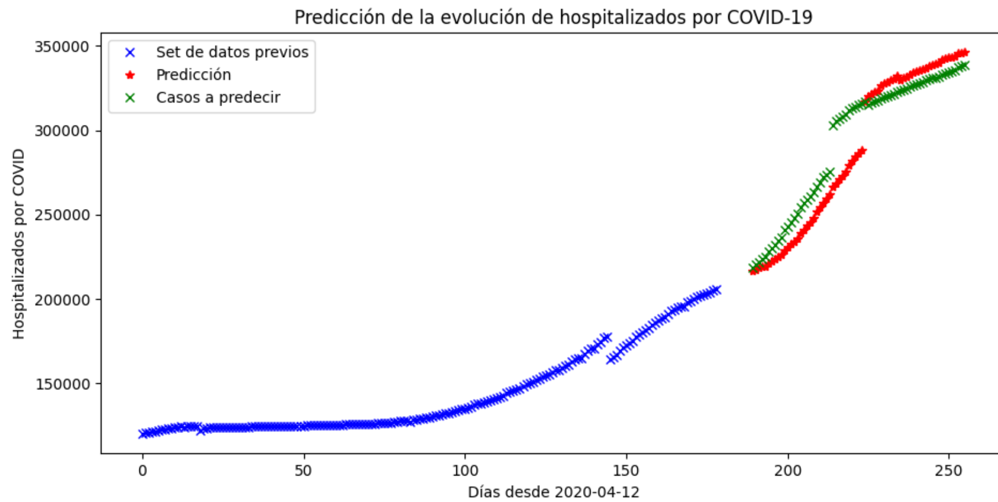

Figura 2-5. Predicción de hospitalizados con $\gamma = 5$, $\alpha_h = 0$, $\alpha_\gamma = 0$ con BDF 70%-30%

En este caso se obtiene una clara mejoría pagando el precio de la necesidad de una base de datos grande. Se prueba a añadir información local en la función de coste consiguiendo una ligera mejoría:

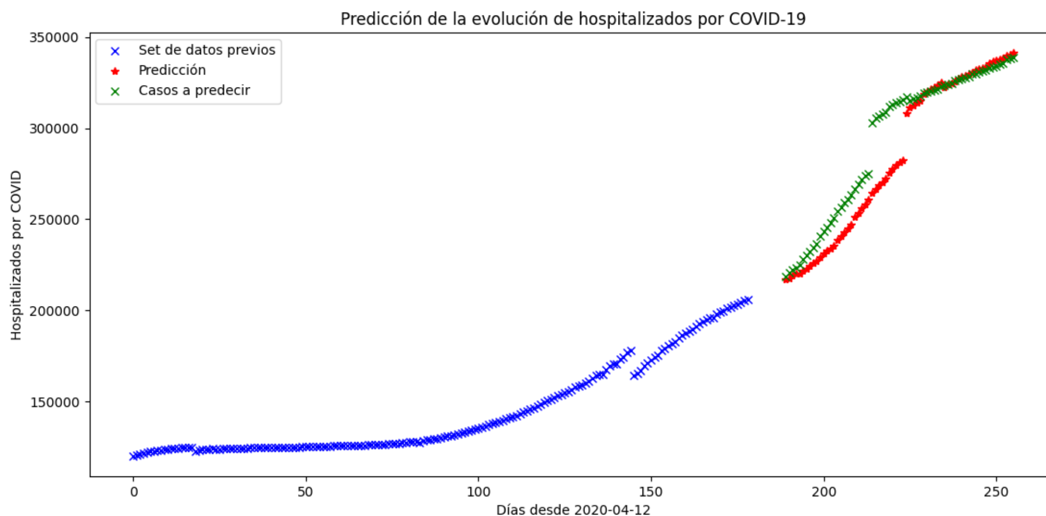

Figura 2-6. Predicción de hospitalizados con $\gamma = 5$, $\alpha_h = 1$, $\alpha_\gamma = 1$ con BDF 70%-30%

2.3.2.2 Predicción con base de datos dinámica

Para montar la base de datos, se ha reducido el número de días de la misma al 25% del total (67 días). Se sigue la misma línea que en la subsección de la demanda eléctrica. Tras cada predicción se olvidará el dato del primer día de BD: x_1 (el más antiguo) y se trasladarán todos los elementos a una posición anterior añadiendo uno nuevo (el más actual) a x_{dbd}.

Al igual que en el apartado anterior, se representará un ejemplo sin pesos y se comparará con otro ejemplo con pesos afinados. El tramo azul representa la longitud del set de datos para la primera predicción

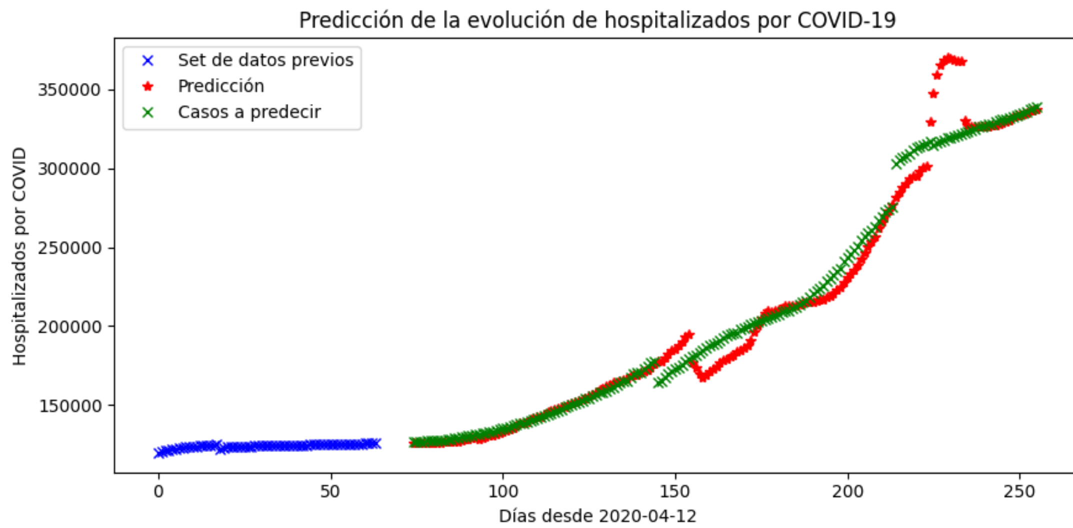

Figura 2-7 Predicción de hospitalizados con $\gamma = 1$, $\alpha_h = 0$, $\alpha_\gamma = 0$ *con BDD 25%-75%*

En primera parte de la predicción, entre los días 70 y 140, la evolución de los hospitalizados no se aleja en gran medida del set de datos a diez días vista. No obstante, comienza a predecir peor en cuanto en la base de datos se incluye el salto discontinuo descendiente pasados 150 días. Es lógico que la predicción baje también inmediatamente después de este descenso a los dv días más tarde.

Debido a que a los 225 días se da un salto ascendente inesperado, la predicción falla completamente. En cuanto los valores de este salto se incluyen en la base de datos, la predicción da un salto también. Una vez se introducen más valores, en torno a los 245 días, se normaliza y se estabiliza.

Se trata de mejorar la predicción con los pesos sin cambios aparentes en los resultados más allá de un decremento en el error relativo de 0.2%.

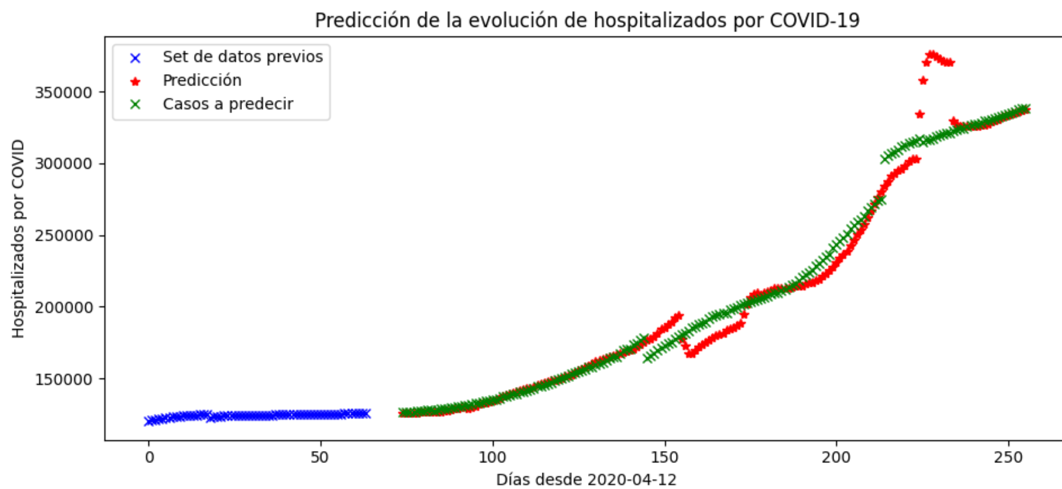

Figura 2-8. Predicción de hospitalizados con $\gamma = 10$, $\alpha_h = 3$, $\alpha_\gamma = 0.5$ *con BDD 25%-75%*

En la primera subsección, la idea de una base de datos dinámica era útil pues BD contenía 120 días de demanda eléctrica por lo que en si misma eran suficientes muestras. No obstante, los 67 días utilizados en el caso de este apartado no parecen suficientes, lo que concuerda con la debilidad de esta metodología de predicción.

Además, el regresor de la demanda eléctrica era superior al utilizado para el COVID-19 pues era relativamente sencillo de montar. Como futura línea de investigación, sería conveniente buscar otros indicadores u otra forma de afrontar el problema como dividir la predicción por provincias usando información local sobre cada una de ellas (densidad de población, colapso de los hospitales etc.).

2.3.2.3 Tabla de resultados

En la siguiente tabla se muestran los resultados más significativos de las cinco predicciones anteriores. En ella se puede observar la relación entre los parámetros: $\gamma, \alpha_\gamma, \alpha_h$, y el comportamiento de la predicción. Los resultados que se han considerado son:

- ERM: el error relativo medio.
- EAM: el error absoluto medio.
- σ: desviación estándar.

Tabla 2. Resultados predicciones COVID-19

$\boldsymbol{\gamma}$	$\boldsymbol{\alpha_h}$	$\boldsymbol{\alpha_\gamma}$	$\boldsymbol{BD}$	ERM	EAM	σ
5	0	0	BDF 20-80	21.15%	52803	48337
5	0	0	BDF 70-30	3.81%	11060	13742
5	1	1	BDF 70-30	3.31%	9376	12642
1	0	0	BDD 25-75	3.43%	8306	15336
10	3	0.5	BDD 25-75	3.44%	8231	14935

3 OPTIMIZACIÓN

No corras más que tu cerebro

José Francisco Torres González

Este TFG está fuertemente ligado a la resolución de un problema estrictamente convexo de optimización. Esto implica que la solución es única. En concreto, la minimización del funcional $J_\gamma(z, D)$ descrito en **2.3 Regresión con** $\mathrm{J\gamma\ (z, D)}$. Esta minimización proporciona los valores óptimos λ que son la solución del problema de optimización. Como se verá, para atacar al problema, se debe dar un enfoque dual. Esto es así puesto que el Primal no es diferenciable y se atiene a unas restricciones de igualdad que no permiten el cómputo directo del mismo. Además, la dimensión de la variable dual será, en la mayoría de los casos, menor que N que es el número de variables del problema primal.

En definitiva, se usará una evolución de los métodos clásicos de *gradient descent* (método FISTA) y se propondrá un algoritmo rápido y fiable. Todo lo desarrollado en este capítulo está concebido para ser implementado en un código de Python donde trabajar con el referido algoritmo descrito en **3.4.1 Implementación de FISTA en Código Python**.

3.1 Problema a minimizar

Se trabajará con el siguiente problema de minimización. Cabe señalar que, respecto al problema de optimización original, la restricción de igualdad de las salidas y_i ya ha sido desacoplada. Definiéndose:

$D = \{x_1, \dots, x_N\}$, $w_i^h \geq 0 \; \forall \left\{w_i^h : i = 1, \dots, N\right\}$, $w_i^\gamma \geq 0 \; \forall \left\{w_i^\gamma : i = 1, \dots, N\right\}$ con $J_\gamma : \mathbb{R} \times \mathbb{R}^{\mathrm{N}} \rightarrow \mathbb{R}$

$$
\begin{aligned}
J_\gamma(x, D) \quad &= \quad \min_{\lambda_1, \dots, \lambda_N} \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i| \\
s.t. \qquad & \qquad x = \sum_{i=1}^{N} \lambda_i x_i \\
& \qquad 1 = \sum_{i=1}^{N} \lambda_i.
\end{aligned}
$$

Siendo $J_\gamma(x, D)$ la función de coste que se quiere minimizar cumpliendo las dos restricciones de igualdad. Las mayores complicaciones que se presentan en este problema son la no linealidad del valor absoluto y las mencionadas restricciones de igualdad que impiden una resolución directa del problema primal.

3.2 Restricciones de igualdad y enfoque dual

Para estas dos restricciones de igualdad se aplicará un enfoque dual que relaje las mismas. Para empezar se "unirán" las dos restricciones de manera que matricialmente:

$$R\lambda = r: \quad R = \begin{bmatrix} x_1 & x_2 \dots x_N \\ 1 & 1 \dots & 1 \end{bmatrix} \in \mathbb{R}^{(n+1)xN}, \qquad \lambda = [\lambda_1 \ \lambda_2 \ \dots \ \lambda_N]^T \in \mathbb{R}^{N}, \qquad r = \begin{bmatrix} x \\ 1 \end{bmatrix} \in \mathbb{R}^{n+1},$$

donde $n+1$ es igual al número de restricciones de igualdad. El problema se plantea de la siguiente forma:

$$J_\gamma^* = \min_{\lambda \in \mathbb{R}^N} \lambda^T H_w \lambda + \|\Gamma_w \lambda\|_1$$

$$s.t. \quad R\lambda = r,$$

donde H_w y Γ_w serán matrices definidas positivas. Serán las matrices que permitan incluir ponderaciones de los datos de entrada para añadir información local tal y como se expone en el apartado **2.2.3 Función de disimilitud para matriz de pesos**.

Para trabajar las restricciones de igualdad del problema Primal se plantea un enfoque dual definiendo un funcional distinto. Este funcional dual, con variable dual $\beta \in \mathbb{R}^{n+1}$, será:

$$\varphi(\beta) = \min_{\lambda \in \mathbb{R}^N} \lambda^T H_w \lambda + \|\Gamma_w \lambda\|_1 + \beta^T (R\lambda - r).$$

La restricción de igualdad se "incluye" en el funcional dual con $\beta^T(R\lambda - r)$. Debido a este término, el mínimo obtenido será siempre menor o igual que J_γ^* siendo λ^* la solución óptima del primal que satisface $R\lambda^* - r = 0$. Por lo tanto:

$$\varphi(\beta) = \min_{\lambda \in \mathbb{R}^N} \lambda^T H_w \lambda + \|\Gamma_w \lambda\|_1 + \beta^T (R\lambda - r) \leq {\lambda^*}^T H_w \lambda^* + \|\Gamma_w \lambda^*\|_1 + \beta^T (R\lambda^* - r) = \qquad (3\text{-}1)$$

$$= {\lambda^*}^T H_w \lambda^* + \|\Gamma_w \lambda^*\|_1 = J_\gamma^*.$$

Es decir, debido al término $\beta^T(R\lambda - r)$, la solución del funcional dual será siempre menor o igual que J_γ^*:

$$J_\gamma^* \geq \varphi(\beta), \forall \lambda.$$

Esto implica que se debe encontrar el valor óptimo de la variable dual β para el cual la solución de $\varphi(\beta^*)$ es igual a J_γ^*. De esta forma, los dos problemas llegarán al mismo resultado (siempre que sean estrictamente factibles). Llegados a este punto, la solución del primal es separable en N problemas de optimización con solución expícita. Mientras que el problema de optimización Primal es convexo, el problema dual es cóncavo puesto que se debe maximizar $\varphi(\beta)$. Se modificará β hasta que se cumpla $\varphi(\beta^*) = J_\gamma^*$ tal y como se representa en la Figura 3-1.

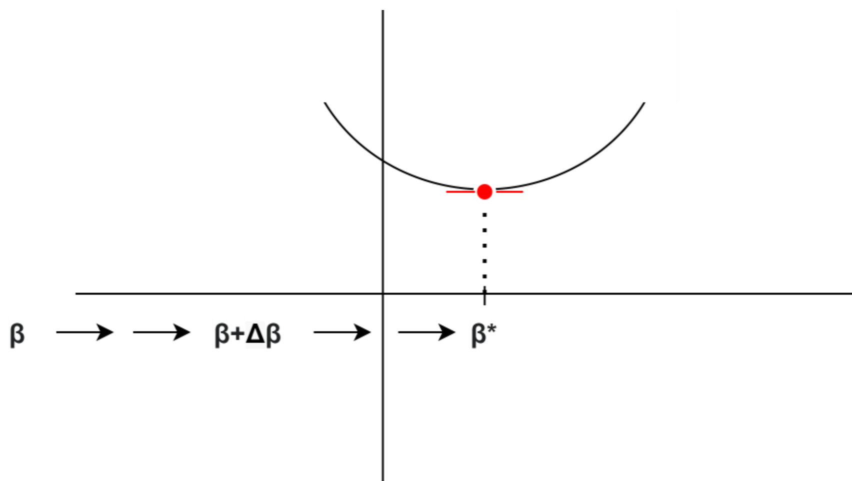

Figura 3-1. Aproximación de $\varphi(\beta)$ a J^*

Dicho esto, dada una $\beta \in \mathbb{R}^{n+1}$, no es complicado encontrar λ_β^* ya que el problema es separable como se ha mencionado. No existen restricciones de igualdad en la optimización del funcional Dual por lo que la

resolución para cada $\lambda^*_{\beta i}$ está desacoplada teniendo que resolverse N problemas escalares. Es muy importante señalar que la dimensión de la variable dual es igual al número de restricciones de igualdad $(n+1)$ lo que en la mayoría de los casos es mucho menor que el número de variables original del primal ($\lambda \in \mathbb{R}^{\mathrm{N}}$). Definiendo $c^T = \beta^T R$, el funcional dual quedaría:

$$\varphi(\beta) = \min_{\lambda_1,\dots,\lambda_N} \left(\sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i| + \sum_{i=1}^{N} c_i \lambda_i \right) - \beta^T r$$

$$\lambda^*_{\beta,i} \rightarrow \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i\,,\ \ i : 1, \dots, N \qquad (3\text{-}2)$$

Se omite el término $-\beta^T r$ puesto que no depende de λ_i. Debido al valor absoluto, que no es diferenciable, el funcional será cuadrático a tramos. Según el signo de λ_i se dividirá en:

$$\lambda^*_{\beta,i} \rightarrow \begin{cases} \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma \lambda_i + c_i \lambda_i & si \ \ \lambda_i > 0 \\ \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 - w_i^\gamma \lambda_i + c_i \lambda_i & si \ \ \lambda_i \le 0. \end{cases} \qquad (3\text{-}3)$$

Cada uno de los λ_i óptimos que minimizan el funcional se conseguirán simplemente igualando a cero la derivada respecto a λ_i del caso $\lambda_i > 0$ y $\lambda_i < 0$. Para $\lambda_i > 0$:

$$\frac{d}{d\lambda_i}\left(w_i^h \lambda_i^2 + w_i^\gamma \lambda_i + c_i \lambda_i\right) = 0 \rightarrow 2 w_i^h \lambda_i^* + w_i^\gamma + c_i = 0 \rightarrow \lambda_i^* = \frac{-(w_i^\gamma + c_i)}{2 w_i^h}.$$

Puesto que en este caso λ_i solo puede ser mayor que cero y los dos pesos w_i^h, w_i^γ provienen de matrices definidas positivas, se debe cumplir $-(w_i^\gamma + c_i) > 0 \rightarrow w_i^\gamma < -c_i$. Esto determina que para $\lambda_i > 0$:

$$\lambda_i^* = \frac{-(w_i^\gamma + c_i)}{2 w_i^h} \qquad si \quad w_i^\gamma < -c_i. \qquad (3\text{-}4)$$

Por otro lado, para $\lambda_i < 0$:

$$\frac{d}{d\lambda_i}\left(w_i^h \lambda_i^2 - w_i^\gamma \lambda_i + c_i \lambda_i\right) = 0 \rightarrow 2 w_i^h \lambda_i^* - w_i^\gamma + c_i = 0 \rightarrow \lambda_i^* = \frac{w_i^\gamma - c_i}{2 w_i^h},$$

y para que se cumpla que $\lambda_i < 0$ debe cumplirse a su vez $\gamma_i - c_i < 0$ por lo que:

$$\lambda_i^* = \frac{w_i^\gamma - c_i}{2 w_i^h} \qquad si \quad w_i^\gamma < c_i\,. \qquad (3\text{-}5)$$

El punto en el que se unen los dos tramos cuadráticos será en $\lambda_i^* = 0$. Será el intervalo en el que ni (3-4) ni (3-5) pueden cumplirse. Este intervalo lo definirán los valores de c_i y w_i^γ :

$$c_i \in \left[-w_i^\gamma, w_i^\gamma\right].$$

El caso (3-4) termina en $c_i = -\gamma_i$ mientras que el caso (3-5) termina en $c_i = w_i^\gamma$. Si c_i está comprendido entre esos valores no se pueden cumplir las condiciones en ninguno de los dos casos anteriores. El término positivo que multiplica el valor absoluto $w_i^\gamma \ge 0$ prevalece sobre cualquier valor negativo del término que multiplica c_i. Por ello, el único mínimo posible será:

$$\lambda_i^* = 0 \quad si \quad w_i^\gamma \ge |c_i|. \qquad (3\text{-}6)$$

En las siguientes gráficas de ejemplo se puede observar el comportamiento que tendría $w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i$ para los 3 casos definidos en (3-4), (3-5), (3-6).

Para $w_i^\gamma < -c_i$, con mínimo en $\frac{-(w_i^\gamma + c_i)}{2w_i^h}$:

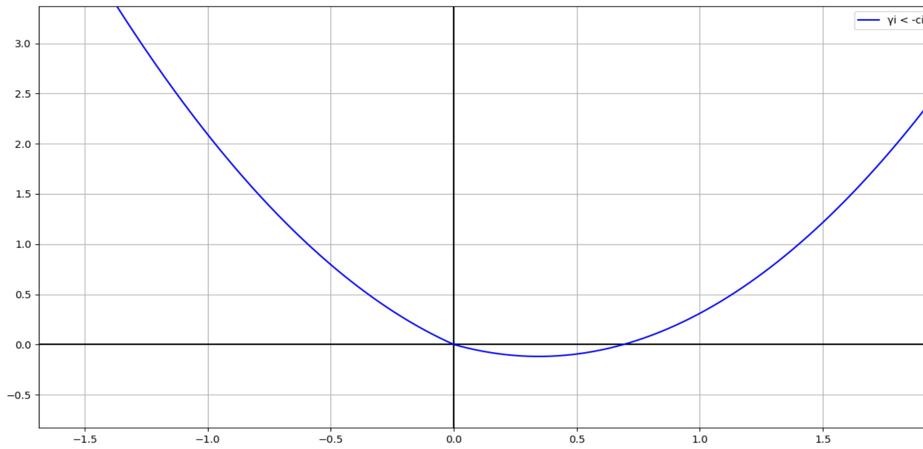

Figura 3-2. Caso $\lambda_i > 0$.

Para $c_i \in [-w_i^\gamma,\ w_i^\gamma]$, con el único mínimo posible en 0:

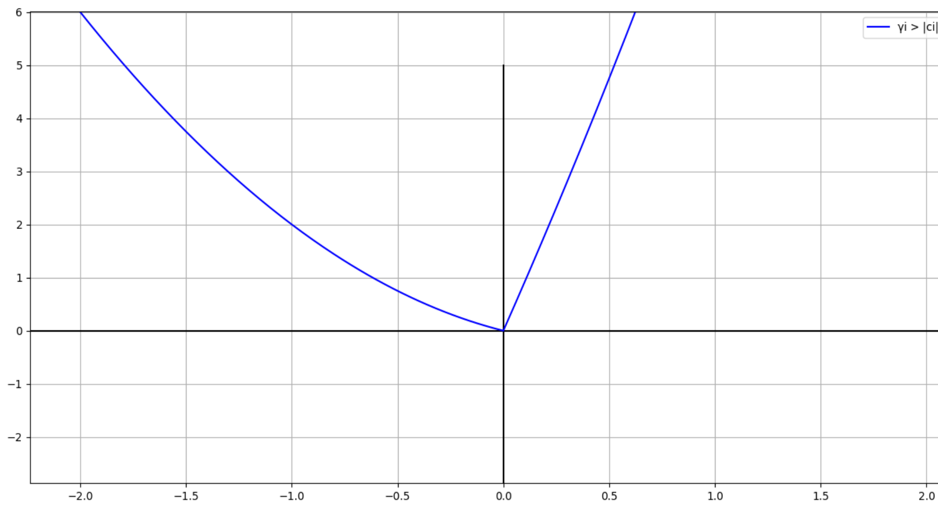

Figura 3-3. Caso $\lambda_i = 0$

Para $w_i^\gamma < c_i$, con mínimo en $\frac{w_i^\gamma - c_i}{2w_i^h}$:

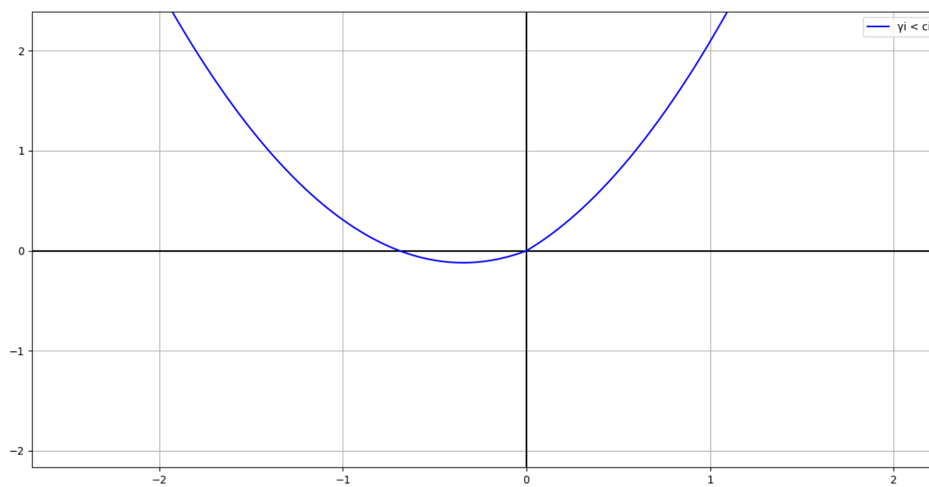

Figura 3-4. Caso $\lambda_i < 0$

En definitiva, la obtención de cada uno de los escalares óptimos que conforman λ_β^* se resume en estos tres casos, donde, como ya se ha mencionado, el caso $\lambda_i = 0$ comprende el rango de valores de c_i que no permitirían cumplir ninguno de los dos casos:

$$\lambda_{\beta,i}^* = \begin{cases} \dfrac{-(\,w_i^\gamma + c_i)}{2 \cdot w_i^h} & si \quad c_i < -\,w_i^\gamma \\ 0 & si \quad c_i \in \,[-w_i^\gamma,\, w_i^\gamma] \\ \dfrac{w_i^\gamma - c_i}{2 \cdot w_i^h} & si \quad w_i^\gamma \, < c_i \end{cases} . \qquad (3\text{-}7)$$

Como puntualización, hay que recordar que para $\alpha_h = 0,\ \alpha_\gamma = 0$ se cumple respectivamente:

$$w_i^h = 1 \ \forall \ \{w_1^\gamma : i = 1, \dots, N\}, \quad w_i^\gamma = \gamma \ \forall \ \{w_1^\gamma : i = 1, \dots, N\}.$$

3.3 Algoritmo Gradencial

Una vez desacoplado el problema, se puede utilizar cualquier método gradencial para hallar la solución escalar $\lambda_{\beta,i}^*$ e iterar modificando β hasta alcanzar el máximo de $\varphi(\beta)$ en $\varphi(\beta^*) = J_\gamma^*$. Es decir, un algoritmo que en cada iteración se acerque a la solución en la dirección del gradiente de la función a optimizar. En concreto se ha elegido el método FISTA (*Fast Iterative Shrinkage-Thresholding Algorithm*). Es una extensión del algoritmo gradencial clásico y, debido a su rapidez, es adecuado para problemas no estrictamente convexos (como el problema dual) y es adecuado para resolver problemas con mucha densidad de datos. Por ello, se ha considerado como óptimo. Además, su ratio de convergencia es, en general, varios órdenes superiores que su predecesor ISTA (FISTA: $O\left(\frac{1}{k^2}\right)$, ISTA: $O\left(\frac{1}{k}\right)$).

3.3.1 Gradiente del funcional dual

Como ya se ha mencionado, la minimización del funcional dual es un problema de optimización cóncavo y, debido a que el primal es estrictamente convexo, se pueden utilizar los resultados presentados en [8] para obtener la siguiente cota inferior de $\varphi(\beta + \Delta\beta)$:

$$\varphi(\beta + \Delta\beta) \geq \varphi(\beta) + \Delta\beta^T\left(R\lambda_\beta^* - r\right) - \frac{1}{4}\Delta\beta^T R H_w^{-1} R^T \Delta\beta, \qquad (3\text{-}8)$$

donde $-\frac{1}{4} \cdot \Delta\beta^T R H_w^{-1} R^T \Delta\beta$ es un término cuadrático negativo, $\left(R\lambda_\beta^* - r\right)$ es el gradiente de $\varphi(\beta)$ y λ_β^* es el valor óptimo de λ para una β dada. Al contrario que el problema primal, el problema dual es diferenciable por lo que diferenciando respecto a $\Delta\beta$ se obtiene el valor óptimo que maximiza la cota inferior de $\varphi(\beta + \Delta\beta)$. Esto es:

$$\left(R\lambda_\beta^* - r\right) - \frac{1}{4} \cdot R H_w^{-1} R^T \Delta\beta^* = 0,$$

por lo que despejando el incremento de β óptimo queda la siguiente expresión:

$$\Delta\beta^* = 2 \cdot (R H_w^{-1} R^T)^{-1}\left(R\lambda_\beta^* - r\right). \qquad (3\text{-}9)$$

Partiendo de (3-2) y (3-9) se pueden entrever los pasos de un algoritmo que llegue a la solución buscada. Teniendo en cuenta que la condición de salida o éxito del algoritmo consiste en cumplir la restricción de igualdad del primal: $\|R\lambda^* - r\|_2 \leq \epsilon$. Por debajo de este límite establecido ϵ, se considerará que el algoritmo ha convergido. Mientras que no se cumpla esta condición, se deberá recalcular (3-2) y (3-9).

3.3.2 ISTA

Tal y como se ha mencionado, el método ISTA es el predecesor del FISTA. Por ello, se expondrá el método ISTA en primera instancia. Este se basa en una aproximación gradencial a la solución por lo que el paso principal consiste en evaluar el gradiente en x_k y hallar x_{k+1} sumando x_k con la cota inferior máxima. En la figura siguiente se toma $\beta = x_k$ y se calcula la cota inferior máxima $\Delta\beta^*$ con λ_β^*.

Figura 3-5. Aproximación de ISTA

El cálculo de β_k estaría dado por la union de (3-2) y (3-9) en la que cada vez que se obtiene un β_k se debe recalcular las componentes de $\lambda_{\beta_k}^* = \lambda_{\beta_k,i} : i = 1, \dots, N.$ Es decir, se deben resolver N problemas de optimización escalares:

$$\lambda_{\beta_k,i}^* \rightarrow \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i \ , i : 1, \dots, N.$$

3.3.3 Implementación de ISTA en Código Python

Esta implementación se usará únicamente para comparar en **3.5.1 ISTA frente a FISTA** el comportamiento y la rapidez de ISTA con el algoritmo de FISTA definido en la siguiente sección.

$\underline{ISTA}$

Se requiere: $H_w \in \mathbb{R}^{\mathrm{NxN}},\ \Gamma_w \in \mathbb{R}^{\mathrm{NxN}},\ r \in \mathbb{R}^{\mathrm{n+1}},\ R \in \mathbb{R}^{\mathrm{(n+1)xN}}$.

Se obtiene: $[\lambda^*, x_k^*]$.

I) Sea la tolerancia permitida ϵ.

II) **Se inicializa** $\beta_0 = 0, k = 0$.

III) $c = R^T \beta_k$.

IV) $\lambda_{\beta_k,i}^* \rightarrow \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i \ ,\ i : 1, \dots, N.$

V) **Si se cumple** $\left\| R\lambda_{\beta_k}^* - r \right\|_2 \leq \epsilon \rightarrow$ **EXIT**.

VI) $\beta_{k+1} = \beta_k + \Delta\beta^* = \beta_k + 2 \cdot (RH_w^{-1}R^T)^{-1}\left(R\lambda_{\beta_k}^* - r\right)$.

VII) $k = k + 1$, **goto** III).

3.4 FISTA

La mayor diferencia entre el método FISTA y el ISTA es la elección del punto en el que evaluar el gradiente de la función a optimizar. En el caso del ISTA, se calcula x_{k+1} considerando únicamente el punto anterior x_k. No obstante, para el algoritmo FISTA, se toma un punto auxiliar cuidadosamente elegido y calculado a través de una combinación lineal de los puntos anteriores x_k, x_{k-1}. Es decir, en la siguiente iteración el gradiente se evaluará en un punto extrapolado β.

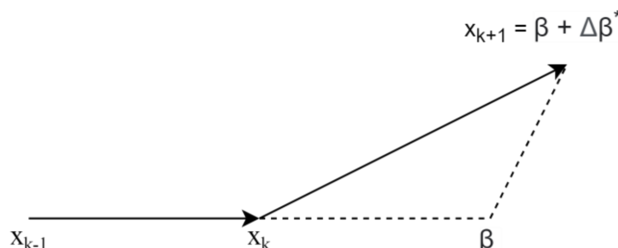

Figura 3-6. Punto extrapolado

Esta inclusión de un punto extrapolado fue propuesta por Nesterov'83. Para calcularlo hará falta introducir el término de *momentum* o de amortiguamiento: $\theta_k = \left(\frac{t_k - 1}{t_{k+1}}\right)$ donde en cada iteración t_k variará de la siguiente forma: $\mathrm{t_k} = \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4t_{k-1}^2}$. Por ello, la relación entre los puntos anteriores y el punto extrapolado será: $\beta_k = x_k + \theta_k \cdot (x_k - x_{k-1})$.

Es importante señalar que FISTA no es un método descendente (tal vez no se cumpla: $J(x_k) > J(x_{k+1})$) y suele tener algunos problemas prácticos de oscilación o *Rippling*. En el punto **3.5 Optimización de la Optimización** se verán con más detalle.

3.4.1 Implementación de FISTA en Código Python

Finalmente, en este TFG se ha decidido dividir el algoritmo FISTA en nueve pasos principales. Este será el algoritmo general. Variantes con la estrategia *Restart* se mostrarán más adelante como ya se ha mencionado:

$\underline{FISTA}$

Se requiere: $H_w \in \mathbb{R}^{\mathrm{NxN}},\ \Gamma_w \in \mathbb{R}^{\mathrm{NxN}},\ r \in \mathbb{R}^{\mathrm{n+1}},\ R \in \mathbb{R}^{\mathrm{(n+1)xN}},\ x_{ini} \in \mathbb{R}^{\mathrm{n+1}}$.

Se obtiene: $[\lambda^*, x_k^*]$.

I) Sea la tolerancia permitida ϵ.

II) **Se inicializa** $\beta_0 = 0;\ x_0,\ x_1 = x_{init};\ \ t_0 = 0, k = 1$.

III) $t_k = \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4t_{k-1}^2}$.

IV) $\beta_k = x_k + \frac{t_{k-1} - 1}{t_k}(x_k - x_{k-1})$.

V) $c = R^T \beta_k$.

VI) $\lambda_{\beta_k,i}^* \rightarrow \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i\ ,\ i : 1, \ldots, N$.

VII) **Si se cumple** $\left\|R\lambda_{\beta_k}^* - r\right\|_2 \leq \epsilon \rightarrow$ **EXIT.**

VIII) $x_{k+1} = \beta_k + \Delta\beta^* = \beta_k + 2 \cdot (RH_w^{-1}R^T)^{-1}\left(R\lambda_{\beta_k}^* - r\right)$.

IX) $k = k + 1$, **goto** III).

Aunque el objetivo principal de este algoritmo es obtener el valor óptimo de λ que minimice el funcional, es muy interesante tener en cuenta el valor final de la variable dual β_k^* como se verá en **4.3.3 Eficiencia del Algoritmo 1.** Respecto al paso VII) se podría usar otro tipo de norma como $\|\cdot\|_1, \|\cdot\|_\infty$, etc.

3.4.2 Verificación de la implementación en Python

Puesto que el grueso del TFG se basa en el buen desempeño de este algoritmo, se ha considerado adecuado realizar una comprobación de esta implementación con la optimización cuadrática de MatLab (*quadProg*). Para poder usar esta función de MatLab es necesario introducir un problema canónico cuadrático con una restricción de igualdad y otra de desigualdad:

$$\min_{x \in \mathbb{R}^{\mathrm{N}}} \frac{1}{2} x^T Q x + q^T x$$
$$s.t.\ \ Ax = b;\ Cx \leq 0,$$

por lo que hay que diseñar Q, x, A, b, C de modo y forma que el problema que resuelva *quadprog* sea idéntico al planteado en este TFG. En primer lugar, la variable de decisión será $x = [\lambda, d]^T \in \mathbb{R}^{\mathrm{Nx2}}$ donde d se define para tratar con el valor absoluto, ya que este no es diferenciable. Se fuerza $-d \leq \lambda \leq d$ (siendo ésta la restricción de desigualdad de la normalización de la suma de λ_i). El problema planteado en el TFG se expresa en términos de $[\ \lambda, d]$. Por todo esto:

$$\lambda^T\lambda + \gamma\overrightarrow{1^T}d \;\rightarrow\; x = [\lambda, d]^T, \qquad x^TQx = 2\cdot\lambda^T\lambda, \qquad q^Tx = \gamma\overrightarrow{1^T}d. \tag{3-10}$$

Comenzando con la desigualdad, se debe conseguir que $Cx \leq 0$ sea lo mismo que $-d \leq \lambda \leq d$. Esta última desigualdad puede expresarse de la siguiente forma:

$$\begin{bmatrix}-d\\ \lambda\end{bmatrix} \leq \begin{bmatrix}\lambda\\ d\end{bmatrix},$$

y pasando el mayor término de la desigualdad al otro miembro queda:

$$\begin{bmatrix}-d-\lambda\\ \lambda-d\end{bmatrix} \leq 0. \tag{3-11}$$

Por ende (3-11) cumple siempre las restricciones que Cx debe cumplir:

$$C\begin{bmatrix}\lambda\\ d\end{bmatrix} = \begin{bmatrix}-d-\lambda\\ \lambda-d\end{bmatrix} \rightarrow C = \begin{bmatrix}-I & -I\\ I & -I\end{bmatrix}.$$

La restricción de igualdad deberá ser la misma que el problema original por lo que:

$$A\begin{bmatrix}\lambda\\ d\end{bmatrix} = r \rightarrow A = [R, \vec{0}],$$

donde $\vec{0} \in \mathbb{R}^{\mathrm{MxN}}$, $R \in \mathbb{R}^{\mathrm{MxN}}$. De la misma manera se debe cumplir: $q^Tx = \gamma\overrightarrow{1^T}d$:

$$q^T\begin{bmatrix}\lambda\\ d\end{bmatrix} = \gamma\vec{1}d \rightarrow q^T = \gamma\cdot[\vec{0}, \vec{1}],$$

donde $\vec{0} \in \mathbb{R}^{\mathrm{N}}$, $\vec{1} \in \mathbb{R}^{\mathrm{N}}$. Por último, se debe cumplir: $x^TQx = 2\cdot\lambda^T\lambda$:

$$[\lambda^T \;\; d^T]Q\begin{bmatrix}\lambda\\ d\end{bmatrix} = 2\cdot\lambda^T\lambda \rightarrow Q = 2\cdot\begin{bmatrix}I & 0\\ 0 & 0\end{bmatrix},$$

donde tanto en C como en Q, la matriz $I \in \mathbb{R}^{\mathrm{NxN}}$ es la matriz identidad.

Una vez el problema tiene el formato necesario para la función *quadprog*, se comprueba, con los mismos valores de entrada, que la implementación de FISTA es correcta obteniéndose la misma solución λ^* en todos los casos.

3.5 Optimización de la Optimización

Durante el desarrollo de cualquier código en cualquier lenguaje de programación, la optimización de este y la reducción de pasos e instrucciones innecesarias está (o debería estar) siempre presente en la cabeza del programador. En algunos programas no es tan necesaria una atención constante a la eficiencia debido a un pequeño coste computacional intrínseco o a que la diferencia entre tardar $0.1\mu s$ o $50ms$ no es relevante. En este trabajo es de obligado cumplimiento bajar de esos $50ms$ a $0.1\mu s$ y a mucho menos si pudiera ser.

Durante todo el TFG se ha procurado velar por la eficiencia del código y sobre todo la eficiencia del algoritmo de este capítulo. Esto se debe a que los demás algoritmos usados en **4 Predicciones Intervalares** se basan en repetir FISTA miles y miles de veces como se verá en **4.4.3 Implementación del Algoritmo 2 en Python**. Las predicciones intervalares se consiguen anidando un algoritmo sobre otro. Y FISTA es la base.

3.5.1 ISTA frente a FISTA

Para empezar, se va a ejemplificar por qué se ha usado FISTA y por qué se ha asegurado que tiene una ratio de velocidad de convergencia superior. Los dos algoritmos se usarán para resolver exactamente el mismo problema y se mostrará una gráfica *iteraciones* – log ($\|R\lambda^*_{\beta_k} - r\|_2$) que ejemplifique el acercamiento a la función óptima por iteración. Es decir, cómo de "eficiente" es cada iteración y cuántas hacen falta para llegar a una precisión predefinida. La norma $\|R\lambda^*_{\beta_k} - r\|_2$ será una medida del grado de optimalidad de la solución actual, donde valores muy pequeños implican que los valores de $\lambda^*_{\beta_k}$ obtenidos por el algoritmo están muy cerca del óptimo.

En otra gráfica se representará *tiempo* - $\log\left(\left\|R\lambda^*_{\beta_k} - r\right\|_2\right)$ para observar el acercamiento a la función óptima por unidad de tiempo. El problema de optimización para testear los algoritmos será el mismo que el utilizado durante todo el trabajo sin pesos (por lo que las matrices $H_w = I, \Gamma_w = \gamma \cdot I$):

$$
\begin{aligned}
J_\gamma(x_k, D) \quad &= \quad \min_{\lambda_1,\dots,\lambda_N} \sum_{i=1}^{N} \lambda_i^2 + \gamma \sum_{i=1}^{N} |\lambda_i| \\
s.t. \qquad & x_k = \sum_{i=1}^{N} \lambda_i x_i \\
& 1 = \sum_{i=1}^{N} \lambda_i .
\end{aligned}
\qquad (3\text{-}12)
$$

Así pues, el problema a optimizar será el mismo que en (2-2):

$$
\begin{aligned}
J_\gamma^* = \min_{\lambda \in \mathbb{R}^N} \lambda^T \lambda + \gamma \|\lambda\|_1 \\
s.t. \quad R\lambda = r,
\end{aligned}
$$

donde el set de datos $D = \{x_i : i = 1, \dots, N\} \subset X$ siendo $N = 2000$. Los N valores de D serán aleatorios entre 1 y 0. Quiere decir que se resolverá un problema de minimización de 2000 variables λ_i aunque la dimensión de la variable dual será 2, ya que $n = 1$.

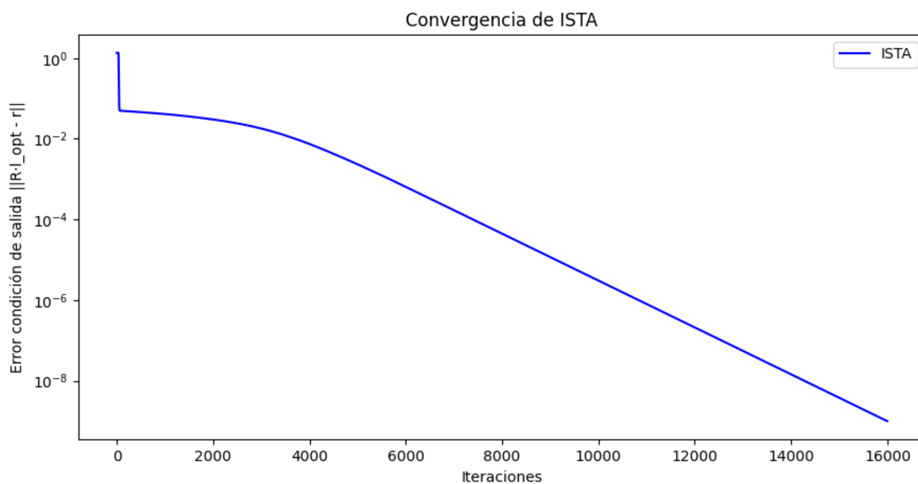

Figura 3-7. Convergencia ISTA

Se puede observar que ISTA ha necesitado 600 iteraciones para llegar a una precisión de 10^{-3} y un poco más de 16000 para llegar a 10^{-9}.

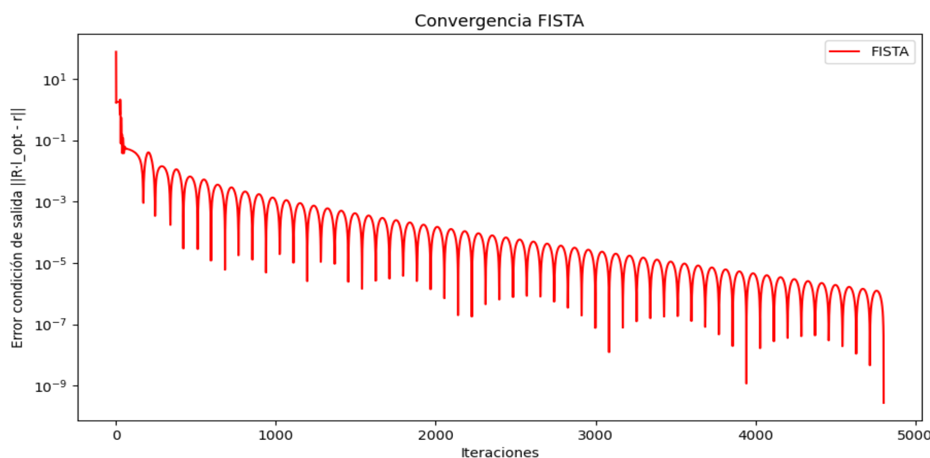

Figura 3-8. Convergencia FISTA

En cambio, el método FISTA consigue unas precisiones de 10^{-3} con tan solo 232 iteraciones. Por su naturaleza no descendente y el problema práctico de *Rippling* antes comentado, se consigue una precisión de $1.6 \cdot 10^{-9}$ a las 3900 iteraciones, pero casi 1000 más en obtener 10^{-9}. Esto indica claramente que hay un gran potencial desaprovechado en este caso. ¿No se podría tratar de detener el algoritmo cuando llegue a un máximo de precisión como en $1.6 \cdot 10^{-9}$ y replantear el siguiente paso si este se aleja de la solución óptima? O ¿no se podría reiniciar el algoritmo cuando ya se ha conseguido una mejora en términos de optimalidad de forma que se asegure la convergencia lineal? Esta última pregunta se tratará en **3.5.3 Problema práctico *Rippling***

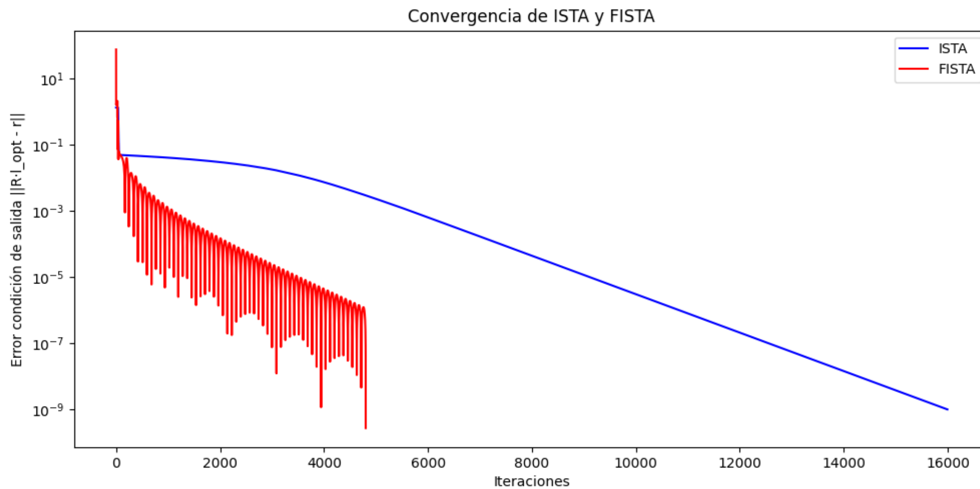

Figura 3-9. Convergencia ISTA - FISTA

En definitiva, el método ISTA provee de una convergencia más constante mientras que FISTA, a priori, es mucho más rápido, pero tiene más posibilidades de problemas prácticos como el que se puede observar.

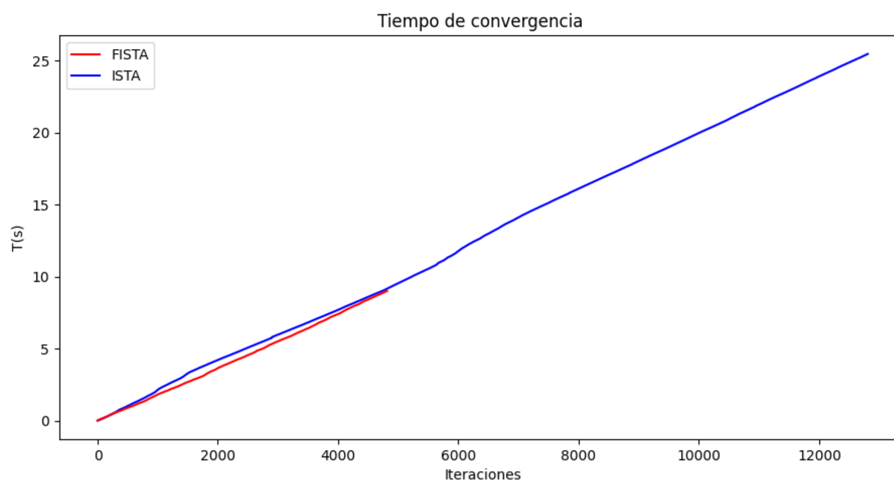

Figura 3-10. Tiempo de cómputo frente a iteraciones de ISTA y FISTA

El hecho de que las dos rectas de esta gráfica estén tan pegadas indica que cada iteración de ISTA y FISTA tarda en ejecutarse un tiempo similar. Es decir, llanamente las iteraciones de FISTA son más eficientes a la hora de acercarse a la solución óptima y no implican un mayor coste computacional ya que los pasos III) y IV), que son los cálculos de más que tiene FISTA frente a ISTA, son despreciables respecto al total.

3.5.2 Eficiencia del código

A la hora de implementar FISTA en un código de Python se ha decidido realizar unos precálculos para minimizar el coste computacional del paso VIII):

$$x_{k+1} = \beta_k + \Delta\beta = \beta_k + (RH_w^{-1}R^T)^{-1}\left(R\lambda_{\beta_k}^* - r\right),$$

donde se realizan 6 multiplicaciones de matrices y dos inversiones cuando el término $(RH_w^{-1}R^T)^{-1}$es constante para todas las iteraciones. Por ello se reorganiza y se calcula previamente:

$$2(RH_w^{-1}R^T)^{-1}r \text{ y } 2(RH_w^{-1}R^T)^{-1}R.$$

Este precálculo se añade como un paso previo al I) de modo que el paso VIII) se reduce a una única multiplicación de matrices y sin inversiones:

$$VIII') \quad x_{k+1} = \beta_k - 2(RH_w^{-1}R^T)^{-1}r + 2(RH_w^{-1}R^T)^{-1}R\lambda_{\beta_k}^*.$$

Se considera que la suma y resta de matrices no tiene impacto apenas en el tiempo de cómputo, por lo que el coste de este paso se ha reducido considerablemente [1]. Hay que decir que para el ejemplo siguiente se ha reducido N de 2000 a 200 valores aleatorios pues no es necesario para representar el inmenso ahorro de tiempo. Se han hecho 10 pruebas y la media ha sido de 0.53 s con modificación y 82.6 s sin esta.

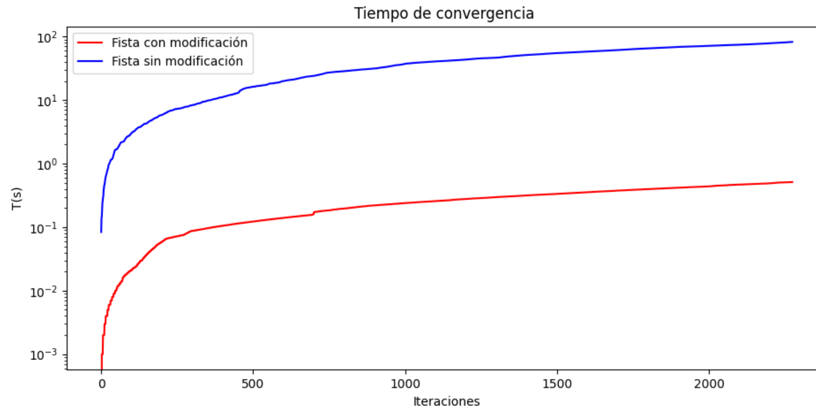

Figura 3-11. Mejora del coste computacional

Ha sido necesario usar un eje de tiempo logarítmico para que se pudiera apreciar algo más que una recta horizontal roja en el eje X. Por supuesto, el tiempo de cómputo es dependiente del ordenador en el que se ejecute el código. No obstante, esta prueba representa el decremento de tiempo para un mismo ordenador, lo que sí es significativo.

3.5.3 Problema práctico *Rippling*

Como se observa en la gráfica de la Figura 3-8, el algoritmo FISTA tiene un comportamiento extraño respecto a otros algoritmos de *gradient descent* o el propio ISTA. Esto es debido a que FISTA no es un algoritmo estrictamente descendente y/o el problema no es estrictamente convexo o bien condicionado, lo que implica que por cada iteración no siempre se obtiene un resultado más cercano al óptimo y puede oscilar. El *momentum* propuesto por [8] del paso III) de FISTA decide en gran medida si el algoritmo oscila o no:

$$\beta_{k+1} = x_{k+1} + \left(\frac{t_k - 1}{t_{k+1}}\right)(x_{k+1} - x_k).$$

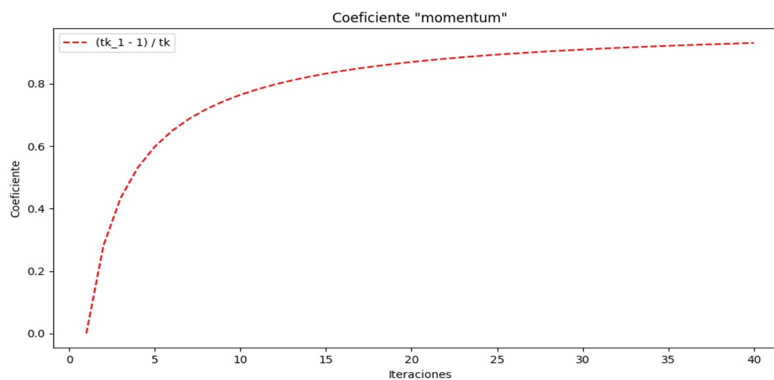

Figura 3-12. Evolución del coeficiente $\frac{t_{k-1}-1}{t_k}$ frente a iteraciones

[1] Los ejemplos mostrados en la subsección anterior ya tenían esta modificación añadida. Tanto FISTA como ISTA.

Cuanto más se acerque el *momentum* a 0, es decir $t_k \cong 1$, más se parecerá el algoritmo FISTA al algoritmo ISTA. Es decir, $\beta_{k+1} \cong x_{k+1}$ enlenteciéndose más la aproximación. En cambio, cuanto más se acerque t_k a 1, más sobreestimado estará el *momentum* y más posibilidades hay de que ocurran las oscilaciones. Existe un punto intermedio entre 0 y 1 de *momentum* que es óptimo. No obstante, el cálculo de este óptimo esta fuera de los límites de este TFG por lo que se limitará a conocer la existencia del mismo y asimilar que no podrá usarse. Habrá que proponer soluciones alternativas.

Durante el cómputo del algoritmo puede llegar una iteración en la que este *momentum* se sobreestime. Por ello, una de las soluciones más aceptadas para solucionar el *Rippling* consiste en reiniciar FISTA (*Restart*). Se toma el mismo punto en el que se encuentra la variable dual β_k y se toma como punto inicial x_{ini} en un algoritmo FISTA reiniciado con $t_0 = 1$. Es decir, se reinicia el *momentum* a 0.

Defínase, por comodidad, $\theta_{k+1} = \frac{t_k - 1}{t_{k+1}}$. En la siguiente gráfica de la Figura 3-13, similar a la que aparece en [12], se ejemplifica la sobreestimación y subestimación de θ_{k+1} sobre el pseudo-óptimo θ^*, que en este caso concreto resulta ser 0.62·θ_{k+1}.

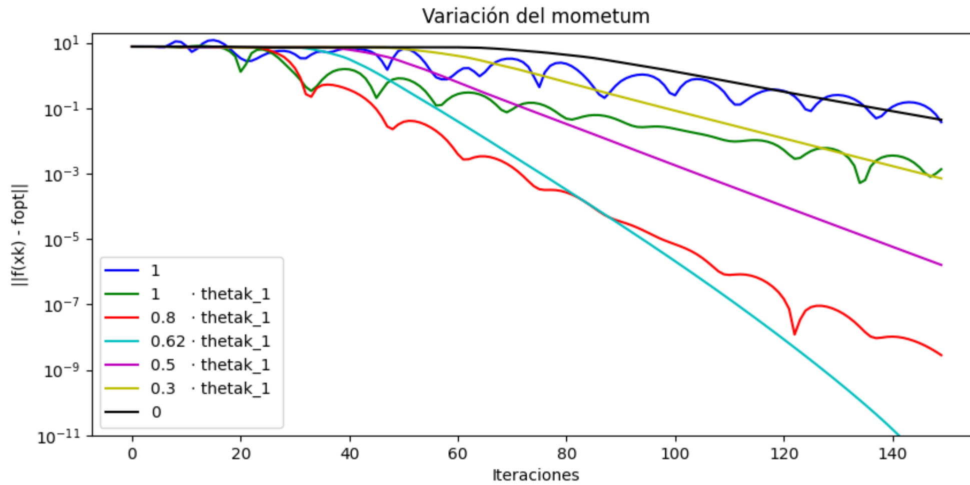

Figura 3-13. Variación del momentum en torno al pseudo-óptimo

Es decir, en determinados problemas de optimización, que no son suaves y convexos, puede ocurrir que el *momentum* θ_{k+1} de [8] no sea el adecuado como presenta la Figura 3-8. En estos casos, cuando FISTA empieza a oscilar, se debe reiniciar tomando el *momentum* a 0. Otra posibilidad es, llegado a una mejora predefinida en términos de optimalidad, reiniciar el algoritmo.

A continuación, se presentará una estrategia simple de reinicio propuesta en [9] que permite hacer frente a las complicaciones planteadas y asegura una convergencia lineal usando la segunda condición de reinicio comentada.

Sea j el contador de reinicios acometidos y sea z_j la condición inicial de cada reinicio. Defínase ahora ρ_j como la norma del gradiente inicial de cada reinicio evaluado en z_j, siendo este el gradiente del funcional dual concretado en **3.3.1 Gradiente del funcional dual**.

La idea principal del algoritmo es la de reiniciar FISTA cada vez que el gradiente disminuya e veces respecto al gradiente inicial (siendo e el número de Euler). Esto es:

$$\|g(y_k)\|_2 \leq \frac{\rho_j}{e}.$$

Cada vez que se cumpla esta condición, se guarda el punto extrapolado y_k en z_{j+1} y se actualiza la condición de salida a la vez que se reinicia el *momentum* con $t_k = 1$:

$$p_{j+1} = \left\|g(z_{j+1})\right\|_2.$$

Se podría decir que, en cada iteración de reinicio, se "exprime" al gradiente hasta que se empequeñece. Además, en [9] se demuestra que si la condición de salida "espera" hasta que el gradiente se reduzca e veces, la convergencia será lineal. Tras varios reinicios, es fácil entender que el gradiente se habrá dividido e^j lo que asegura la convergencia. Con estos pasos añadidos a **3.4.1 Implementación de FISTA en Código Python** se monta el siguiente algoritmo:

En primer lugar, defínanse, por comodidad, un pequeño algoritmo que calcule ΔS y λ_s para $s, H_w, \Gamma_w,$ r, y R^T llamado $MaxCotInf$:

<u>$MaxCotInf$</u>

Se requiere[2]: s, H_w, R^T, r, Γ_w.

Se obtiene: $\Delta S, \lambda_s$.

I) $c = R^T s$.

II) $\lambda_{s,i}^* \to \arg\min_{\lambda_i \in \mathbb{R}} w_i^h \lambda_i^2 + w_i^\gamma |\lambda_i| + c_i \lambda_i$, $i : 1, \ldots, N$.

III) $\Delta S = 2 \cdot (R H_w^{-1} R^T)^{-1} (R\lambda_s^* - r)$.

IV) **Devuelve** $[\Delta S, \lambda_s^*]$.

<u>$FISTA\ Based\ Restart$</u>

Se requiere: lo necesario para FISTA.

Se obtiene: $[\lambda^*, x_k^*]$.

I) Sea la tolerancia permitida ϵ.

II) **Se inicializa** $t_0 = 1, k = 0,\ j = 0,\ z_0 = x_{ini}$.

III) $[\Delta Z, \lambda_{z_0}^*] = MaxCotInf(z_0, H_w, R^T, r, \Gamma_w)$.

IV) $\rho_0 = \|R\lambda_{z_0}^* - r\|_2$.

V) $x_0,\ \beta_0 = z_0 + \Delta Z$.

VI) **do**

VII) $k = k + 1$.

VIII) $[\Delta\beta,\ \lambda_{\beta_k}^*] = MaxCotInf(\beta_k, H_w, R^T, r, \Gamma_w)$.

IX) $x_k = \beta_k + \Delta\beta$.

X) $t_k = \frac{1}{2} + \frac{1}{2}\sqrt{1 + 4t_{k-1}^2}$.

XI) $\beta_k = x_k + \frac{t_{k-1}-1}{t_k}(x_k - x_{k-1})$.

XII) **Si se cumple** $\|R\lambda_{\beta_k}^* - r\|_2 \le \frac{\rho_j}{e}$ **hacer**

XIII) $j = j + 1$.

XIV) $z_j = \beta_k, t_k = 1$.

XV) $[\Delta Z, \lambda_{z_j}^*] = MaxCotInf(z_j, H_w, R^T, r,\ \Gamma_w)$.

XVI) $\rho_j = \|R\lambda_{z_j}^* - r\|_2$.

XVII) $x_k,\ \beta_k = z_j + \Delta Z$.

XVIII) **Fin**

XIX) **while** $p_j \ge \epsilon$;

XX) **Devuelve** $\lambda_{z_j}^*, x_k$.

[2] La matriz H_w y Γ_w ya contienen los pesos asociados al término cuadrático de la función de coste y al valor absoluto respectivamente.

Se recupera la Figura 3-8 en la que se apreciaban unas oscilaciones excesivas y se compara con el nuevo algoritmo para una precisión de 10^{-14}.

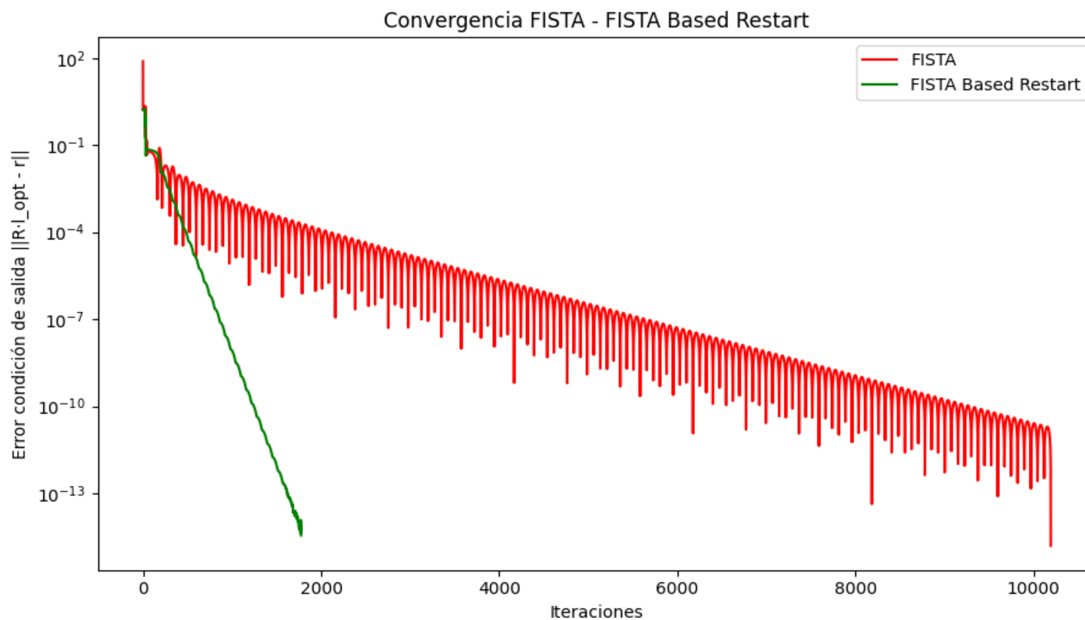

Figura 3-14. Convergencia FISTA – FISTA restart

Para este caso, es un acierto usar la estrategia de reinicio. Tal y como se aseguraba en [9], la convergencia es lineal y la ratio de convergencia es muy superior a FISTA con *rippling*. Es interesante señalar que el periodo de las pequeñas oscilaciones, que no es más que el número de veces que sucede el reinicio, no depende del tamaño de la norma del gradiente. Esto es así porque los reinicios ocurren con una relación fija entre gradientes de $\frac{1}{e}$. Esto se ve aún más claro al usar unos ejes logarítmicos y que la convergencia sea una línea recta ya que la ratio de convergencia dependerá de e^{j}.

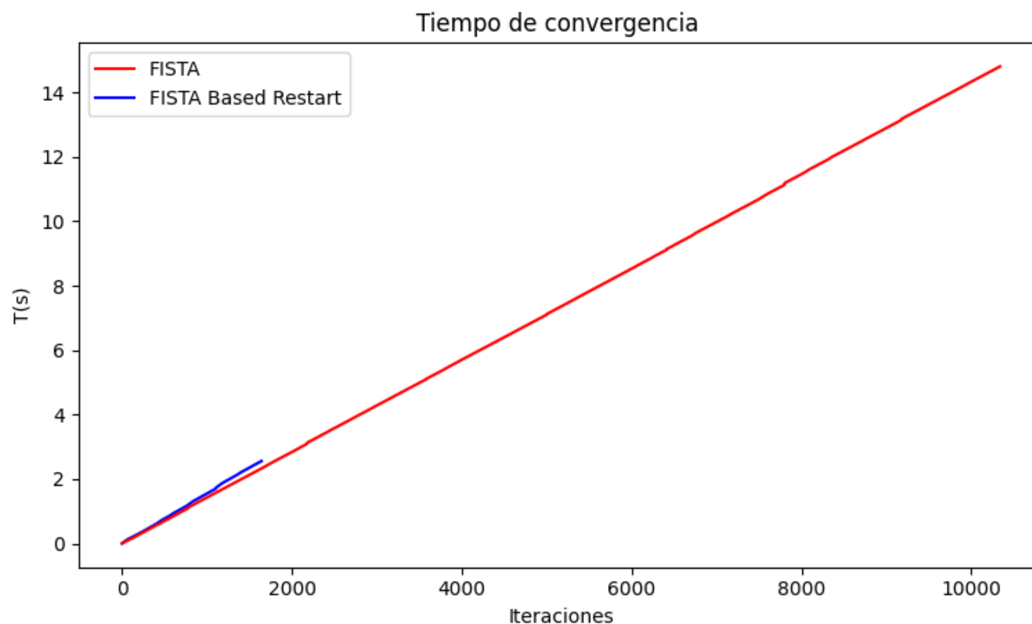

Figura 3-15. Tiempo de cómputo por iteración. FISTA - FISTA restart

Esta gráfica es interesante ya que puede parecer que una iteración de FISTA con reinicio consume más tiempo ya que llama a $MaxCotInf$ hasta tres veces. Esto implica repetir tres veces el paso más costoso de todo el algoritmo. No obstante, $MaxCotInf$ cuenta con los precálculos definidos en la subsección **3.5.2 Eficiencia del código** por lo que su coste computacional es barato. Desafortunadamente, no siempre es adecuado usar la estrategia de reinicio pues para problemas bien acotados en los que FISTA no oscila u oscila poco pueden llegar a enlentecer el cómputo, por lo que sería necesario tenerlo en cuenta.

4 PREDICCIONES INTERVALARES

La información es poder.

Úrsula Morón Pachón

En este capítulo se unirán los dos pilares desarrollados en **2 Cálculo de la Similitud y Disimilitud** y en **3 Optimización**. Su objetivo es, en vez de obtener una predicción numérica de una serie de salidas como ocurre en **2.3 Regresión con** Jy (z, D)**,** obtener una predicción intervalar de las mismas. Prefijando una probabilidad para la cual una salida y_k pertenece un intervalo, se calcula dicho intervalo minimizando su longitud. Es decir, se proveerá de un rango de valores donde, con una probabilidad elegida, se encuentre y_k.

Haciendo uso de las funciones de disimilitud se obtienen las funciones de densidad empírica condicionadas de cada una de las salidas que se quieren estimar. Teniendo la $fdec$, se puede obtener la función de distribución empírica condicionada con la que se podrá calcular la probabilidad de que la salida se encuentre entre dos valores superior e inferior. Se impondrá esta probabilidad (normalmente: 0.9-0.95) y encontrarán los valores superior e inferior para los cuantiles superior e inferior de la probabilidad impuesta.

Por simplicidad en el desarrollo de los fundamentos de las predicciones intervalares, se omitirá la inclusión de información local (pesos) tanto en el término cuadrático del funcional $J_\gamma(z, D)$ como en el valor absoluto, por lo que $H_w = I$, $\Gamma_w = \gamma \cdot I$. En las subsecciones **4.3.2 Implementación del Algoritmo 1 en Python** y en **4.4.3 Implementación del Algoritmo 2 en Python** se volverán a incluir.

4.1 Función de Densidad Empírica

Calculando la disimilitud de un vector z respecto a un conjunto de datos D con lo definido en el capítulo **2 Cálculo de la Similitud y Disimilitud** se puede obtener una función de densidad empírica. Es decir, se puede caracterizar la probabilidad de los valores de $z \in \mathbb{Z}$ para un set de datos D (se debe tener presente la relación entre funciones de similitud y disimilitud de (2-1)).

Para ello se hará uso de la función de disimilitud descrita en (2-2). En esencia, la función de densidad empírica se construye con la relación entre la similitud de un vector dado z respecto a la integral de toda la similitud del dominio $\mathbb{Z}$. Formalmente:

Siendo el conjunto $D = \{z_1, \dots, z_N\} \subset \mathbb{Z}$ con los parámetros que definen la familia de funciones $\gamma \geq 0, c \geq 0$, la función de densidad empírica, $fde_{\gamma,c}(z, D)$, se define como:

$$fde_{\gamma,c}(z, D) = \frac{\exp\left(-cJ_\gamma(z, D)\right)}{\int_{\mathbb{Z}} \exp\left(-cJ_\gamma(\hat{z}, D)\right) d\hat{z}}.$$

4.2 Función de Densidad Empírica Condicionada

Definida la función de densidad empírica para $z_i = [x_i, y_i]$, se puede obtener una estimación intervalar de la salida y_k condicionada a un x_k dado. Es lógico plantear una función de densidad empírica condicionada puesto que se quiere partir de una entrada x_k conocida para estimar un rango de valores de $\hat{y}_k$. Específicamente, siendo $x_k \in \mathcal{X},\ \ y_k \in \mathcal{Y} \subseteq \mathbb{R}$ y siendo D:

$$D = \left\{ z_i = \begin{bmatrix} y_i \\ x_i \end{bmatrix} : i = 1, \dots, N \right\} \subset \mathcal{X} \times \mathcal{Y},$$

la función de densidad empírica condicionada a x_k se define como:

$$fdec_{\gamma,c}(y, x_k, D) = \frac{\exp\left(-cJ_\gamma\left(\begin{bmatrix} y \\ x_k \end{bmatrix}, D\right)\right)}{\int_{\mathcal{Y}} \exp\left(-cJ_\gamma\left(\begin{bmatrix} \hat{y} \\ x_k \end{bmatrix}, D\right)\right) d\hat{y}}, \forall y \in \mathcal{Y} \qquad (4\text{-}1)$$

Siendo γ el parámetro definido en el capítulo anterior y c el parámetro dado para la relación entre funciones de similitud y disimilitud (2-1). De esta forma, (4-1) permite hallar la probabilidad de que se dé la salida y siendo la entrada x_k. Por un lado, aplicando la regresión con funciones de disimilitud, se obtenía una predicción $\hat{y}_k$ para x_k . Por otro lado, con $fec_{\gamma,c}$, se obtiene la probabilidad de que una salida y_k ocurra dada una entrada x_k. Cabe señalar que (4-1) provee de una familia de funciones de densidad empírica condicionada que dependen de dos parámetros: c y γ. Más adelante se podrá ver en detalle en que influye cada uno de ellos.

4.2.1 Discretización del conjunto $\mathcal{Y}$

Puesto que la idea de calcular la función de densidad empírica condicionada es obtener una estimación intervalar dada una entrada x_k con una probabilidad predefinida, se debe facilitar su cómputo.

Dicho esto, la mejor forma de aproximar la integral de (4-1) del dominio $\mathcal{Y}$ es con un sumatorio. Para ello, se debe simplificar el conjunto de las salidas $\mathcal{Y} \subseteq \mathbb{R}$. Idealmente, los límites de la integral deberían ser $[-\infty, +\infty]$ para contener todas las salidas posibles, lo que complica considerablemente el cálculo numérico de la misma. Es mucho más adecuado tomar $\overline{\mathcal{Y}}$ como un set finito delimitado por el máximo y el mínimo del set de datos D: $\overline{\mathcal{Y}} = \{\bar{y}_1, \dots, \bar{y}_M\}$.

Donde se cumple que $\bar{y}_j < \bar{y}_{j+1} : \ j = 1, \dots, M-1$. En definitiva, el set $\overline{\mathcal{Y}}$ se define como:

$$\bar{y}_1 = \min_{i=1,\dots,N} y_i$$

$$\bar{y}_M = \max_{i=1,\dots,N} y_i$$

$$\bar{y}_j = \bar{y}_1 + \left(\frac{\bar{y}_M - \bar{y}_1}{M-1}\right)(j-1), j = 1, \dots, M. \qquad (4\text{-}2)$$

Esto es así para asegurar que, con una probabilidad razonable, la salida y_k pertenezce a $\overline{\mathcal{Y}}$. Debido a esta simplificación y a la naturaleza de la regresión con funciones de disimilitud, la predicción de valores muy alejados del set de datos D se verá comprometida como ya se ha visto. Así pues, se define la función de densidad empírica condicionada discreta como:

$$\overline{fdec}_{\gamma,c}(y, x_k, D) = \frac{\exp\left(-cJ_\gamma\left(\begin{bmatrix} y \\ x_k \end{bmatrix}, D\right)\right)}{\sum_{j=1}^{M} \exp\left(-cJ_\gamma\left(\begin{bmatrix} \bar{y}_j \\ x_k \end{bmatrix}, D\right)\right)}. \qquad (4\text{-}3)$$

La aproximación de la integral será mejor cuantas mayores particiones M se tomen para crear el set $\overline{\mathcal{Y}}$. Como es lógico, también aumentará el coste computacional del que se trata de huir. En esencia, (4-3) calcula la similitud del par $[y, x_k]$ y lo normaliza con la suma de todos los valores de similud posibles para el set $\overline{\mathcal{Y}}$. Al normalizar entre 0 y 1 y por construcción de cualquier función de distribución se cumple:

$$\sum_{l=1}^{M} \overline{fdec}_{\gamma,c}(\bar{y}_l, x_k, D) = 1. \quad (4\text{-}4)$$

4.2.2 Función de distribución empírica condicionada discreta de $\overline{fdec}_{\gamma,c}$

De ahora en adelante, se definirá la función de distribución discreta de $\overline{fdec}_{\gamma,c}(y, x_k, D)$ como $Prob_{\overline{\mathcal{Y}}|x_k}$. Más concretamente, denotando $\bar{y}_l$ como el elemento l-ésimo de $\overline{\mathcal{Y}}$ (recordando: $\bar{y}_l < \bar{y}_{l+1},\ l = 1, \dots, M-1$), la probabilidad de que la salida y_k condicionada a x_k sea menor o igual que $\bar{y}_l$ será:

$$Prob_{\overline{\mathcal{Y}}|x_k}\{y \le \bar{y}_l\} = \sum_{j=1}^{l} \overline{fdec}_{\gamma,c}\big(\bar{y}_j, x_k, D\big), \quad (4\text{-}5)$$

de la misma manera, la probabilidad de que la salida y_k dado un x_k sea mayor o igual a $\bar{y}_l$ será:

$$Prob_{\overline{\mathcal{Y}}|x_k}\{y \ge \bar{y}_l\} = \sum_{j=l}^{M} \overline{fdec}_{\gamma,c}\big(\bar{y}_j, x_k, D\big). \quad (4\text{-}6)$$

4.2.3 Cuantiles empíricos condicionados superior e inferior

Siguiendo con la función de distribución descrita y teniendo en cuenta (4-5) y (4-6), se puede estimar un intervalo superior e inferior donde se encuentre la salida y para para una probabilidad elegida. Más formalmente: dada una entrada x_k, los parámetros que definen la función de similitud $\gamma \ge 0, c \ge 0$ y τ, se define y_τ^+ (el τ-cuantil condicionado superior $\tau \in (0,1)$) como el primer/menor valor de $\overline{\mathcal{Y}}$ que cumple:

$$Prob_{\overline{\mathcal{Y}}|x_k}\{y \le y_\tau^+\} \ge 1 - \tau,$$

y de la misma manera, el τ-cuantil empírico inferior denotado como y_τ^- se establece como el último/mayor valor de $\overline{\mathcal{Y}}$ que cumple:

$$Prob_{\overline{\mathcal{Y}}|x_k}\{y \ge y_\tau^-\} \ge 1 - \tau,$$

por lo que se determina que el valor a predecir y_k se encuentra contenido en un intervalo $[y_\tau^-,\ y_\tau^+]$ para una probabilidad mayor o igual a $1 - 2\tau$:

$$Prob_{\overline{\mathcal{Y}}|x_k}\{y \in [y_\tau^-,\ y_\tau^+]\} \ge 1 - 2\tau.$$

La longitud media de los intervalos estará fuertemente ligada a la elección de τ para los cuantiles superiores e inferiores. Un menor valor de τ implica asegurar que las salidas están contenidas en el intervalo con una mayor probabilidad y un mayor valor de τ implica unos intervalos con más violaciones en los que la probabilidad de que la salida esté contenida en el intervalo será menor.

4.3 Algoritmo 1. Estimación Intervalar

Habiendo definido un método sencillo de obtener el conjunto discreto $\overline{\mathcal{Y}}$ y las ecuaciones (4-5) y (4-6), se pueden entrever los pasos de un algoritmo que permita la predicción intervalar de la salida y_k, dada una entrada x_k, con una probabilidad de $1 - 2\tau$. Este algoritmo estará basado en el definido en [1]. Tal y como se detallará más adelante, se harán unas modificaciones a la hora de implementarlo en Python para minimizar el coste computacional.

4.3.1 Explicación del Algoritmo 1

Partiendo de un valor de entrada x_k, del cuantil τ (superior e inferior), de los parámetros $\gamma \geq 0, c \geq 0$, del set de datos D y del conjunto discreto de salidas $\overline{\mathcal{Y}} = \{\bar{y}_1, \dots, \bar{y}_M\}$ creado a partir de (4-2), el algoritmo se puede dividir en dos partes bien diferenciadas:

En primera instancia, se debe calcular la función de densidad empírica condicionada discreta dado x_k por lo que para cada $\bar{y}_j,\ j = 1, \dots, M$ se deberá calcular la función de similitud y dividir por el factor de normalización. Con esta función de densidad se obtendrá la función de distribución condicionada. Específicamente:

1. Se crea un vector $S = [s_1, \dots, s_M]$ donde se guarda la similaridad de cada par $[\bar{y}_j, x_k]$ con el conjunto D:

$$s_j = \exp\left(-cJ_\gamma\left(\begin{bmatrix}\bar{y}_j\\ x_k\end{bmatrix}, D\right)\right). \quad (4\text{-}7)$$

2. En otro vector $P = [p_1, \dots, p_M]$ se guarda la probabilidad de cada par $[\bar{y}_j, x_k]$ usando (4-3):

$$p_j = \overline{fdec}_{\gamma,c}(\bar{y}_j, x_k, D) = \frac{s_j}{\sum_{l=1}^{M} s_l}.$$

En segundo lugar, se realizarán dos sumatorios del vector P: partiendo desde el principio de $\overline{\mathcal{Y}}$ hasta M y partiendo desde M hasta el menor valor de $\overline{\mathcal{Y}}$. Ambos sumatorios se detendrán en cuanto la suma acumulada supere la probabilidad $1 - \tau$. Concretamente:

3. Se calculan los índices l_τ^+ y l_τ^- referidos al τ-cuantil superior e inferior condicionado:

$$l_\tau^- = el\ mayor\ índice\ l\ con\ el\ que\ se\ cumple \sum_{j=l}^{M} p_j \geq 1 - \tau.$$

$$l_\tau^+ = el\ menor\ índice\ l\ con\ el\ que\ se\ cumple \sum_{j=1}^{l} p_j \geq 1 - \tau.$$

4. Se guardan los valores inferior y superior del intervalo como $y_\tau^- = \bar{y}_{l_\tau^-}, y_\tau^+ = \bar{y}_{l_\tau^+}$.

En definitiva, para una entrada x_k el valor y_k estará contenido en el intervalo $\left[\bar{y}_{l_\tau^-}, \bar{y}_{l_\tau^+}\right]$ con una probabilidad de $1 - 2\tau$. Tal y como se ha implementado en Python (más de esto en **6.2.3 *ec_pdf***), los dos primeros pasos del algoritmo 1 se han simplificado en una función que devuelve la función de densidad empírica condicionada. Se puede tomar el máximo de esta función como una predicción de y_k. Dicho de otro modo, la predicción $\hat{y}$ para x_k podría calcularse como la salida condicionada $\bar{y}_l$ con mayor probabilidad del set $\overline{\mathcal{Y}}$:

Partiendo de un set $D = \{[x_i, y_i]:\ i = 1, \dots, N\} \subset \mathcal{X} \times \mathcal{Y}$ con la relación $y_i = x_i^2$ donde $\mathcal{X}$ contiene un conjunto de valores entre 0 y 10. Se quiere estimar el valor y para $x = 6$. Para el set $\overline{\mathcal{Y}}$ se ha tomado $y_1 = 0, y_M = 100$ con $M = 500$.

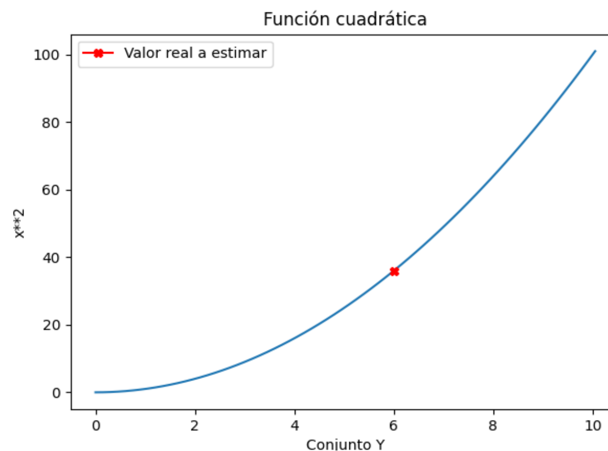

Figura 4-1. Función cuadrática por estimar

Haciendo uso de la primera parte del algoritmo se obtiene la función de densidad empírica condicionada. Como cabía esperar, el pico de la función está en torno a 36 siendo la marca roja donde debería encontrarse este máximo.

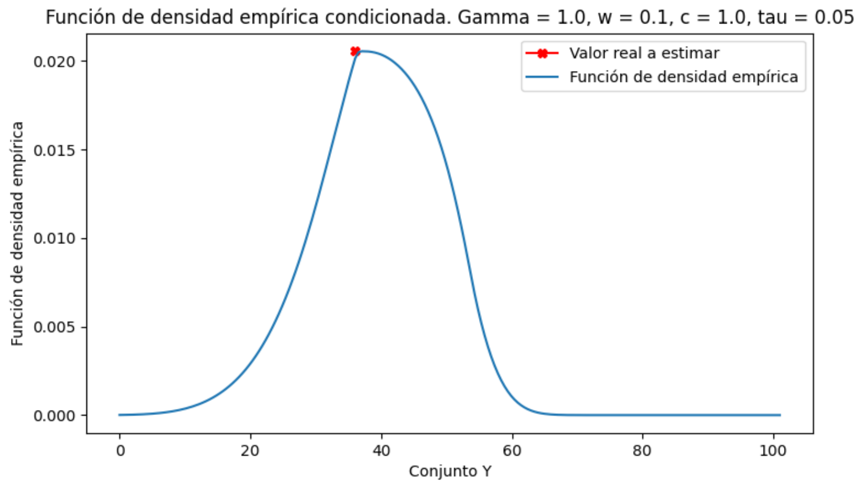

Figura 4-2. Función de densidad empírica condicionada

Siguiendo con la segunda parte del algoritmo, se obtiene el cuantil máximo y mínimo especificados por τ:

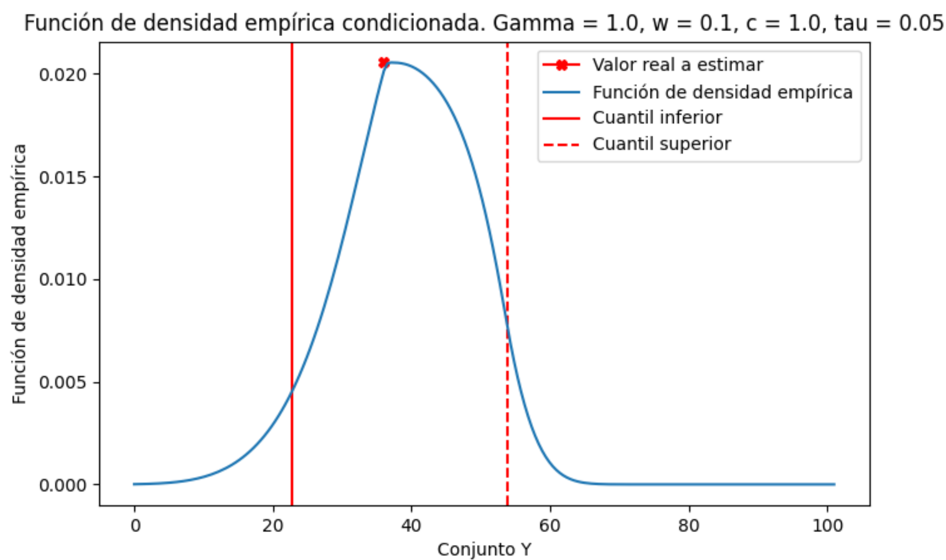

Figura 4-3. Predicción intervalar de x^2 para $x = 6$

Teniendo en cuenta la propiedad de las funciones de distribución de (4-4), si se hiciera el sumatorio del área bajo la curva desde $\bar{y}_{121} = 24{,}2$ hasta $\bar{y}_1 = 0$, es decir, un sumatorio a la inversa del usado para calcular el cuantil inferior, el resultado debería sería igual a τ. Es la probabilidad de que la salida y sea inferior a y_τ^-:

$$Prob_{\bar{y}|x_k}\{y \leq y_\tau^-\} = \sum_{j=1}^{l_\tau^-} p_j \geq \tau. \quad (4\text{-}8)$$

De la misma manera ocurrirá para el cuantil superior si se tomase el mayor valor de l que cumpliera:

$$Prob_{\bar{y}|x_k}\{y \geq y_\tau^+\} = \sum_{j=l_\tau^+}^{M} p_j \geq \tau.$$

Puesto que el rango del intervalo es considerable, se aumenta el valor de c disminuyendo el valor de la similitud obtenido con (4-6). Esto reduce a su vez el rango de los intervalos centrándolos en el valor que más

disminuye la función de disimilitud:

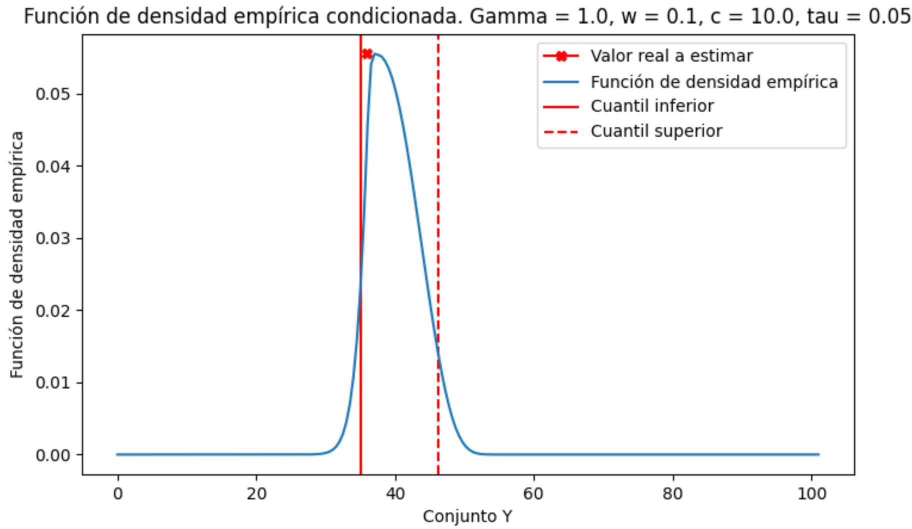

Figura 4-4. Predicción intervalar de x^2 para $x = 6$ con c aumentada

4.3.2 Implementación del Algoritmo 1 en Python

Como se indicaba al principio de este capítulo, se ha omitido en todo momento la opción de añadir información local a la función de coste de $J_\gamma(z, D)$. Esto ha sido así pues carecía de interés de cara a explicar la predicción intervalar. Entran en juego nuevamente como se verá en el paso III) del algoritmo 1. Cabe decir que se ha usado la función de disimilitud descrita en **2.2.3 Función de disimilitud para matriz de pesos**. Es necesario añadir que la obtención de $\overline{\mathcal{Y}}$ usando (4-2) se ha resumido en una función aparte: $Set_{\overline{\mathcal{Y}}}(D)$.

Antes de mostrar los diecisiete pasos usados para el Algoritmo 1, se deben recordar algunas nociones del enfoque dual y las restricciones de igualdad utilizados en **3.2 Restricciones de igualdad y enfoque dual**. En esta sección, se aislaba la variable de decisión y de las restricciones de igualdad para después estimarla obteniendo $\hat{y}$ de la combinación lineal:

$$\hat{y} = \sum_{i=1}^{N} \lambda_i^* y_i.$$

Para el caso del algoritmo 1 no se desacoplará esta restricción de igualdad puesto que no se quiere calcular $\hat{y}$. Se quiere calcular la disimilitud que tendrá cada uno de los valores de $\overline{\mathcal{Y}}$ para una entrada x_k y el conjunto $D \in \mathbb{R}^{\mathrm{N}}$. Es decir, habrá que resolver los M problemas de optimización siguientes:

$$\begin{aligned} J_{\gamma,j}\left(\begin{bmatrix}\bar{y}_j \\ x_k\end{bmatrix}, D\right) = \min_{\lambda_1,\ldots,\lambda_N} & \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i| \\ s.t. \quad & \begin{bmatrix}\bar{y}_j \\ x_k\end{bmatrix} = \sum_{i=1}^{N} \lambda_i \begin{bmatrix}y_i \\ x_i\end{bmatrix} \\ & 1 = \sum_{i=1}^{N} \lambda_i \, . \end{aligned} \quad (4\text{-}9)$$

Nuevamente, se "unen" las tres restricciones[3] de igualdad:

[3] Puesto que y_i es de dimensión 1, la variable dual solo verá aumentada su dimensión en 1 al igual que n.

$$R_{yx}\lambda = r_{yx} : R_{yx} = \begin{bmatrix} y_1 & y_2 \dots y_N \\ x_1 & x_2 \dots x_N \\ 1 & 1 \dots 1 \end{bmatrix} \in \mathbb{R}^{(n+2)xN}, \qquad \lambda = [\lambda_1 \ \lambda_2 \ \dots \ \lambda_N]^T \in \mathbb{R}^N, \qquad r_{yx} = \begin{bmatrix} \bar{y}_j \\ x_k \\ 1 \end{bmatrix} \in \mathbb{R}^{n+2}.$$

$$J^*_{\gamma,j} = \min_{\lambda \in \mathbb{R}} \lambda^T H_w \lambda + \|\Gamma_w \lambda\|_1 \quad \text{s.t.} \ R_{yx}\lambda = r_{yx}. \tag{4-10}$$

De cara a la implementación del Algoritmo 1 se debe recordar que los parámetros α_h y α_γ caracterizarán los pesos de la diagonal de las matrices H_w y Γ_w respectivamente usando $P(r, D, \alpha)$ como la función de disimilitud definida en (2-4) y que $x_k \in \mathbb{R}^n$, será la entrada que condiciona la salida y_k.

<u>*Algoritmo* 1</u>

Se requiere: $c \geq 0,\ \gamma \geq 0,\ x_k \in \mathbb{R}^n,\ D \in \mathbb{R}^{(n+1)xN}, \alpha_h \geq 0,\ \alpha_\gamma \geq 0,\ x_{ini}^{FISTA} \in \mathbb{R}^{n+2},\ \tau \in (0,1),\ M.$

Se obtiene: $[y_\tau^-, y_\tau^+]$.

I) $\bar{\mathcal{Y}} = Set_{\bar{\mathcal{Y}}}(D, M),\ j = 1,\ R_{yx} = \begin{bmatrix} y_1 & y_2 \dots y_N \\ x_1 & x_2 \dots x_N \\ 1 & 1 \dots 1 \end{bmatrix}, Prob = 0,\ l = 0.$

II) **while** $j \leq M$ **do**

III) $r_{yx} = \begin{bmatrix} \bar{y}_j \\ x_k \\ 1 \end{bmatrix}.$

IV) $H_w = P(r_{yx}, D, \alpha_h);\ \Gamma_w = \gamma \cdot P(r_{yx}, D, \alpha_\gamma).$

V) $[\lambda^*,\ x_{ini}] = FISTA(R_{yx},\ r_{yx},\ H_w,\ \Gamma_w,\ x_{ini}).$

VI) $J_{dj} = \lambda^{*T} H_w \lambda^* + \|\gamma_w \lambda^*\|_1.$

VII) $s_j = \exp(-cJ_{dj}), \quad j = j + 1.$

VIII) **Fin**

IX) $factor\ de\ normalización = \sum_{j=1}^{M} s_j.$

X) **while** $Prob < 1 - \tau$ **do**

XI) $Prob = Prob + \frac{s_l}{factor\ de\ normalización}, \quad l = l + 1.$

XII) **Fin**

XIII) $l_\tau^+ = l,\ \ l = 0,\ Prob = 0.$

XIV) **while** $Prob < 1 - \tau$ **do**

XV) $Prob = Prob + \frac{s_{M-l}}{factor\ de\ normalización}, \quad l = l + 1.$

XVI) **Fin while**, $l_\tau^- = l$

XVII) $[y_\tau^-, y_\tau^+] = [\bar{y}_{l_\tau^-},\ \bar{y}_{l_\tau^+}].$

4.3.3 Eficiencia del Algoritmo 1

Los pasos X) – XII) y XIV) - XVI) se han programado de manera ligeramente distinta para mejora del coste computacional. Estos cambios no se han añadido directamente para una mejor comprensión conceptual del algoritmo.

Por cada iteración de XI) y de XV) se realiza una división por el factor de normalización, por lo que al final del algoritmo se han realizado un poco menos de $2 \cdot M$ divisiones. En vez de esto, se guarda en una variable lim la restricción de desigualdad $(1 - \tau) \cdot factor\ de\ normalización$ de manera que en los pasos

XI) y XV) pasa a sumarse únicamente s_l y s_{M-l} respectivamente. Por consiguiente, en X) y XIV) se compara con el límite guardado en lim. Para XI) - XV):

$$Prob = Prob + s_l.$$

Si no se cumple: $Prob > (1-\tau) * factor\ de\ normalización$ → **exit.**

De esta forma se han ahorrado en torno a $2 \cdot M$ divisiones innecesarias por cada vez que se usase el Algoritmo 1. Puede parecer una nimiedad. No obstante, como se verá en el Algoritmo 2, la cantidad de veces que es necesario hacer uso del Algoritmo 1 invita a mirar con lupa cada línea de código.

Al describir como se implementaría el método FISTA en **3.4.1 Implementación de FISTA en Código Python** se remarcó la importancia de devolver la variable dual β obtenida para la λ^* solución. Esto es así por la naturaleza del Algoritmo 1 donde para una sola predicción intervalar se resuelven M problemas de optimización (se recorre todo el set discreto de $\overline{\mathcal{Y}}$). Cada uno de los problemas de optimización será muy parecido al anterior, por lo que tiene sentido aprovechar este conocimiento. Como ya se ha remarcado, el algoritmo FISTA será utilizado hasta la saciedad[4].

Dado $\bar{y}_l : l = 1, \dots, M$, la diferencia entre el problema de optimización de l y $l+1$ es la siguiente:

$$J_\gamma\left(\begin{bmatrix}\bar{y}_{l+1}\\ x_k\end{bmatrix}, D\right) = \min_{\lambda_1,\dots,\lambda_N} \sum_{i=1}^{N} w_i^h \lambda_i^2 + \sum_{i=1}^{N} w_i^\gamma |\lambda_i|$$

$$s.t. \quad \begin{bmatrix}\bar{y}_{l+1}\\ x_k\end{bmatrix} = \sum_{i=1}^{N} \lambda_i \begin{bmatrix}y_i\\ x_i\end{bmatrix} \qquad (4\text{-}11)$$

$$1 = \sum_{i=1}^{N} \lambda_i.$$

Es decir, un pequeño cambio en una de las restricciones de igualdad. Sabiendo que por construcción de $\overline{\mathcal{Y}}$ el incremento entre y_l y y_{l+1} es pequeño $\left(\frac{\bar{y}_M - \bar{y}_1}{M-1}\right)$ puesto que se trata de asemejar al incremento de un diferencial, se puede asumir con confianza que la solución óptima para λ_l^* será parecida a λ_{l+1}^*.

A continuación, se muestra la evolución de variable dual para cada uno de los M problemas de optimización de la división del set $\overline{\mathcal{Y}}$. Es decir, la variable dual solución para cada $\bar{y}_l$. Se comprueba el supuesto de que generalmente la solución de la variable dual cambia muy poco de $\bar{y}_l$ a $\bar{y}_{l+1}$. Para un problema de optimización de 200 puntos donde la variable dual tendrá dimensión $\mathbb{R}^3$ y tomando $M = 500$:

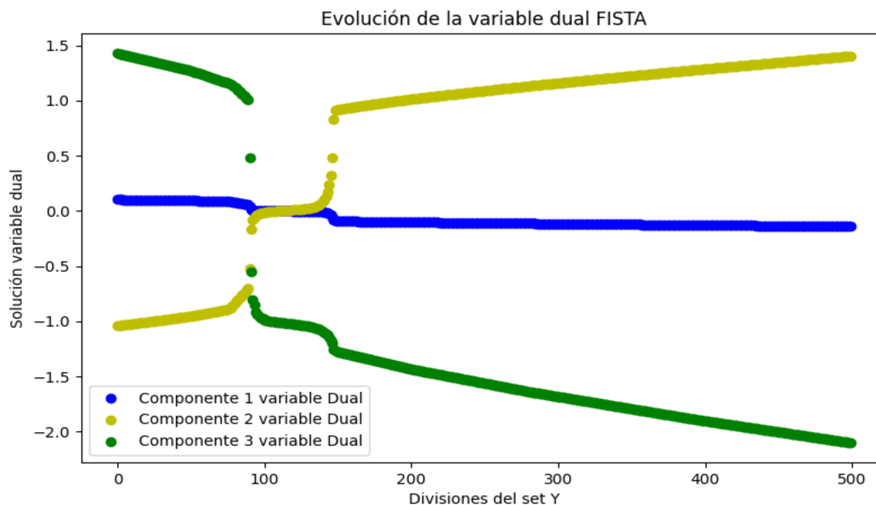

Figura 4-5. Evolución de la variable dual

[4] En el Algoritmo 2 será necesario calcular V·R + D predicciones intervalares donde V = dimensión set de validación, R = repeticiones hasta obtener c_{opt} y D = dimensión del set de datos del que se quiere obtener las predicciones intervalares. Es decir, $N_{FISTA} \cong (V \cdot R + D) \cdot M$. Demasiado como para no revisar con detalle el algoritmo FISTA.

En general, la evolución de la variable dual es casi recta y cambia suavemente salvo por un intervalo en 80-150 donde dos de sus componentes varían drásticamente. Una forma de aprovechar que la variable dual cambia suavemente, en muchas ocasiones, es utilizar la variable óptima anterior como condición inicial del siguiente problema de optimización a resolver. Aplicando lo dicho, el número de iteraciones de FISTA necesarias para alcanzar cada una de estas soluciones se reduce sensiblemente salvo en el tramo 80-150 donde el algoritmo no se puede apoyar en la condición inicial anterior. En la gráfica de la Figura 4-6 se muestra lo que se acaba de explicar. Donde la variable dual de y_{l-1} no se parece a la dual de y_l se necesitan más iteraciones. Queda reflejado el interés de la variable dual anterior para la dual inicial del siguiente caso.

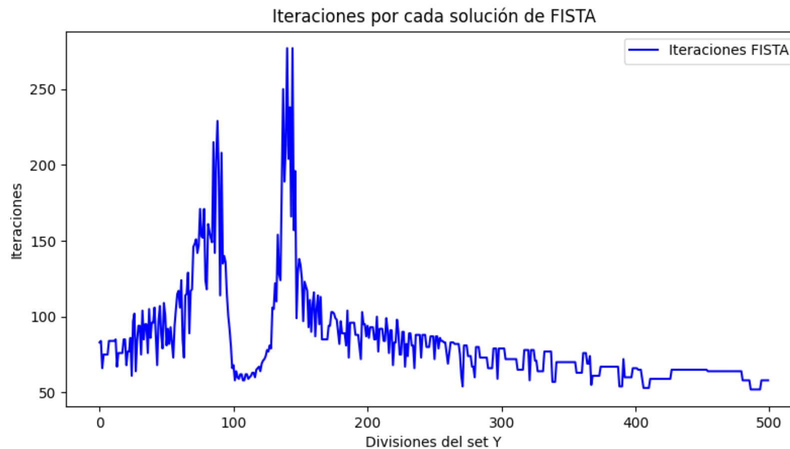

Figura 4-6. Iteraciones FISTA para cada una de las soluciones de y_l

Finalmente, se presentan unas gráficas comparativas para la resolución de una misma predicción intervalar que muestran la diferencia de iteraciones totales necesarias con la ayuda de la variable dual anterior y sin ella. La predicción intervalar es:

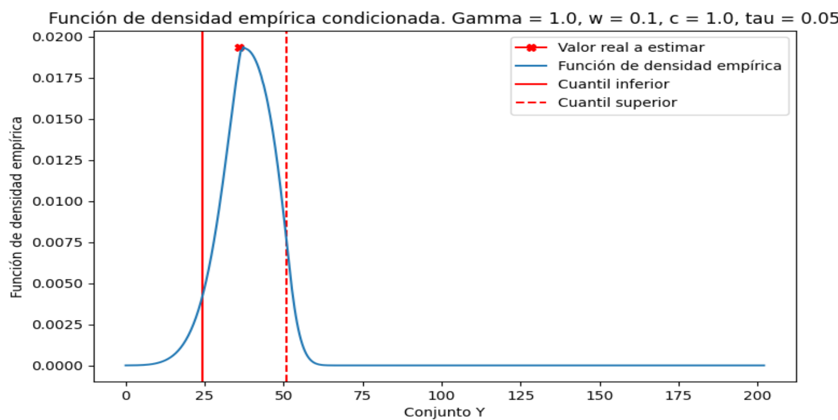

Figura 4-7. Predicción Intervalar

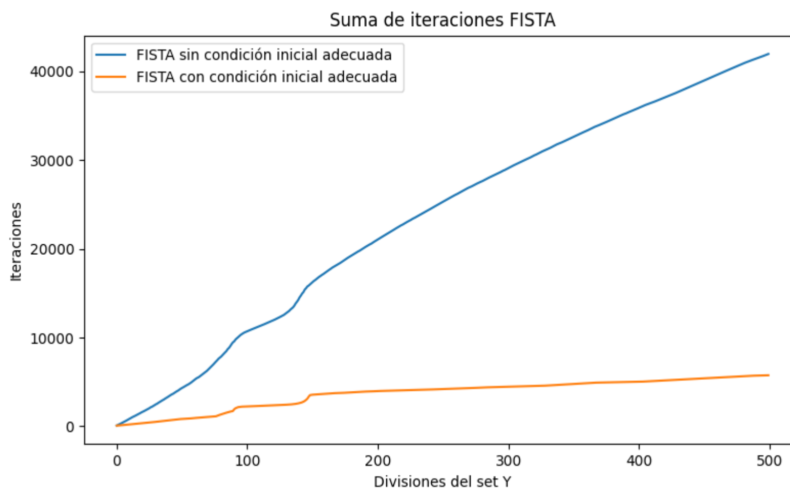

Figura 4-8. FISTA para distintas condiciones iniciales.

4.4 Algoritmo 2. Valor óptimo de *c*

La predicción intervalar obtenida gracias al Algoritmo 1 depende de los parámetros γ y c. Como se observó en la Figura 4-3 y en la Figura 4-4, mayores valores de c ajustan y reducen el intervalo mientras que pequeños valores de c expanden dicho intervalo. Esto es debido a la relación entre las funciones de similitud y disimilitud definida en (2-1). Grandes valores de c reducen el intervalo centrado en el punto de mayor similitud. Pequeños valores de c devuelven una distribución plana en la que cada elemento de $\overline{\mathcal{Y}}$ tendrá una probabilidad parecida que tiende a $\frac{1}{M}$ para $c \to 0$.

Por un lado, al aumentar c, se consiguen intervalos más pequeños. No obstante, esto provocará, como se verá más adelante, que una mayor parte de las salidas no estén contenidas en el intervalo. Por otro lado, disminuir c y asegurar que más salidas estén contenidas en $[y_\tau^-, y_\tau^+]$ implica intervalos más grandes e inespecíficos, lo que carece de utilidad. En concreto, el Algoritmo 2 se centrará en, dada una γ, obtener el mayor valor de c_γ que asegure que el intervalo contiene las salidas con una fracción de violaciones del intervalo inferior a la probabilidad especificada.

4.4.1 Justificación del Algoritmo 2

A continuación, se mostrarán varios ejemplos en los que, sin un conocimiento previo del problema, no se puede proponer un valor de c adecuado. Esta fuera de discusión la posibilidad de sintonizar manualmente el parámetro c pues habría que contar el número de violaciones (salidas y_k que se salen del intervalo) y disminuir o aumentar aleatoriamente este valor de c.

Para el siguiente ejemplo se ha tomado una función ascendente simple con dos escalares aleatorios de varianza 2 y 3. Cabe decir que el set de datos D no contiene al set de validación V: $D \cap V = \emptyset$

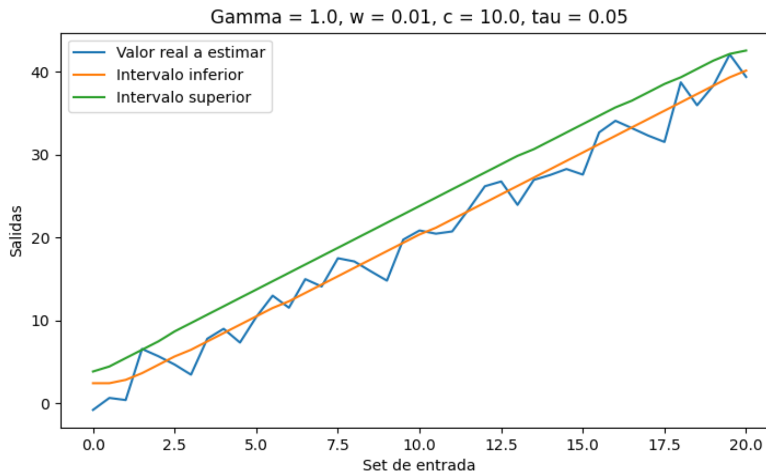

Figura 4-9. Predicción intervalar con c exagerada

Como cabría esperar, grandes valores de c reducen el intervalo a costa de que una gran fracción de las salidas no estén contenidas en el mismo, lo que hace la predicción inservible.

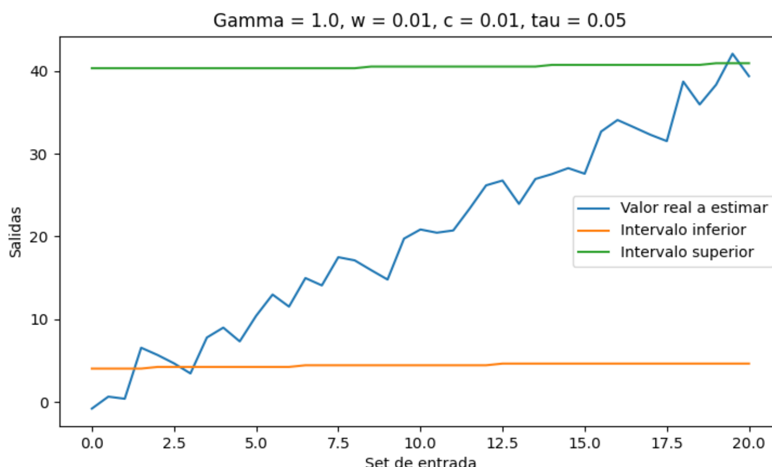

Figura 4-10. Predicción intervalar con c muy disminuida

Valores muy pequeños de c dan una predicción intervalar plana en la que todo punto tiene prácticamente la misma probabilidad. Los intervalos resultan ser el máximo y el mínimo de $\overline{\mathcal{Y}}$. No hace falta hacer un Trabajo de Fin de Grado de predicciones intervalares para dar esta estimación.

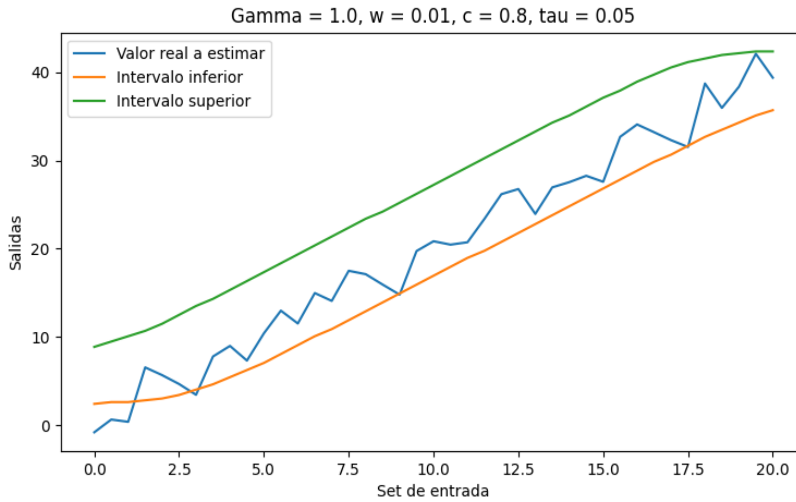

Figura 4-11. Predicción intervalar con c plausible

Probando un valor de c más moderado se consigue un intervalo más ajustado que en la Figura 4-10 y sin ser inservible como la Figura 4-9.

En definitiva, se puede apreciar como el valor óptimo de c_γ no es trivial ni se puede calcular a priori. Por esto se ve necesario un algoritmo que permita obtener c óptimo para una γ y τ dados.

4.4.2 Explicación del Algoritmo 2

Sin más dilación y teniendo en cuenta todo lo necesario para el Algoritmo 1, el Algoritmo 2 que se plantea trata de encontrar el máximo valor de c que garantice que las salidas estén contenidas en los intervalos obtenidos de forma que la fracción de violaciones en el conjunto de validación corresponda a la especificación probabilística. El set de validación del Algoritmo 2 será:

$$V = \left\{ \begin{bmatrix} \tilde{y}_s \\ \tilde{x}_s \end{bmatrix}, s = 1, \dots, N_v \right\},$$

Donde, haciendo uso del Algoritmo 1, se obtienen las predicciones intervalares de $\tilde{y}_s$: $s = 1, \dots, N_v$ para una γ, τ, c y $\tilde{x}_s$ dadas. De esta forma, se calculan N_v intervalos. Para saber si la c dada es demasiado grande se cuenta el número de veces que la salida $\tilde{y}_s$ no estaba contenida en su intervalo I_s. El número de violaciones se contará para cada cuantil (superior e inferior). Así pues, si el número de violaciones es superior al número total de salidas $\tilde{y}_s$ multiplicado por el cuantil τ que cumple la especificación probabilística, significará que hay demasiadas $\tilde{y}_s$ que no están contenidas en I_s. El intervalo I_s será desechado junto a la c usada:

$$\textbf{Si se cumple: } \frac{n_{vio}^{+}}{N_v} > \tau \text{ , } \frac{n_{vio}^{-}}{N_v} > \tau \rightarrow \textbf{disminuir } c.$$

De esta manera se comprueba si es necesario disminuir o aumentar c en la siguiente iteración del algoritmo. Si el número de violaciones es inferior al límite, entonces se aumentará c. En caso contrario, se deberá disminuir.

Para esta tarea, hace falta una c_{max} y una c_{min} de partida. Como es lógico, c_{min} será igual a 0. En las iteraciones en las que no se supere el límite de violaciones se tomará la c usada como una c válida guardándose en c_{min}. En las iteraciones en las que se supere el límite de violaciones, se descartará la c usada guardándose en c_{max}, para probar de nuevo con una c entre c_{max} y c_{min}. Puesto que en algún momento habrá que finalizar el algoritmo, se define una precisión ε para la cual si se deja de cumplir $c_{max} - c_{min} > \varepsilon$, se dé por terminada a búsqueda de c_γ óptimo.

4.4.3 Implementación del Algoritmo 2 en Python

Se debe tener en cuenta lo siguiente del Algoritmo 1 :

$$\gamma \geq 0,\ D \in \mathbb{R}^{(n+1)xN},\ \alpha_h \geq 0,\ \alpha_\gamma \geq 0,\ x_{ini}^{FISTA} \in \mathbb{R}^{n+2},\ \tau \in (0,1),\ M.$$

<u>*Algoritmo 2*</u>

Se requiere: lo necesario para Algoritmo 1 junto a $c_{max} \geq 0,\ \ \varepsilon > 0,\ V = \left\{\begin{bmatrix}\tilde{y}_s \\ \tilde{x}_s\end{bmatrix}, s = 1, \ldots, N_v\right\} \subset \mathcal{X} \times \mathcal{Y}$.

Se obtiene: $\left[c_\gamma^*,\ I_\tau\right] \to I_\tau = \left\{\begin{bmatrix}I_s^- \\ I_s^+\end{bmatrix}, s = 1, \ldots, N_v\right\}$.

I) $c_{min} = 0$.

II) **while** $c_{max} - c_{min} \geq \varepsilon$ **do**

III) $c = 0.5 \cdot (c_{max} + c_{min})$.

IV) $n_{vio}^+ = 0,\ n_{vio}^- = 0,\ s = 0$.

V) **while** $s < N_v$ **do**

VI) $[I_s^-,\ I_s^+] = Algoritmo1(D,\ \tilde{x}_s,\ \gamma,\ c,\ x_{ini}^{FISTA},\ \tau,\ \alpha_h,\ \alpha_\gamma,\ M)$.

VII) **Si se cumple**: $I_s^- > \tilde{y}_s \to n_{vio}^- = n_{vio}^- + 1$.

VIII) **Si se cumple**: $I_s^+ > \tilde{y}_s \to n_{vio}^+ = n_{vio}^+ + 1$.

IX) **Si se cumple**: $\frac{n_{vio}^+}{N_v} \geq \tau\ ,\ \frac{n_{vio}^-}{N_v} \geq \tau \to c_{max} = c,\ break$.

X) $s = s + 1$.

XI) **Fin while**

XII) **Si se cumple** $\frac{n_{vio}^+}{N_v} < \tau\ ,\ \frac{n_{vio}^-}{N_v} < \tau \to c_{min} = c$.

XIII) **Fin while**

XIV) **Si se cumple** $c == c_{max}$ → No se puede conseguir una c_γ^* para estos $\varepsilon, \gamma, \alpha_\gamma, \alpha_h$ dados.

XV) **Si se cumple** $c == c_{min} \to\ c_\gamma^* = c,\ I_\tau = [I^-, I^+]$.

Es interesante señalar un cambio respecto al algoritmo propuesto en [1] donde se calculan todos los intervalos ($[I_s^-, I_s^+], s : 1, \ldots, N_v$) de una sola vez dada una c. Tras esto, se comprobaba el número de violaciones para cada $\tilde{y}_s$ con su intervalo $[I_s^-, I_s^+]$. Sin embargo, es mucho más eficiente calcular uno a uno el intervalo $I_s = [I_s^-, I_s^+]$ y comprobar al mismo tiempo si $\tilde{y}_s$ no está dentro del mismo. De esta forma, si antes de recorrer todo el set de validación se ha superado el límite de violaciones, no hará falta calcular los intervalos restantes con el Algoritmo 1.

Dado un set de validación con $N_v = 20$, y suponiendo que tras calcular el intervalo en $s = 6$ se ha superado el límite de violaciones, se han ahorrado $N_v - s = 14$ cálculos de intervalos. Si además se añade que se han necesitado 10 aproximaciones para obtener c_γ^* el Coste Computacional Reducido sería:

$$CCR = \sum_{aprox=1}^{10} N_v - s^{aprox}, \tag{4-12}$$

donde N_v será siempre el máximo número de intervalos posibles (número de salidas del set de Validación) y s^{aprox} es el índice cuando el bucle V) finaliza. Es decir, s^{aprox} es el número de intervalos calculados para esa aproximación. En resumen, CCR será el número de veces que no ha hecho falta llamar al Algoritmo 1 que, como ya se vio, tiene un coste computacional considerable.

4.5 Alcance del Fin Último

Llegado a este nivel, una vez expuestos los dos pilares fundamentales del TFG y planteados los dos algoritmos que se apoyan en dichos pilares, se alcanza el fin último consistente en proveer predicciones intervalares útiles aplicadas a casos reales.

4.5.1 Predicción intervalar de la demanda eléctrica española

En esta subsección se muestran ejemplos de predicciones intervalares para la demanda eléctrica española. A su vez, se muestran los resultados de la predicción de la media condicionada $\hat{y}_m$ aproximada como el centro del intervalo $[y_{0.5}^{-}, y_{0.5}^{+}]$.

En primer lugar, se establecen los parámetros del Algoritmo 1 y 2 que serán fijos para todos los ejemplos. Para ello, es lógico reutilizar los conceptos descritos en **2.3.1 Demanda eléctrica española.**

Se vuelve a utilizar el concepto de base de datos dinámica definiendo el mismo set de datos D que en (2-8). No obstante, el tamaño de la base de datos $BD = x_i : i = 1, \dots, dbd$ será de $dbd = 90$ en lugar de ciento veinte días.

Para empezar, se debe encontrar el c_{opt} por lo que se hace uso del Algoritmo 2 con un set de validación de $dbd \cdot 2$ días y con la siguiente configuración general:

Precisión $\varepsilon = 0.01$ - divisiones del set $\overline{\mathcal{Y}}$: $M = 300$ - cuantil inferior $\tau = 0.05$ y cuantil superior $\tau = 0.95$ - tamaño del set de validación $N_v = dbd \cdot 2 - 14$.

Se define una base de datos específica para el set de datos de validación: $BD_v = x_i : 1, \dots, dbd \cdot 2$

$$V = \left\{ \begin{bmatrix} \widetilde{y_s} \\ \widetilde{X_s} \end{bmatrix}, s = 1, \dots, N_v \right\},$$

donde $\widetilde{y_s}$ se define como la demanda dentro de 7 días para la entrada $\widetilde{X_s}$:

$$\widetilde{y_s} = \{x_{s+14} : s = 1, \dots, N_v\},$$

donde $\widetilde{X_s}$ se define con la misma estructura de datos que el regresor que se presenta en **2.3.1 Demanda eléctrica española**:

$$\widetilde{X_s} = \left\{ \begin{bmatrix} x_{7+s} \\ x_{6+s} \\ x_s \\ ds_s \end{bmatrix} : s = 1, \dots, N_v \right\}.$$

Tras usar el Algoritmo 2 y obtener c_{opt} para los cuantiles dados, se emplea al Algoritmo 1 con esta c_{opt} para predecir los 2049[5] intervalos de los 5 años de demanda. Como se ve, es importante que el set de validación tenga un número representativo de datos pues la c_{opt} que se obtiene del Algoritmo 2 tendrá que ser adecuada para el resto de los días.

Después de esta introducción sobre los pasos que se han seguido, se muestran los distintos casos en los que se trata de plasmar como la variación de los parámetros c_{max}, γ, α_h y α_γ está relacionada con los resultados.

Después de los ejemplos, se indicará un criterio para encontrar el parámetro γ óptimo. Este consiste en minimizar una función de coste Q_γ que penalice la longitud media del intervalo y el error de la predicción respecto a la media tal como se especifica [1].

[5] $DiasTotales - dbd = 2139 - 90 = 2049\ días$

4.5.1.1 Predicción intervalar sin información local en la función de coste

A continuación, se muestran los ejemplos de predicción intervalar con $\gamma = 1$ y $\alpha_h = 0$, $\alpha_\gamma = 0$ de tal forma que el funcional no tendrá información local sobre la demanda pasada. Se verá que, en general, la inclusión de pesos es aconsejable de cara a unos buenos resultados y aún más siendo una serie temporal tan dependiente de la demanda de días pasados.

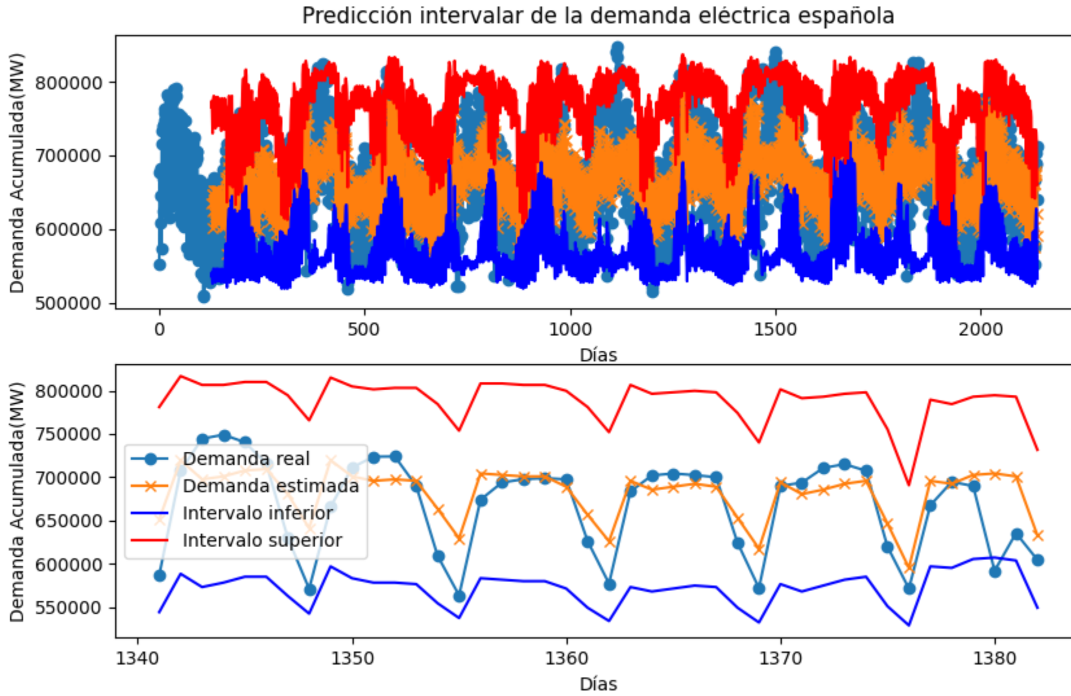

Figura 4-12. Predicción intervalar para $\gamma = 1,\ \alpha_h = 0, \alpha_\gamma = 0\ , c = 1.797$

Como se aprecia en Figura 4-12 y en la Figura 4-13, la predicción intervalar es excesivamente conservadora y provee de unos intervalos demasiado amplios como para ser útiles. En algunos tramos es prácticamente el máximo y el mínimo de $\overline{\mathcal{Y}}$ como intervalo superior e inferior respectivamente.

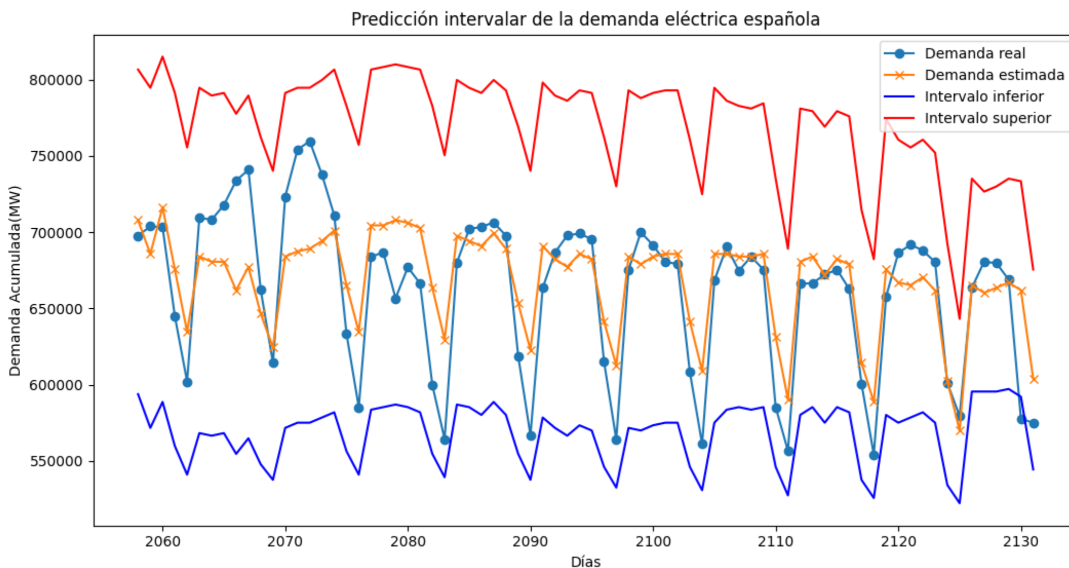

Figura 4-13. Predicción intervalar para $\gamma = 1,\ \alpha_h = 0,\ \alpha_\gamma = 0,\ c = 1.797$

A continuación, se prueba el impacto que tiene en los resultados disminuir γ sin ponderación ya que existe la posibilidad de que la predicción anterior no fuera aceptable por un valor de c muy reducido.

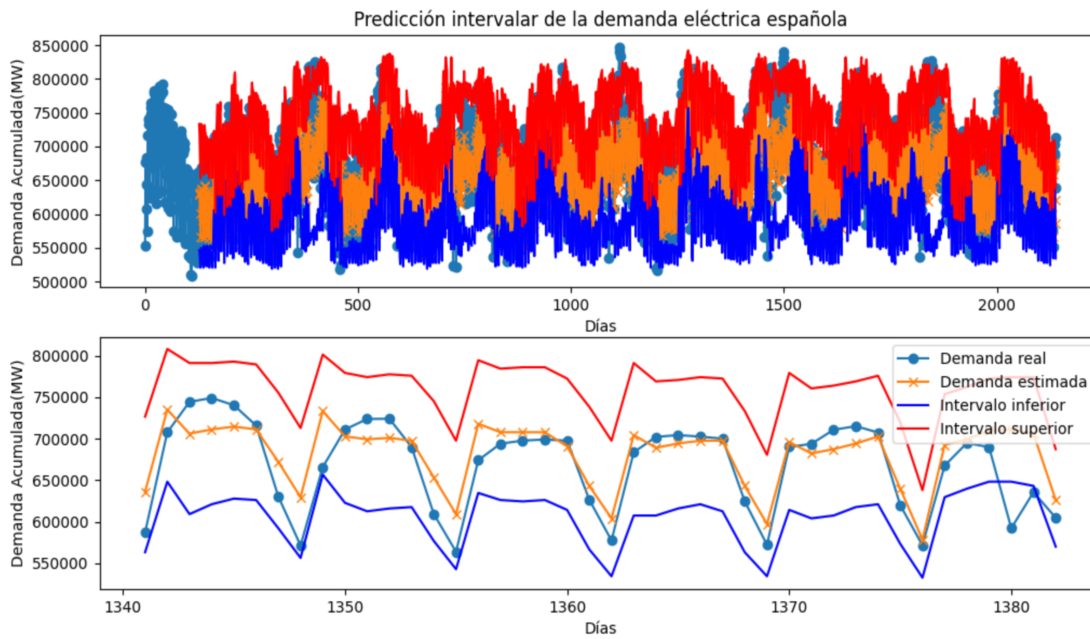

Figura 4-14. Predicción intervalar para $\gamma = 0.1,\ \alpha_h = 0,\ \alpha_\gamma = 0,\ c = 7.58$

Gracias a reducir γ, la función de distribución condicionada de cada intervalo se ajusta mejor a la caracterización del set de datos D ya que se consigue un mayor valor de c. Es decir, el equilibrio encontrado entre c y γ mejora los resultados como se verá en la tabla de **4.5.1.3 Tabla de resutados.**

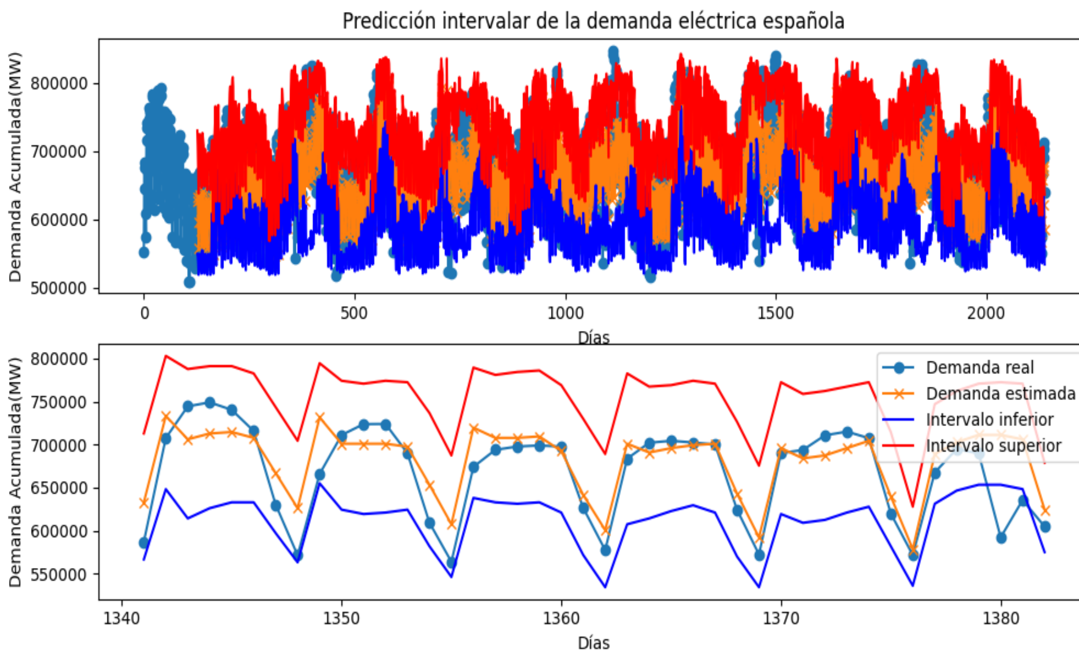

Figura 4-15. Predicción intervalar para $\gamma = 3,\ \alpha_h = 0,\ \alpha_\gamma = 0,\ c = 0.625$

Al incrementar γ a tres, los intervalos se ajustan mejor al comportamiento de la demanda (que con $\gamma = 1$) pese a que el valor de c se reduce debido a una mayor tasa de violaciones. En resumen, a pesar de reducir γ hasta 0.1, se consigue un mayor valor de c que permite ajustar mejor los intervalos. Al incrementar γ hasta 3 se llega prácticamente a la misma solución debido al equilibrio que debe haber entre γ y c.

Parece que, en general, no introducir información local al algoritmo tiene un límite claro de mejoría y más aún cuando solo hay un parámetro a sintonizar (realmente c se escoge como el óptimo para un γ dado).

4.5.1.2 Predicción intervalar con información local en la función de coste

Usando pesos que penalicen los datos pasados que no son similares se consigue que la predicción intervalar se ajuste mejor en general. En concreto los intervalos inferiores.

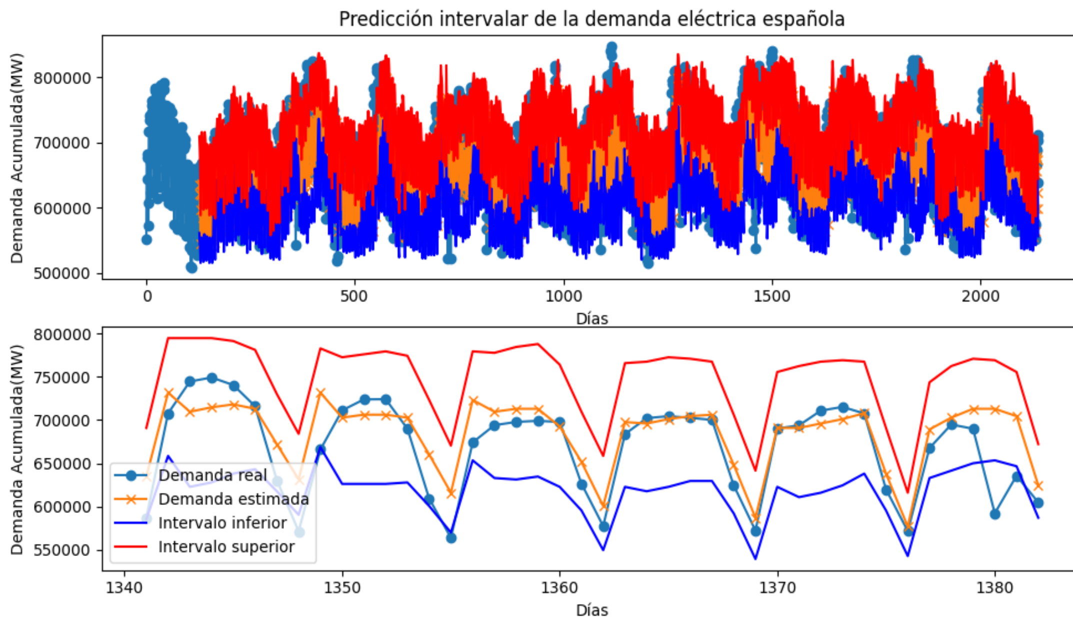

Figura 4-16. Predicción intervalar para $\gamma = 1,\ \alpha_h = 1.4,\ \alpha_\gamma = 1.4,\ c = 0.875$

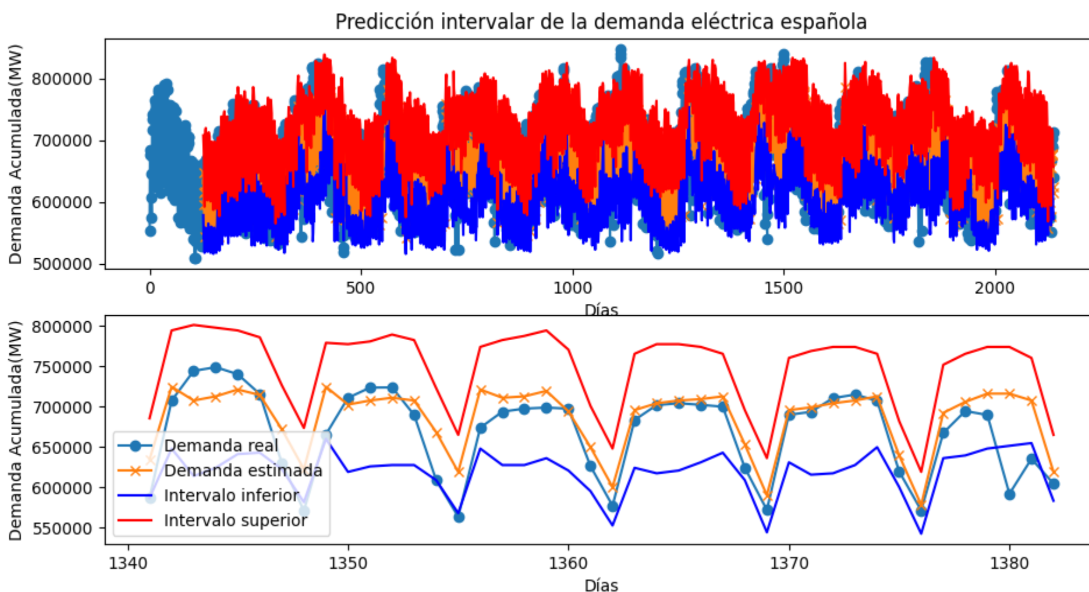

Figura 4-17. Predicción intervalar para $\gamma = 3,\ \alpha_h = 3,\ \alpha_\gamma = 3,\ c = 0.1875$

Al igual que en los casos anteriores, aumentar la γ junto a α_h y α_γ puede llevar a una predicción excesivamente agresiva que obligue al algoritmo 2 a reducir enormemente el valor de c. A efectos prácticos, elevar excesivamente estos parámetros es contraproducente reflejándose en que, para cumplir la restricción de los dos cuantiles, se debe "aplanar" la función de distribución de cada punto. Es mejor encontrar el equilibrio óptimo entre c y α_h , α_γ de modo y forma que c tenga un valor más elevado.

Para encontrar este equilibrio se ha planteado una función de coste que penalice la longitud del intervalo y el error de predicción. Esta función se define en **4.5.2 Función de coste Q_γ**. A continuación, se muestra la gráfica con los valores óptimos de $\gamma, \alpha_\gamma, \alpha_h$ referidos a los intervalos [0, 0.3], [0, 0.5] y [0, 0.5] respectivamente.

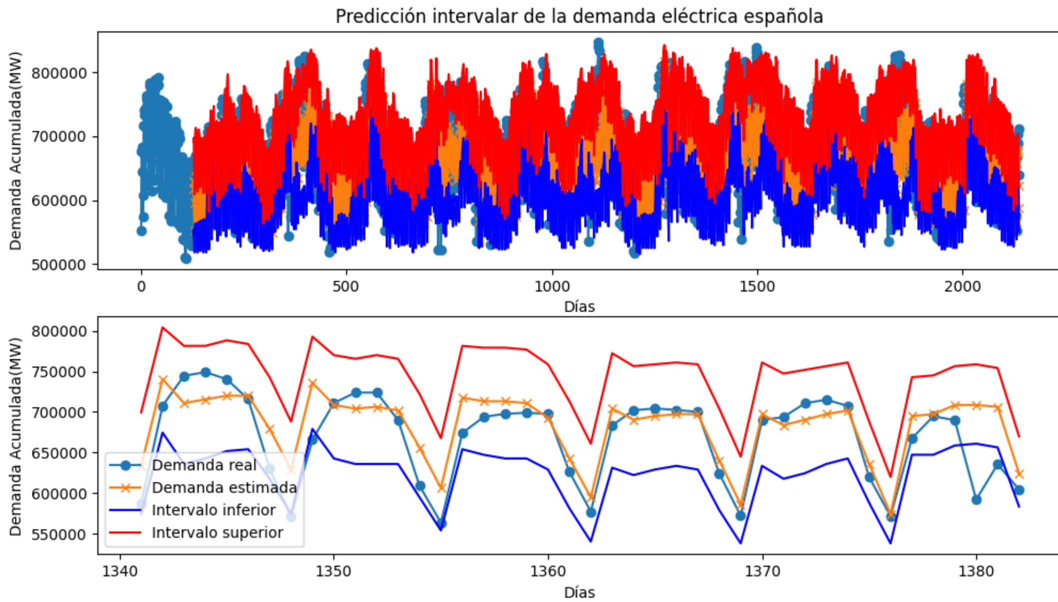

Figura 4-18. Predicción intervalar para $\gamma = 0.2,\ \alpha_h = 0.5,\ \alpha_\gamma = 0.23,\ c = 4.76$

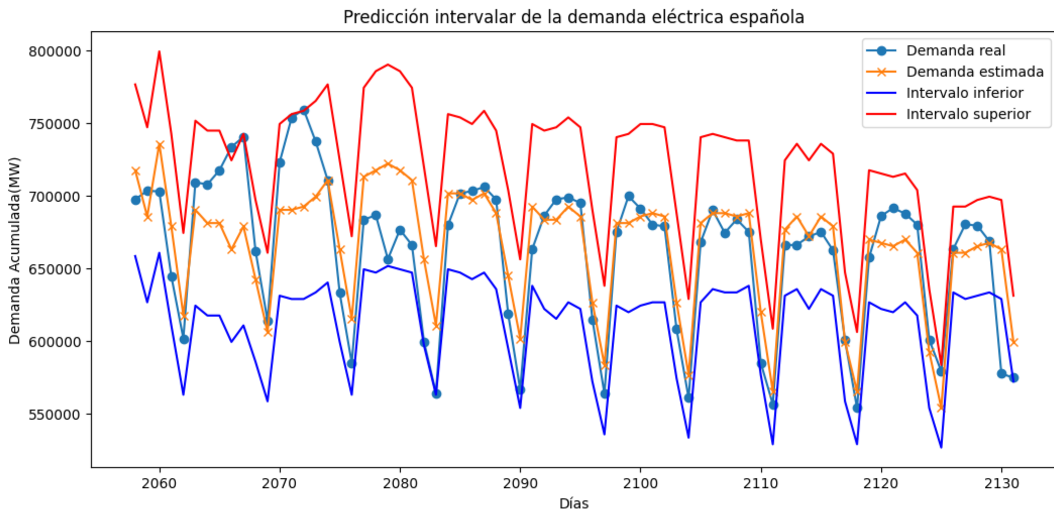

Figura 4-19. Predicción intervalar para $\gamma = 0.2,\ \alpha_h = 0.5,\ \alpha_\gamma = 0.23,\ c = 4.76$

Esta predicción es la mejor de todas las realizadas ya que obtiene los mejores resultados de error relativo y de longitud media del intervalo. Esto es a gracias la función de coste Q_γ.

4.5.1.3 Tabla de resutados

En la siguiente tabla se muestran los resultados más significativos de las 6 predicciones intervalares anteriores. En ella se puede observar la relación entre los parámetros $\gamma, \alpha_\gamma, \alpha_h, \tau$ y el comportamiento de la predicción intervalar. Los resultados que se han considerado son:

- c: el valor óptimo de c para una γ dada.
- ERM: el error relativo medio.
- EAM: el error absoluto medio.
- LMI: longitud media de intervalo.
- σ: desviación estándar.

Tabla 3. Resultados predicciones intervalares de la demanda eléctrica

γ	α_h	α_γ	τ	c	ERM	EAM	LMI	σ
0.1	0	0	0.1	7.58	5.05%	34372	57124	46456
1	0	0	0.1	1.79	5.27%	36055	63271	47889
3	0	0	0.1	0.703	5.38%	36870	66329	48668
1	1.4	1.4	0.1	0.859	5.28%	36205	60517	47857
3	3	3	0.1	0.234	5.44%	37155	54559	49185
0.2	0.5	0.23	0.1	4.76	5.04%	34420	56342	46327

4.5.2 Función de coste Q_γ

Como se ha podido observar en los ejemplos anteriores, sintonizar tantos parámetros y esperar encontrar la combinación óptima es una tarea entre ardua e imposible. Por eso, en esta última subsección, se plantea una función de coste Q_γ que penaliza tanto la longitud media del intervalo como la disparidad entre el valor real y_k con su predicción $\tilde{y}_s$. Así, para tres vectores de posibles valores de $\alpha_\gamma \in A_\gamma, \alpha_h \in \mathrm{A_h}, \gamma \in \Gamma$ se puede encontrar la combinación óptima que minimice Q_γ. Esto es:

$$\gamma_\tau^*, \alpha_\gamma^*, \alpha_h^* \rightarrow \arg \min_{\alpha_\gamma, \alpha_h, \gamma} \sum_{s=1}^{N_v} \left(\left(y_\tau^+(\tilde{x}_s, \gamma, \alpha_\gamma, \alpha_h, c_\gamma) - y_\tau^-(\tilde{x}_s, \gamma, \alpha_\gamma, \alpha_h, c_\gamma) \right)^2 + \left(\tilde{y}_s - y_m(\tilde{x}_s, \gamma, \alpha_\gamma, \alpha_h, c_\gamma) \right)^2 \right)$$

$$s.t. \qquad \alpha_\gamma \in A_\gamma, \qquad \alpha_h \in \mathrm{A_h}, \qquad \gamma \in \Gamma$$

donde $y_m(\tilde{x}_s, \gamma, \alpha_h, \alpha_\gamma\ c_\gamma)$ se ha obtenido como la media condicionada $\hat{y}_m$ aproximada como el centro del intervalo $[y_{0.5}^-, y_{0.5}^+]$. No se debe confundir el vector de valores posibles de γ (Γ) con la matriz de pesos asociada al término de valor absoluto del funcional (Γ_w).

5 CONCLUSIONES

Recuerda siempre tus prioridades

José Francisco Torres González

El objetivo de este trabajo ha consistido en desarrollar una metodología que provea de predicciones intervalares útiles. Como se ha comprobado, el cálculo de las predicciones intervalares conlleva un coste computacional muy elevado por lo que, además de proporcionar predicciones intervalares, se ha tratado de optimizar su obtención.

Para comenzar, se ha planteado una familia de funciones de disimilitud que consisten en un problema de optimización estrictamente convexo dependiente del parámetro γ. Esta familia estima la función de densidad empírica de las salidas y_k que se quieren predecir, condicionadas a una entrada x_k. Para un caso particular, donde $\gamma = 0$, el método se reduce a la obtención de la función de densidad normal multivariada. Esto implica que la regresión con esta familia de funciones es, en realidad, una generalización de los métodos clásicos de predicción.

También se ha visto que la estimación con funciones de disimilitud ofrece una mayor garantía para casos en los que el set de datos D tenga una cantidad suficiente de muestras representativas. Sin embargo, la eficacia de esta estimación decae si la anterior condición no se cumple ya que las salidas y_k a predecir se salen excesivamente del dominio de D provocando que el método no sea efectivo. Es decir, la metodología tiene ciertas garantías de funcionar sólo si la base de datos es suficientemente representativa.

Por otro lado, el problema de optimización convexa que plantea la familia de funciones de disimilitud se ha abordado usando una formulación dual que relaje las restricciones de igualdad convexas de dicho problema. Esta formulación dual tiene, entre otras, la característica de que el número de variables a calcular se reduzca a $n + 1$ variables de decisión duales donde $n + 1$ es el número de restricciones de igualdad. Así mismo, el funcional dual es diferenciable en contraposición del primal. Además, una vez resuelto el problema dual, las N variables primales pueden obtenerse directamente resolviendo N problemas de optimización escalares.

Puesto que el problema dual no es estrictamente convexo, se ha aplicado un método gradencial acelerado llamado FISTA. Este método ha demostrado tener una ratio de convergencia mayor que los métodos clásicos de gradiente descendente. Sin embargo, carecía de sentido elegir un algoritmo de estas características si la implementación práctica no fuera buena ya que, en ese caso, se limitaría sensiblemente su eficacia. Por ello se ha tratado de optimizar su implementación en el código.

Así mismo, tal y como se ha advertido en las gráficas, el método FISTA no es un método de gradiente descendente. Es decir, no se puede asegurar que, en cada iteración, el algoritmo se aproxime estrictamente a la solución óptima por lo que, dependiendo del problema, el algoritmo puede oscilar (*rippling*).

Para solucionar este inconveniente se ha seguido el planteamiento de reinicio propuesto por [9] que consiste en reiniciar el *momentum*, propuesto por [8], a 0 ya que las oscilaciones ocurren cuando este se sobreestima. En [9] se demuestra una convergencia lineal si se cumple que siempre que el gradiente se reduzca e veces (siendo e el número de Euler) respecto a un gradiente inicial, se reinicie el algoritmo.

Lo anteriormente expuesto se ha comprobado aplicando esta estrategia de reinicio en todos los casos estudiados en los que se presentaban oscilaciones, obteniendo una convergencia lineal y una rapidez muy superior.

Finalmente, para obtener predicciones intervalares, se parte de las funciones de densidad empíricas condicionadas que habían sido estimadas con las funciones de disimilitud. Se han usado principalmente dos algoritmos: el primero calcula la función de densidad condicionada para una salida y_k y, con su función de distribución, se estima un cuantil condicionado inferior y un cuantil condicionado superior. Puesto que el Algoritmo 1 depende de los parámetros γ y c se ha necesitado de un segundo algoritmo para poder estimar el c_{opt} para una γ dada.

También, se ha minimizado una función de coste Q_γ que penalizaba tanto la longitud media de los intervalos como la disparidad entre la estimación $\hat{y}_k$ y su valor real. Esta función de coste ha permitido encontrar la sintonización adecuada de γ (con su c_{opt}) para las predicciones intervalares. Sin ella, por lo que se ha visto en los diferentes resultados planteados en el TFG, hubiera sido muy complicado obtener esos parámetros.

Por último, en los casos presentados, ha sido mucho más determinante el valor de c, que reduce y ajusta la longitud del intervalo, que los demás parámetros a sintonizar referidos a la inclusión de información local de la función de coste $J(z, D)$. Es decir, en el equilibrio que debe haber entre los demás parámetros a sintonizar y c, es preferible una predicción con α_γ y α_h relativamente bajos que permitan un valor de c más alto. Esta configuración es la que minimiza la función de coste Q_γ que además proporciona la γ que mejor ajusta la forma de la función de densidad. No obstante, el cálculo de las predicciones intervalares y en concreto Q_γ es muy caro computacionalmente. De ahí la importancia de haber intentado conseguir un código lo más depurado y eficiente posible.

6 LIBRERÍA DE FUNCIONES PYTHON

Carecer de libros propios es el colmo de la miseria.

Benjamin Franklin

Cada paso que hemos dado en este Trabajo de Fin de Grado ha sido gracias a desarrollar un código paralelo de Python en el maravilloso entorno de Pycharm. Este IDE (*Integrated Development Environment*) facilita la vida a cualquier programador y supera con creces, sin menospreciarlo, el entorno de MatLab. Permite autocompletar código de manera inteligente salvándonos de innumerables errores y ahorrándonos tiempo. Además de que permite que nos movamos por el código de manera mucho más cómoda.

Así pues, las funciones que hemos desarrollado durante todo el trabajo permiten, con bastante facilidad, predecir con regresión de funciones de disimilitud y proveer de predicciones intervalares.

Sin mermar esta simplicidad de manejo, hemos incluido una batería de opciones en **6.1 OPTIONS.py** necesaria para configurar cada una de las funciones de las dos librerías principales (*disimilarity_regression.py* e *Interval_Estimation.py*). No importa si decidimos no incluir la clase *OPTIONS* ya que todas las funciones tienen valores por defecto.

A parte de las dos librerías principales, se incluirán tanto los códigos de las predicciones como los códigos 'laboratorio' que nos han permitido avanzar en el desarrollo del TFG.

No hace falta remarcar que sería recomendable descargar una versión de Python para que hagamos uso de estas librerías.

6.1 OPTIONS.py

Es crucial que importemos esta librería general para cada uno de los códigos que queramos hacer. Aunque todas las funciones son autosuficientes, es lógico (y necesario) que queramos cambiar los parámetros con los que se predice. Así pues, si queremos movernos por estas opciones tan solo habrá que llamar al método *.mostrar()* que no devolverá el estado de cada una de las variables junto a una pequeña explicación de que hace cada una.

No todas las opciones tienen un impacto en las funciones. Por ejemplo, si queremos estimar con funciones de disimilitud usando **6.3 disimilarity_regression.py**, carece de sentido que modifiquemos *.tau* ya que este parámetro define el cuantil inferior y superior de las predicciones intervalares.

Además de parámetros que configuran el funcionamiento de las funciones, la clase *OPTIONS* nos permite elegir si queremos que se impriman por pantalla las iteraciones medias, el tiempo de ejecución, los máximos de la matriz de pesos…

```
    import numpy as np
class OPTIONS:
    def __init__(self):
        self.max_iter = 50000
        self.max_error = 1e-3
        self.M = 200
        self.tiempo_computo = False
        self.iter = True
        self.pond_similitud = False
        self.pond_agua_pasada = False
        self.per_train_val_test = [60, 20, 20, True]
        self.w = 0
        self.gam_w = 0
        self.tau = 0.05
        self.Y = None
        self.D_dinamica = False
        self.Norm = False



    def mostrar(self):
        print(f"Iteraciones máximas (.max_iter): {self.max_iter}")
        print(f"Cota superior de error (.max_error): {self.max_error}")
        print(f"Número de intervalos discretos para el conjunto Y (.M): {self.M}")
        print(f"Muestra el tiempo de cómputo (.tiempo_computo): {self.tiempo_computo}")
        print(f"Muestra las iteraciones medias y máximas a parte de gráficas de convergencia
(.iter): {self.iter}")
        print(f"Información local según similitud del punto a predecir (.ponderacion_similitud):
{self.pond_similitud}")
        print(f"Ponderacion del dato en función de si es reciente (.ponderacion_agua_pasada):
{self.pond_agua_pasada}")
        print(f"Porcentaje de división de los datos en Entrenamiento, validación y
ensayo(.per_train_val_test[train, val, test, True o False si se quiere usar]): "
              f"{self.per_train_val_test}")
        print(f"Parámetro para controlar la importancia de los pesos (.w): {self.w}")
        print(f"Define el cuantil superior e inferior del intervalo (.tau): {self.tau}")
        print(f"Parámetro para controlar la información local de de gamma (.gam_w):
{self.gam_w}")
        print(f"Set discreto de Y. None por defecto (.Y): {self.Y}")
        print(f"Base de datos dinámica (.D_dinamica): {self.D_dinamica}")
        print(f"Normalización de los datos (.Norm): {self.Norm}")


    """Nos permite asegurar de que si los valores de los pesos se van de madre, se normalicen
para un
         Parámetro máximo que incluyamos"""
    def pesos_similitud(H, R, z, w, max_peso=None):
    N = H[0, :].size
    m = z[:, 0].size
    R_k = np.zeros([m, 1])
    exp_peso = np.zeros([1, N])
    Hw = np.eye(N)
    for k in range(0, N):
        R_k[:, 0] = R[:, k]
        exp_peso[0, k] = w * np.linalg.norm(R_k - z) ** 2 #Para empezar se calcula el exponente

    if max_peso is not None:
        max_exp = np.max(exp_peso) #Averiguamos cual sería el exp max que haría superar el lim
        max_exp_peso = np.log(max_peso) #Para que np.exp no se vaya de madre
```

```
        if max_exp > max_exp_peso:
            exp_peso = exp_peso * (max_exp_peso/max_exp) #Normalizamos con el peso máximo

    for k in range(0, N):
        Hw[k, k] = H[k, k] * np.exp(exp_peso[0, k])
    max_Hw = np.max(Hw)
    return Hw, max_Hw
```

6.2 Interval_Estimation.py

Esta es la librería principal que tenemos que importar si nuestra intención es la de predecir intervalarmente. Normalmente tendremos que llamar primero a ***algorithm2*** para que nos devuelva el c_{opt}. Esta función tiene la posibilidad de dividir automáticamente el set de datos en set de validación y entrenamiento con los porcentajes definidos en *opt.per_train_test.* Si no fuera el caso, también se pueden introducir por separado como argumentos los sets de entrenamiento y validación. A parte de esta división, si estamos trabajando una serie temporal en la que la base de datos es dinámica, lo lógico es incluir los datos del set de validación en el set de entrenamiento conforme se vayan usando. Para ello se usa *opt.D_dinamica = True*. Si en cualquiera de estas opciones se mete la pata, por ejemplo, si los porcentajes de la división no suman 100, se corregirá automáticamente.

Después de llamar al Algoritmo 2 y conseguir el c_{opt} para una γ dada, tenemos que usar ***algorithm1*** para que nos devuelva predicciones intervalares. Existe la posibilidad tanto de que nos devuelva una predicción intervalar como de que nos devuelva el mismo número de intervalos que la dimensión de la entrada x_k. En general, si se usa una base de datos dinámica, lo lógico es predecir intervalo a intervalo ya que set_x y set_y cambian. Si simplemente se tiene un set de entrenamiento set_x y set_y y un set de validación x_k entonces se devolverán tantos intervalos como columnas tenga x_k.

Después, si queremos concretamente la función de densidad empírica de una salida y_k para x_k en un set de datos D, usamos ***ec_pdf***. Y por último si por alguna razón queremos saber cómo se ha dividido internamente el conjunto de datos de discretos de $\mathcal{Y}$ usamos ***set_Yhat***.

6.2.1 Importaciones necesarias

```
    #import Optimization.FISTA_vBasedRestart as FISTA
import Optimization.FISTA_v5 as FISTA
import numpy as np
import time
from Optimization.OPTIONS import pesos_similitud
from Optimization.OPTIONS import OPTIONS

"""Aquí vamos a tratar de mejorar las iteraciones/error FISTA usando el mismo funcional dual que
en el anterior
 cálculo óptimo como x_ini"""

"""En caso de que nuestro problema oscile a la hora de resolverlo con FISTA habrá que cambiar e
importar FISTA_vBasedRestart"""
```

6.2.2 set_Yhat

```
"""Esta función nos devuelve el conjunto discreto de Y para un número de divisiones M dado"""
def set_Yhat(set_y, M=None):
    if M is None or M < 50:
        M = 200
    dim_yj = set_y[:, 0].size
    y1 = np.zeros([dim_yj, 1])
    y1[:, 0] = np.min(set_y) / 1
    yM = np.max(set_y)*1
    Y = np.zeros([1, M])
    incr_yj = (yM - y1) / (M - 1)
    for j in range(0, M):
        Y[0, j] = y1 + incr_yj * j
    return Y, y1, yM
```

6.2.3 *ec_pdf*

```
    """Esta función comprende los dos primeros pasos del algoritmo 1"""
def ec_pdf(set_x, set_y, xk, gamma, c, opt=None):
    Inicio = time.time()
    if not isinstance(opt, OPTIONS):
        if opt is not None:
            print("Debes usar la clase IntervalEstimation.OPTIONS() para modificar los valores "
                  "por defecto")
            print("Prueba IntervalEstimation.OPTIONS.mostrar() para ver las opciones")
        opt = OPTIONS()

    N = set_x[0, :].size
    dim_xk = xk[:, 0].size
    dim_yj = set_y[:, 0].size
    H = 1*np.eye(N)

    # Lo primero es obtener el conjunto Y (perteneciente a R^M) tomando el mínimo y máximo de
                                                                                      set_y
    # Si no nos han dado el conjunto Y lo creamos
    if not isinstance(opt.Y, (list, np.ndarray)):
        Y, y1, yM = set_Yhat(set_y, opt.M)
    else:
        Y = opt.Y
        y1 = np.array([[Y[0, 0]]])

    """Tratamiento de las restricciones"""
    R_x = np.append(set_x, np.ones([1, N]), axis=0) #  Restricción de x y vector de 1
    r_xk = np.append(xk, np.ones([1, 1]), axis=0)   #  Restricción de xk y 1

    #  Incluimos las 3 restricciones de igualdad. lambda @ y = yj // lambda @ x = xk
    #  // sum(lambda) = 1
    R_yx = np.append(set_y, R_x, axis=0)
    #  La mejor opción es crear r solo una vez y no usar append más pues consume mucho.
    r_yx = np.append(y1, r_xk, axis=0)

    """Matriz de Pesos"""
    Hw = 1*np.eye(N)
    Hgam_w = 1*np.eye(N)
    max_Hw = 0
    R_yx_k = np.zeros([dim_xk + dim_yj + 1, 1])

    """Inicialización del algoritmo"""
    #  Inicio ec_pdf
    integral_Js = 0
    ecp = np.zeros([1, opt.M])
    gamma_w = gamma * np.ones([1, N])

    #  Inicio FISTA
    x_ini = np.ones([dim_xk + dim_yj + 1, 1])
    convergencia = False
    l_opt = np.zeros([N, 1])
    max_similitud = 0
    sum_iter = 0
    max_iteracion = 0

    #  Inicio tiempos
    T_pesos = 0
    T_fista = 0

    for j in range(0, opt.M):
        r_yx[0, :] = Y[0, j]
        Inicio_pesos = time.time()
        if opt.pond_similitud: #Pesos
            Hw, max_Aux = pesos_similitud(H, R_yx, r_yx, opt.w, 500000)
            if max_Aux > max_Hw:
                max_Hw = max_Aux
        if opt.pond_similitud: #Pesos
            Hgam_w, max_Aux = pesos_similitud(H, R_yx, r_yx, opt.gam_w, 500000)
            gamma_w = gamma * np.ones([1, N]) @ Hgam_w
        T_pesos = T_pesos + time.time() - Inicio_pesos #  Tiempo necesitado para los pesos

        ini_fista = time.time()
        [l, convergencia, iter, J_yj, x_ini] = FISTA.algorithm(R_yx, r_yx, Hw, gamma_w, x_ini,
                                                               opt.max_iter, opt.max_error)
        T_fista = T_fista + time.time() - ini_fista

        sum_iter = sum_iter + iter
        if max_iteracion < iter:
```

```
            max_iteracion = iter
            interJ = j
        if not convergencia:  #  Si FISTA no converge no tiene sentido seguir
            return [0, 0, 0, 0, False, 0, 0]

        """Similitud máxima y l_opt"""
        Js_yj = np.exp(-c * J_yj)
        if max_similitud < Js_yj:
            max_similitud = Js_yj
            l_opt = l
        ecp[0, j] = Js_yj
    integral_Js = np.sum(ecp)  #Factor de normalización de ecdf

    T_total = time.time() - Inicio
    if opt.tiempo_computo:
        print(f"Cómputo total: {round(T_total, 3)}s, fista: {round(100*T_fista/T_total, 3)}%, "
              f"Cáculo pesos: {round(100*T_pesos/T_total, 3)}%")

    iter_media = sum_iter / opt.M
    if opt.iter:
        print(f"Iteraciones medias: {round(iter_media, 2)} , máxima: {max_iteracion} en
                    {interJ}")

    return [integral_Js, ecp, Y, convergencia, max_Hw, l_opt, iter_media]
```

6.2.4 *algorithm1*

```
    """Predicción intervalar [y-, y+] de un solo intervalo"""

def algorithm1(set_x, set_y, xk, gamma, c, opt=None):

    #  Si lo que se ha metido no es de la clase OPTIONS se crea
    if not isinstance(opt, OPTIONS):
        if opt is not None:
            print("Debes usar la clase IntervalEstimation.OPTIONS() para modificar los valores
por defecto")
            print("Prueba IntervalEstimation.OPTIONS.mostrar() para ver las opciones")
        opt = OPTIONS()

    """Pasos I) y II) del algoritmo"""
    integral, ecp, Y, convergencia, max_Hw, l_opt, iter_media = ec_pdf(set_x, set_y, xk, gamma,
                                                                       c, opt)

    if not convergencia:
        return [0, 0, 0, 0, False, 0, 0]

    """Pasos III) y IV) del algoritmo"""
    l_lower = 0
    l_upper = 0
    #  Mejora de cómputo, nos ahorramos M - 1 divisiones al no dividir todos lo dj por integral
    lower_constrain = (1 - opt.tau)*integral
    upper_constrain = lower_constrain
    #  Con esto conseguimos una prediccion del centro del intervalo [y-50, y+50]
    median_constrain = 0.5 * integral
    median_flag = True
    ec_distr = 0

    #  Se recorre la función de distribución en los dos sentidos
    for j in reversed(range(0, opt.M)):
        ec_distr = ec_distr + ecp[0, j]
        if (ec_distr >= median_constrain) and median_flag:
            median_flag = False
            y_hat1 = Y[0, j]
        if ec_distr >= lower_constrain:
            l_lower = Y[0, j]
            break
    ec_distr = 0
    median_flag = True
    for j in range(0, opt.M):
        ec_distr = ec_distr + ecp[0, j]
        if (ec_distr >= median_constrain) and median_flag:
            median_flag = False
            y_hat2 = Y[0, j]
        if ec_distr >= upper_constrain:
            l_upper = Y[0, j]
            break
```

```
    #  Media de la prediccion ym
    y_hat = (y_hat1 + y_hat2)/2

    return ecp/integral, y_hat, l_lower, l_upper, Y, convergencia, l_opt, iter_media
```

6.2.5 *algorithm2*

```
    """Valores óptimos para c y gamma"""
"""Se da la opción de dar una serie de valores set_val y calcular los intervalos para estos o que
se haga una división
     (a partir de set_x y set_y )automática entre set de entrenamiento y validación usando
opt.per_train_test"""


def algorithm2(set_x, set_y, setx_val, sety_val, gamma, cmax, tol, opt=None):
    if not isinstance(opt, OPTIONS):
        if opt is not None:
            print("Debes usar la clase disimilarity_regression.OPTIONS() para modificar los
valores por defecto")
            print("Prueba disimilarity_regression.OPTIONS.mostrar() para ver las opciones")
        opt = OPTIONS()
    N_SET = set_x[0, :].size
    M_SET = set_x[:, 0].size

    #  División del set de datos X en D y en el set de validación
    if opt.per_train_val_test[3]:
        if np.sum(opt.per_train_val_test[0:-1]) != 100:
            opt.per_train_val_test = [60, 20, 20]
            print("La división de los datos no suma 100%. Valor por defecto [60% 20% 20%, True]")

        N_training = (N_SET * opt.per_train_val_test[0]) // 100
        Dset_x = np.zeros([M_SET, N_training])
        Dset_y = np.zeros([1, N_training])
        Dset_x[:, :] = set_x[:, 0: N_training]
        Dset_y[:, :] = set_y[:, 0: N_training]

        N_validation = (N_SET * opt.per_train_val_test[1]) // 100
        Vset_x = np.zeros([M_SET, N_validation])
        Vset_y = np.zeros([1, N_validation])
        Vset_x[:, :] = set_x[:, N_training: N_training + N_validation]
        Vset_y[:, :] = set_y[:, N_training: N_training + N_validation]

    #  Si no se escoge la opción de división entonces se toma el set de validación de los
argumentos
    else:
        if not isinstance(setx_val, (list, np.ndarray)) or not isinstance(sety_val,
                                                                          (list, np.ndarray)):
            print("Error: set_val no contiene datos y no se ha escogido la división "
                  "automática en opt.per_train_val_test")
            return[0, 0, False, 0]
        N_validation = setx_val[0, :].size

        Vset_x = np.zeros([M_SET, N_validation])
        Vset_y = np.zeros([1, N_validation])
        Dset_x = np.zeros([M_SET, N_SET])
        Dset_y = np.zeros([1, N_SET])

        Vset_x[:, :] = setx_val[:, :]
        Vset_y[:, :] = sety_val[:, :]
        Dset_x[:, :] = set_x[:, :]
        Dset_y[:, :] = set_y[:, :]

    #  Set Y
    if not isinstance(opt.Y, (list, np.ndarray)):
        opt.Y, _, _ = set_Yhat(Dset_y, opt.M)

    #  Base de datos dinámica
    if opt.D_dinamica:
        Dset_x_ini = np.zeros([M_SET, N_SET])
        Dset_y_ini = np.zeros([1, N_SET])
        Dset_x_ini[:, :] = Dset_x[:, :]
        Dset_y_ini[:, :] = Dset_y[:, :]

    #  I)
    cmin = 0
```

```
    c = 0
    conver = False
    xs = np.zeros([M_SET, 1])
    Is = np.zeros([N_validation, 2])
    #  II)
    while cmax - cmin >= tol:
        if opt.D_dinamica: #Para que vuelvan a su valor original
            Dset_x[:, :] = Dset_x_ini[:, :]
            Dset_y[:, :] = Dset_y_ini[:, :]

        #  III)
        c = 0.5*(cmin + cmax)

        #  Esto se hace para mejora de cómputo. No hace falta calcular el resto de intervalos si
ya se
        #  incumplen las restricciones de nv
        #  IV) + V) + VIII) + IV)
        nv_upper = 0
        nv_lower = 0

        print(f"cmin = {cmin}, cmax = {cmax}, c = {c}")
        print("")
        set_yhat = np.zeros([1, N_validation])
        for s in range(0, N_validation):
            xs[:, 0] = Vset_x[:, s]
            ecp_integral, set_yhat[0, s], Is[s, 0], Is[s, 1], _, conver, l_opt, _= \
                algorithm1(Dset_x, Dset_y, xs, gamma, c, opt)
            if not conver:
                return[0, 0, False, 0]

            if Is[s, 1] <= Vset_y[:, s]:
                nv_upper = nv_upper + 1
            if Is[s, 0] >= Vset_y[:, s]:
                nv_lower = nv_lower + 1

            #  Así podemos ver el transcurso del algoritmo 2
            print(f"y-: {Is[s, 0]} Vset_y[:, s] = {Vset_y[:, s]} y_hat = {set_yhat[0, s]} "
                  f"y+: {Is[s, 1]}, s = {s}")
            print(f"nv_lower = {nv_lower}, nv_upper = {nv_upper}")

            if (nv_lower/N_validation >= opt.tau) or (nv_upper/N_validation >= opt.tau):
                cmax = c
                break

            #  Se trasladan los valores de Vset a Dset y se olvidan los más antiguos de Dset.
            if opt.D_dinamica:
                Dset_x[:, 0:-1] = Dset_x[:, 1:]
                Dset_y[:, 0:-1] = Dset_y[:, 1:]
                Dset_x[:, -1] = Vset_x[:, s]
                Dset_y[:, -1] = Vset_y[:, s]


        #  VI) + VII + X)
        if (nv_lower/N_validation < opt.tau) and (nv_upper/N_validation < opt.tau):
            cmin = c


    return c, Is, conver, set_yhat
```

6.3 disimilarity_regression.py

Gracias a esta librería seremos capaces de predecir con funciones de disimilitud en el ámbito de la regresión. La función básica es *regression* que nos devuelve una salida o salidas y_k para una entrada o entradas x_k. Debemos introducir como argumentos, el set de datos D como *setx* y *sety*, la entrada/entradas, el parámetro γ y por supuesto una instancia de la clase *OPTIONS*. Para esta función NO serán útiles los siguientes atributos (de *OPTIONS*) *opt.tau, opt.Y, opt.per_train_val_test* y *opt.M*. Podremos variar los pesos

Lo más interesante de esta librería es la segunda función que contiene. Esta permite realizar un testeo completo con sets de entrenamiento validación y test. Tenemos la opción de normalizar los datos con *opt.Norm,* de montar una base de datos dinámica con *opt.D_dinamica*, y de dibujar todas las gráficas de cada una de las predicciones. Tan solo será necesario introducir como argumentos Setx y Sety. Automáticamente se

hará una repartición entre sets de entrenamiento, validación y test (con *opt.per_train_val_test*) por lo que si tenemos un set de validación o de test predefinido debemos añadirlo a Setx y Sety.

6.3.1 Importaciones Necesarias

```
import numpy as np
import Optimization.FISTA_v5 as FISTA
#import FISTA_vBasedRestart as FISTA
import matplotlib.pyplot as plt
import time
from OPTIONS import OPTIONS
from OPTIONS import pesos_similitud
```

6.3.2 *regression*

```
"""Con esta funcion seremos capaces de predecir la salida o salidas yk referida a la entrada o
entradas xk"""

def regression(SETx, SETy, xk, gamma, H, opt=None):
    if not isinstance(opt, OPTIONS):
        if opt == (not None):
            print(
                "Debes usar una instancia de disimilarity_regression.OPTIONS() para modificar los
valores por defecto")
            print("Prueba disimilarity_regression.OPTIONS.mostrar() para ver las opciones")
        opt = OPTIONS()
    #  Normalización de los datos
    if opt.Norm:
        min_x = np.min(SETx)
        max_x = np.max(SETx) + 1
        A = (SETx - min_x) / (max_x - min_x)
        b = (xk - min_x) / (max_x - min_x)

        min_y = np.min(SETy)
        max_y = np.max(SETy)
        set_y = (SETy - min_y) / (max_y - min_y)
    else:
        A = SETx
        set_y = SETy
        b = xk

    Inicio = time.time()
    N = A[0, :].size
    M = A[:, 0].size
    tam_x = b[0, :].size

    # Incluimos la condición de igualdad de sum(lambda) = 1 insertando una fila de
    # unos en la matriz de datos A y llamándola R. Lo mismo con el dato de entrada

    R = np.append(A, np.ones([1, N]), axis=0)
    r = np.append(b, np.ones([1, tam_x]), axis=0)
    Hw = 1*np.eye(N)

    convergencia = 0
    y_predict = np.zeros([set_y[:, 0].size, tam_x])  # set_y podría tener otra dimensión de b. A
lo mejor es escalar
    l_opt = np.zeros([N, 1])
    z = np.zeros([M + 1, 1])
    R_k = np.zeros([M + 1, 1])

    """Set de validación"""
    sum_iter = 0
    x_ini = np.ones([M + 1, 1])
    Iter_max = 0
    sum_time = 0
    max_Hw = 0
    max_Aux = 0
    gamma_w = gamma * np.ones([1, N])
    for i in range(0, tam_x):
        z[:, 0] = r[:, i]
        if opt.pond_similitud: #  Añadimos información local al algoritmo
            Hw, max_Aux = pesos_similitud(H, R, z, opt.w, 10000)
            Hgam_w, _ = pesos_similitud(H, R, z, opt.gam_w, 10000)
            gamma_w = gamma * np.ones([1, N]) @ Hgam_w
        elif opt.pond_agua_pasada:#  ponderación según datos recientes o antiguos
```

```
            pass
        if max_Hw < max_Aux:
            max_Hw = max_Aux

        [l_opt, convergencia, iter, _, x_ini] = FISTA.algorithm(R, z, Hw, gamma_w, x_ini,
opt.max_iter, opt.max_error)

        sum_iter = sum_iter + iter
        if Iter_max < iter:
            Iter_max = iter

        if not convergencia:
            break
        sol = set_y @ l_opt
        y_predict[:, i] = sol[:, 0]

    if opt.iter:
        print(f"Media de iteraciones: {sum_iter / tam_x}, Máxima iteración: {Iter_max}")
    if opt.tiempo_computo:
        print(f"Tiempo de cómputo: {time.time() - Inicio}")
    #  Se desnormaliza
    if opt.Norm:
        y_predict_sinnorm = y_predict * (max_y - min_y) + min_y
    else:
        y_predict_sinnorm = y_predict

    return [y_predict_sinnorm, convergencia, max_Hw, sum_iter]
```

6.3.3 *training_validation_test_regression*

```
"""Sets de entrenamiento, validación y ensayo"""


def training_validation_test_regression(SETx, SETy, opt=None):
    if not isinstance(opt, OPTIONS):
        if opt == (not None):
            print("Debes usar la clase disimilarity_regression.OPTIONS() para modificar los
valores por defecto")
            print("Prueba disimilarity_regression.OPTIONS.mostrar() para ver las opciones")
        opt = OPTIONS()
    N_SET = SETx[0, :].size
    M_SET = SETx[:, 0].size

    if sum(opt.per_train_val_test) != 100:
        opt.per_train_val_test = [60, 20, 20]
        print("La división de los datos no suma 100%. Valor por defecto [60% 20% 20%]")
    N_training = (N_SET * opt.per_train_val_test[0]) // 100
    training_set_x = SETx[:, 0: N_training]
    training_set_y = SETy[:, 0: N_training]

    N_validation = (N_SET * opt.per_train_val_test[1]) // 100
    validation_set_x = SETx[:, N_training: N_training + N_validation]
    validation_set_y = SETy[:, N_training: N_training + N_validation]

    N_test = (N_SET * opt.per_train_val_test[2]) // 100
    test_set_x = SETx[:, N_training + N_validation: N_training + N_validation + N_test]
    test_set_y = SETy[:, N_training + N_validation: N_training + N_validation + N_test]

    repetir = 1

    gamma = 0

    #  Primero repetiremos hasta que se considere que el parametro gamma está bien ajustado.
Después se pasará al test_set
    while repetir != -1:
        # Matriz Hessiana de Pesos J = Lambda.T * H * Lambda + gamma * sum(abs(Lambda))
        H = 1 * np.eye(N_training)
        gamma = float(input("Dale la gamma: "))
        if opt.pond_similitud or opt.pond_agua_pasada:
            opt.w = float(input("Introduce el parámetro para ponderar los datos: "))
            opt.gam_w = float(input("Introduce el parámetro para ponderar gamma: "))

        if opt.D_dinamica:
            Dset_x = np.array(training_set_x[:, :])
            Dset_y = np.array(training_set_y[:, :])
```

```
            Vset_x = np.array([validation_set_x[:, 0]])
            Vset_y = np.array([validation_set_y[:, 0]])
            y_val_predict = np.zeros([1, N_validation])
            maxHw = 0
            for kk in range(0, N_validation):
                Vset_x[:, 0] = validation_set_x[:, kk]
                Vset_y[:, 0] = validation_set_y[:, kk]

                y_val_predict[0, kk], convergencia, auxHw, _ = regression(Dset_x, Dset_y, Vset_x,
gamma, H, opt)
                if auxHw > maxHw:
                    maxHw = auxHw
                Dset_x[:, 0:-1] = Dset_x[:, 1:]
                Dset_y[:, 0:-1] = Dset_y[:, 1:]
                Dset_x[:, -1] = Vset_x[:, 0]
                Dset_y[:, -1] = Vset_y[:, 0]
        else:
            [y_val_predict, convergencia, maxHw, sum_iter] = regression(training_set_x,
training_set_y, validation_set_x, gamma, H, opt)
        print(f"Max Hw: {maxHw}")
        if convergencia:
            plt.close('all')
            plt.figure(figsize=(13, 6))
            print("Error absoluto de validación: ")
            print(np.mean(np.abs(y_val_predict - validation_set_y)))
            rel_err = (y_val_predict - validation_set_y) / validation_set_y
            print(f"Error relativo: {round(np.mean(np.abs(rel_err))*100, 3)}%")
            x = np.arange(0, N_training)
            y_train = plt.plot(x, np.array(training_set_y[0, :]), 'b-')[0]
            x = np.arange(N_training, N_training + N_validation)
            y_pred = plt.plot(x, y_val_predict[0, :], 'r--')[0]
            x = np.arange(N_training, N_training + N_validation)
            y_val = plt.plot(x, validation_set_y[0, :], 'g-')[0]

            plt.title("Predicción de la evolución del COVID-19")
            plt.xlabel("Días desde 2020-04-12")
            plt.ylabel("Hospitalizados por COVID")
            plt.legend([y_train, y_pred, y_val], ["Set de datos previos", "Predicción", "Casos a
predecir"])
            plt.show(block=False)
            plt.pause(0.5)

        else:
            print("No converge para este numero máximo de iteraciones o este gamma dado")

        print(" ")
        if input("¿Quieres repetir y probar otra gamma (y/n)? ") == 'y':
            repetir = 1
        else:
            repetir = -1

    H = 1 * np.eye(N_training)

    if opt.D_dinamica:
        Dset_x = np.array(training_set_x[:, :])
        Dset_y = np.array(training_set_y[:, :])
        Tset_x = np.array([test_set_x[:, 0]])
        Tset_y = np.array([test_set_y[:, 0]])
        y_test_predict = np.zeros([1, N_test])
        maxHw = 0
        for kk in range(0, N_test):
            Tset_x[:, 0] = test_set_x[:, kk]
            Tset_y[:, 0] = test_set_y[:, kk]

            y_test_predict[0, kk], convergencia, auxHw, _ = regression(Dset_x, Dset_y, Tset_x,
gamma, H, opt)
            if auxHw > maxHw:
                maxHw = auxHw
            Dset_x[:, 0:-1] = Dset_x[:, 1:]
            Dset_y[:, 0:-1] = Dset_y[:, 1:]
            Dset_x[:, -1] = Tset_x[:, 0]
            Dset_y[:, -1] = Tset_y[:, 0]
    else:
        [y_test_predict, convergencia, auxHw, _] = regression(training_set_x, training_set_y,
test_set_x, gamma, H, opt)
    if convergencia:
        plt.close('all')
        plt.figure(figsize=(10, 10))
        print("Error absoluto del ensayo: ")
```

```
        print(np.mean(np.abs(y_test_predict - test_set_y)))
        x = np.array([np.arange(0, N_training)])
        y_train = plt.plot(x, training_set_y, 'b-')[0]

        x = np.array([np.arange(N_training, N_training + N_validation)])
        y_pred = plt.plot(x, y_val_predict, 'r--')[0]
        y_val = plt.plot(x, validation_set_y, 'g-')[0]

        x = np.array([np.arange(N_training + N_validation, N_training + N_validation + N_test)])
        y_test = plt.plot(x, test_set_y, 'k-')[0]
        y_test_pred = plt.plot(x, y_test_predict, 'm--')[0]
        plt.xlabel("Entradas")
        plt.ylabel("Salidas")


        plt.legend([y_train, y_pred, y_val, y_test, y_test_pred], ['Set de entrenamiento',
'Predicción validación', 'Set de validación', 'Set Test', 'Predicción Test'])
        plt.show()
    else:
        print("No converge para este numero máximo de iteraciones o este gamma dado")
```

6.4 Estimacion_Intervalar_Demanda.py

Este archivo es un ejemplo de uso de la librería *IntervalEstimation.* Se hacen predicciones intervalares de la demanda eléctrica española para una sintonización que elijamos con *OPTIONS*. Es un buen ejemplo de una base de datos dinámica y del uso de la opción *opt.D_dinamica = True* del Algoritmo 2. Como la cantidad de parámetros que debemos sintonizar para las predicciones intervalares es amplia, tenemos la opción de un modo *Auto*. Se elegirán automáticamente los parámetros óptimos $\gamma^*, \alpha_h^*, \alpha_\gamma^*$ entre un rango de valores como se explica en **4.5.2 Función de coste Q_γ**.

6.4.1 Importaciones Necesarias

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from time import time
import IntervalEstimation_v6 as ie
from Optimization.OPTIONS import OPTIONS
```

6.4.2 Cuerpo del Código

```
demanda_nacional = pd.read_csv('DailyData.txt', header=0)
demanda_nacional.columns = ['demanda', 'dia', 'dia_semana']

Demanda_total = np.array([demanda_nacional.demanda])
Dia_semana = np.array([demanda_nacional.dia_semana])
dias_totales = Demanda_total[0, :].size
I_norm = np.zeros([dias_totales, 2])

dias_vista = 7  # Siete días vista
dias_base_datos = 60  # Días en los que apoyar la siguiente predicción
setx = np.zeros([4, dias_base_datos - dias_vista - 7])
sety = np.zeros([1, dias_base_datos - dias_vista - 7])
xk = np.zeros([4, 1])
predict_norm = np.zeros([1, dias_totales])

setx_val = np.zeros([4, dias_base_datos*2])
sety_val = np.zeros([1, dias_base_datos*2])


opt = OPTIONS()

"""Inicialización de parámetros para el algoritmo"""
#  Matriz hessiana
H = np.eye(dias_base_datos)
#  Normalización
demanda_max = np.max(Demanda_total)
demanda_min = np.min(Demanda_total)
opt.M = 150
opt.Y ,_ ,_ = ie.set_Yhat((Demanda_total - demanda_min)/(demanda_max - demanda_min), opt.M) #Se
```

```
establece el conjunto de salidas Y
gamma = 0
opt.iter = False
opt.per_train_val_test[3] = False
opt.D_dinamica = True
opt.max_error = 1e-2
opt.max_iter = 100000
opt.tau = 0.1
#  Tolerancia de c y cmin
tol = 1e-1
T = np.arange(0.1, 0.2, 0.01)
GW = np.arange(0., 0.1, 0.01)
while gamma > -1:
    auto = float(input("¿Automatizar? "))
    if auto != 1:
        gamma = float(input("Dale la gamma: "))
        opt.w = float(input("Dale a los pesos: "))
        opt.gam_w = float(input("Dale a los pesos de gamma: "))
        cmax = float(input("Dale a la cmax: "))
        if opt.w <= 0:
            opt.pond_similitud = False
        else:
            opt.pond_similitud = True


    """c óptimo para un gamma dado en un set de validación con 50% set D y 50% set validación"""
    setx[0, :] = Demanda_total[0, 0: dias_base_datos - 7 - dias_vista]  # Hace una semana
    setx[1, :] = Demanda_total[0, 6: dias_base_datos - 1 - dias_vista]  # Ayer
    setx[2, :] = Demanda_total[0, 7: dias_base_datos - dias_vista]  # Hoy
    sety[0, :] = Demanda_total[0, dias_vista + 7: dias_base_datos]  # Demanda en dias_vista

    setx_norm = (setx - demanda_min) / (demanda_max - demanda_min)
    sety_norm = (sety - demanda_min) / (demanda_max - demanda_min)
    setx_norm[3, :] = (Dia_semana[0, dias_vista + 7: dias_base_datos] - 1) / 6  # Día de la
semana

    setx_val[0, :] = Demanda_total[0, dias_base_datos - 7 - dias_vista: dias_base_datos*3 - 7 -
dias_vista]  # Hace una semana
    setx_val[1, :] = Demanda_total[0, dias_base_datos - 1 - dias_vista: dias_base_datos*3 - 1 -
dias_vista]  # Ayer
    setx_val[2, :] = Demanda_total[0, dias_base_datos - dias_vista: dias_base_datos*3 -
dias_vista]  # Hoy
    sety_val[0, :] = Demanda_total[0, dias_base_datos: dias_base_datos*3]  # Demanda en
dias_vista

    setx_val_norm = (setx_val - demanda_min) / (demanda_max - demanda_min)
    sety_val_norm = (sety_val - demanda_min) / (demanda_max - demanda_min)
    setx_val_norm[3, :] = (Dia_semana[0, dias_base_datos: dias_base_datos * 3] - 1) / 6  # Día de
la semana

    if auto == 1:
        opt.iter = False
        opt.pond_similitud = True
        opt.w = 0.5
        minQ = 1000000
        for t in T:
            print(f"Cómputo total: {((t-1)*100)/4}%")
            for gw in GW:
                opt.gam_w = gw
                print(f"gamma: {t}, peso: {gw}, peso: {opt.w}")
                c, I_val, conver, set_yhat = ie.algorithm2(setx_norm, sety_norm, setx_val_norm,
sety_val_norm, t, 10, tol, opt)

                if conver:
                    AuxQ = np.sum((I_val[:, 1] - I_val[:, 0])**2 + (sety_val_norm - set_yhat)**2)
                    if AuxQ < minQ:
                        minQ = AuxQ
                        final_gamma = t
                        final_pesogam = gw
                        final_c = c
                        print(f"En progreso: final_gamma: {final_gamma}, final_pesogam:
{final_pesogam}, final_c: {final_c}")
        print(f"final_gamma: {final_gamma}, final_pesogam: {final_pesogam}, final_c: {final_c}")

    else:
        c, I_val, conver, _ = ie.algorithm2(setx_norm, sety_norm, setx_val_norm, sety_val_norm,
gamma, cmax, tol, opt)

    if not conver:
```

```
        print("El algoritmo2 no converge para este gamma, cmax o peso dado")
        break
    print("")
    print("")
    print(f"cmax obtenido: {c}. Predicción intervalar: ")
    max_Hw = 0
    max_iter = 0
    sum_iter = 0
    porcentaje_computo = 0
    time_ini = time()
    for kk in range(dias_base_datos*2, dias_totales - dias_vista):
        setx[0, :] = Demanda_total[0, kk - dias_base_datos: kk - 7 - dias_vista] #Hace una semana
        setx[1, :] = Demanda_total[0, kk - dias_base_datos + 6: kk - 1 - dias_vista] #Ayer
        setx[2, :] = Demanda_total[0, kk - dias_base_datos + 7: kk - dias_vista] #Hoy

        sety[0, :] = Demanda_total[0, kk - dias_base_datos + dias_vista + 7: kk] #Demanda en
dias_vista

        xk[0, 0] = Demanda_total[0, kk - dias_vista]
        xk[1, 0] = Demanda_total[0, kk - 1]
        xk[2, 0] = Demanda_total[0, kk]

        setx_norm = (setx - demanda_min) / (demanda_max - demanda_min)
        sety_norm = (sety - demanda_min) / (demanda_max - demanda_min)
        xk_norm = (xk - demanda_min) / (demanda_max - demanda_min)

        setx_norm[3, :] = (Dia_semana[0, kk - dias_base_datos + dias_vista + 7: kk] - 1) / 6 #Día
de la semana
        xk_norm[3, 0] = (Dia_semana[0, kk + dias_vista] - 1) / 6

        opt.iter = False
        _, y_hat, I_norm[kk + dias_vista, 0], I_norm[kk + dias_vista, 1], _, conver, l_opt, iter
= ie.algorithm1(setx_norm, sety_norm, xk_norm, gamma, c, opt)
        if not conver:
            print("No converge para este gamma o peso dado")
            break
        sum_iter = sum_iter + iter
        predict_norm[0, kk + dias_vista] = y_hat
        if porcentaje_computo <= 100*(kk - dias_base_datos*2) / (dias_totales - dias_vista):
            print(f"Completo al {porcentaje_computo}%")
            porcentaje_computo = porcentaje_computo + 1
        if kk == (dias_totales - dias_base_datos)//4:
            print("25%")
        if kk == (dias_totales - dias_base_datos)//2:
            print("50%")
        if kk == 3*(dias_totales - dias_base_datos)//4:
            print("75%")

    t_total = time() - time_ini

    if conver:
        predict = predict_norm * (demanda_max - demanda_min) + demanda_min
        Is = I_norm * (demanda_max - demanda_min) + demanda_min
        error_abs = np.mean(np.abs(Demanda_total[0, dias_base_datos + dias_vista: -1] -
predict[0, dias_base_datos + dias_vista: -1]))
        set_er_rel = (Demanda_total[0, dias_base_datos + dias_vista: -1] - predict[0,
dias_base_datos + dias_vista: -1]) / predict[0, dias_base_datos + dias_vista: -1]
        error_rel = round(np.mean(np.abs(set_er_rel))*100, 3)
        if opt.iter:
            print(f"Iteraciones medias: {round(sum_iter/(dias_totales - dias_base_datos -
dias_vista), 3)}. Máxima iteración: {max_iter}")
        if opt.pond_similitud:
            print(f"Máximo peso: {max_Hw}")
        print(f"Error absoluto: {error_abs}. Error relativo: {error_rel}%. Tiempo para predicción
intervalar: {t_total}")
        print("")

        x = np.arange(0, dias_totales)
        fig, axs = plt.subplots(2)


        axs[0].set_title("Predicción intervalar de la demanda eléctrica española")
        axs[0].set_ylabel("Demanda Acumulada(MW)")
        axs[0].set_xlabel("Días")
        axs[1].set_ylabel("Demanda Acumulada(MW)")
        axs[1].set_xlabel("Días")

        real, = axs[0].plot(x, Demanda_total[0, :], marker='o')
        estimada, = axs[0].plot(x[dias_base_datos*2 + dias_vista:], predict[0, dias_base_datos*2
```

```
+ dias_vista:], marker='x')
        I_lower, = axs[0].plot(x[dias_base_datos*2 + dias_vista:], Is[dias_base_datos*2 +
dias_vista:, 0], 'b')
        I_upper, = axs[0].plot(x[dias_base_datos*2 + dias_vista:], Is[dias_base_datos*2 +
dias_vista:, 1], 'r')

        axs[1].plot(x[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108],
Demanda_total[0, dias_totales - dias_vista * 114: dias_totales - dias_vista * 108], marker='o')
        axs[1].plot(x[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108],
predict[0, dias_totales - dias_vista * 114: dias_totales - dias_vista * 108], marker='x')
        axs[1].plot(x[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108],
Is[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108, 0], 'b')
        axs[1].plot(x[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108],
Is[dias_totales - dias_vista * 114: dias_totales - dias_vista * 108, 1], 'r')

        plt.legend([real, estimada, I_lower, I_upper], ["Demanda real", "Demanda estimada",
"Intervalo inferior", "Intervalo superior"])

        plt.figure(2)
        real, = plt.plot(x[dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], Demanda_total[0, dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], marker='o')
        estimada, = plt.plot(x[dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], predict[0, dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], marker='x')
        I_lower, = plt.plot(x[dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], Is[dias_totales - dias_base_datos - dias_vista*3: dias_totales - dias_vista, 0],
'b')
        I_upper, = plt.plot(x[dias_totales - dias_base_datos - dias_vista*3: dias_totales -
dias_vista], Is[dias_totales - dias_base_datos - dias_vista*3: dias_totales - dias_vista, 1],
'r')

        plt.title("Predicción intervalar de la demanda eléctrica española")
        plt.ylabel("Demanda Acumulada(MW)")
        plt.xlabel("Días")
        plt.legend([real, estimada, I_lower, I_upper], ["Demanda real", "Demanda estimada",
"Intervalo inferior", "Intervalo superior"])
        plt.show()
    else:
        plt.plot(Demanda_total)
        plt.show()
```

6.5 Regresion_Demanda.py

Este código toma los mimos datos de demanda eléctrica que el archivo anterior y trata de predecir, usando la librería *disimilarity_regression.py* esta demanda a 7 días vista. Aunque de serie se prediga a 7 días vista, el código permite que cambiemos este número. Al igual que para la predicción intervalar, haremos uso de una instancia de *OPTIONS* para configurar los parámetros que queramos de cara a la predicción.

6.5.1 Importaciones Necesarias

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import disimilarity_regression_v4 as dr
from time import time
from OPTIONS import OPTIONS
```

6.5.2 Cuerpo del Código

```
demanda_nacional = pd.read_csv('DailyData.txt', header=0)
demanda_nacional.columns = ['demanda', 'dia', 'dia_semana']

Demanda_total = np.array([demanda_nacional.demanda])
Dia_semana = np.array([demanda_nacional.dia_semana])
dias_totales = Demanda_total[0, :].size
print(dias_totales)
dias_vista = 7  # Siete días vista
dias_base_datos = 120
setx = np.zeros([4, dias_base_datos - dias_vista - 7])
sety = np.zeros([1, dias_base_datos - dias_vista - 7])
xk = np.zeros([4, 1])
```

```
predict_norm = np.zeros([1, dias_totales])

opt = OPTIONS()

"""Inicialización de parámetros para el algoritmo"""
#  Matriz hessiana
H = np.eye(dias_base_datos - dias_vista - 7)
#  Normalización
demanda_max = np.max(Demanda_total)
demanda_min = np.min(Demanda_total)
gamma = 0
opt.Norm = False
opt.max_error = 1e-3
opt.iter = False
conver = False

while gamma > -1:
    opt.tiempo_computo = False
    opt.iter = False
    gamma = float(input("Dale la gamma: "))
    opt.w = float(input("Dale a los pesos: "))
    opt.gam_w = float(input("Dale a los pesos de gamma: "))
    if (opt.w == 0) and (opt.gam_w <= 0):
        opt.pond_similitud = False
    else:
        opt.pond_similitud = True

    max_Hw = 0
    max_iter = 0
    sum_iter = 0
    tini = time()
    kkitermax = -1
    for kk in range(dias_base_datos, dias_totales - dias_vista):
        setx[0, :] = Demanda_total[0, kk - dias_base_datos: kk - 7 - dias_vista] #Hace una semana
        setx[1, :] = Demanda_total[0, kk - dias_base_datos + 6: kk - 1 - dias_vista] #Ayer
        setx[2, :] = Demanda_total[0, kk - dias_base_datos + 7: kk - dias_vista] #Hoy

        sety[0, :] = Demanda_total[0, kk - dias_base_datos + dias_vista + 7: kk]

        xk[0, 0] = Demanda_total[0, kk - dias_vista]
        xk[1, 0] = Demanda_total[0, kk - 1]
        xk[2, 0] = Demanda_total[0, kk]

        setx_norm = (setx - demanda_min) / (demanda_max - demanda_min)
        sety_norm = (sety - demanda_min) / (demanda_max - demanda_min)
        xk_norm = (xk - demanda_min) / (demanda_max - demanda_min)

        setx_norm[3, :] = (Dia_semana[0, kk - dias_base_datos + dias_vista + 7: kk] - 1) / 6 #Día
de la semana
        xk_norm[3, 0] = (Dia_semana[0, kk + dias_vista] - 1) / 6

        predict_norm[0, kk + dias_vista], conver, aux_Hw, iter, = dr.regression(setx_norm,
sety_norm, xk_norm, gamma, H, opt)
        if not conver:
            print("No converge para este gamma o peso dado")
            break
        if aux_Hw > max_Hw:
            max_Hw = aux_Hw
        sum_iter = sum_iter + iter
        if iter > max_iter:
            max_iter = iter
            kkitermax = kk
        if kk == (dias_totales - dias_base_datos)//2:
            print("50%")

    opt.iter = True
    opt.tiempo_computo = True
    tfin = time() - tini
    if conver:
        predict = predict_norm * (demanda_max - demanda_min) + demanda_min
        error_abs = np.mean(np.abs(Demanda_total[0, dias_base_datos + dias_vista:] - predict[0,
dias_base_datos + dias_vista:]))
        set_er_rel = (Demanda_total[0, dias_base_datos + dias_vista:] - predict[0,
dias_base_datos + dias_vista:]) / predict[0, dias_base_datos + dias_vista:]
        error_rel = round(np.mean(np.abs(set_er_rel))*100, 3)
        if opt.iter:
            print(f"Iteraciones medias: {round(sum_iter/(dias_totales - dias_base_datos -
dias_vista), 3)}. Máxima iteración: {max_iter} en {kkitermax}")
        if opt.pond_similitud:
```

```
            print(f"Máximo peso: {max_Hw}")
        if opt.tiempo_computo:
            print(f"Tiempo total: {tfin}s")
        print(f"Error absoluto: {error_abs}. Error relativo: {error_rel}%")
        print("")


    x = np.arange(0, dias_totales)
    fig, axs = plt.subplots(2)

    axs[0].set_title("Demanda real y estimada. Error relativo = " + str(round(error_rel, 2)) +
"%")
    axs[0].set_ylabel("Demanda Acumulada(MW)")
    axs[0].set_xlabel("Días")
    axs[1].set_ylabel("Demanda Acumulada(MW)")
    axs[1].set_xlabel("Días")

    real, = axs[0].plot(x, Demanda_total[0, :], marker='o')
    estimada, = axs[0].plot(x, predict[0, :], marker='x')
    axs[0].legend([real, estimada], ["Real", "Estimada"])

    fact_enf = 321
    real_enfoque, = axs[1].plot(x[dias_totales - dias_base_datos - dias_vista - fact_enf:
dias_totales - fact_enf],
                                Demanda_total[0, dias_totales - dias_base_datos - dias_vista -
fact_enf: dias_totales

- fact_enf], marker='o')
    estimada_enfoque, = axs[1].plot(x[dias_totales - dias_base_datos - dias_vista - fact_enf:
dias_totales - fact_enf],
                                    predict[0, dias_totales - dias_base_datos - dias_vista -
fact_enf: dias_totales

- fact_enf], marker='x')
    axs[1].legend([real, estimada], ["Real", "Estimada"])

    plt.show()
```

6.6 Regresion_Covid.py

Este ejemplo es aún mejor para entender las posibilidades de la función *training_validation_test_regression* ya que se puede comparar distintas opciones como base de datos dinámica, división de los datos para unos porcentajes dados, probar las opciones de iteraciones medias, tiempo de cómputo medio…Como siempre, será importante crear una instancia de la clase *OPTIONS* para poder cambiar todas estas opciones.

6.6.1 Importaciones Necesarias

```
import pandas as pd
import numpy as np
import disimilarity_regression as dr
```

6.6.2 Cuerpo del Código

```
nacional_covid19 = pd.read_csv('nacional_covid19.csv', header=0)

datos_dataframe = pd.DataFrame([nacional_covid19['fecha'], nacional_covid19['hospitalizados'],
nacional_covid19['casos_pcr']]).T
datos_dataframe.columns = ['fecha','hosp', 'casos_total']

#Nos quitamos de en medio todos los NaN y convertimos a array
datos_dataframe.hosp = datos_dataframe.hosp.fillna(value = 0, limit = 2)
datos_dataframe.hosp = datos_dataframe.hosp.fillna(method = 'ffill')

datos_dataframe.fecha = datos_dataframe.fecha.fillna(value = 0, limit = 2)
datos_dataframe.fecha = datos_dataframe.fecha.fillna(method = 'ffill')

datos_dataframe.casos_total = datos_dataframe.casos_total.fillna(value = 0, limit = 2)
datos_dataframe.casos_total = datos_dataframe.casos_total.fillna(method = 'ffill')
```

```
#Creamos arrays propios con los datos
hospitalizados = np.array(datos_dataframe.hosp)
casos_total = np.array(datos_dataframe.casos_total)
dias_previos = 10

SETx = np.array([hospitalizados[66: -dias_previos]])
N = SETx[0,:].size
M = SETx[:, 0].size
min_x = np.min(SETx)
max_x = np.max(SETx)
SETx_norm = (SETx - min_x)/(max_x - min_x)

SETy = np.array([hospitalizados[66 + dias_previos: ]])
print(N)
SETy_norm = (SETy - min_x)/(max_x - min_x)
"""Normalizar los datos"""

opt = dr.OPTIONS()
opt.mostrar()
opt.tiempo_computo = False
opt.iter = False
opt.pond_similitud = True
opt.per_train_val_test = [25, 75, 0, True]
opt.max_error = 1e-3
opt.max_iter = 100000
opt.w = 0.00001
opt.D_dinamica = True
opt.Norm = True
dr.training_validation_test_regression(SETx, SETy, dias_previos, opt)
```

6.7 Fista.py

Esta librería, junto a **6.8 Fista_Based_Restart.py,** es la piedra angular de la optimización del TFG. Con ella resolveremos el problema de optimización convexa con un enfoque dual. Se divide en dos funciones. La primera *lambda_opt* calcula las componentes óptimas de λ_{β_k} para una β dada con las matrices de pesos Γ_w y H_w. La segunda resuelve el problema de optimización planteado en **3.1 Problema a minimizar** devolviendo el λ^* para una precisión que hayamos definido.

6.7.1 Importaciones Necesarias

```
import numpy as np
```

6.7.2 *lambda_opt*

```
    def lambda_opt(gamma, c, h):
    # VI) Cálculo de Lambda en función de Beta. Problema desacoplado
    N = c[:, 0].size
    L = np.zeros([N, 1])
    gam = 0
    for kk in range(0, N):
        gam = gamma[0, kk]
        if (c[kk, 0] <= gam) and (c[kk, 0] >= -gam):
            L[kk, 0] = 0
        elif c[kk, 0] > 0:
            L[kk, 0] = 0.5*(gam - c[kk, 0]) / h[kk, kk]
        else:
            L[kk, 0] = -0.5*(gam + c[kk, 0]) / h[kk, kk]
    return L
```

6.7.3 *algorithm*

```
def algorithm(R, b, H, gamma, x_ini, max_iter, max_error):
    N = R[0, :].size
    M = R[:, 0].size
```

```
    if not isinstance(gamma, (list, np.ndarray)): #Convertimos gamma en array si no lo es
        gamma = np.ones([1, N]) * gamma
    convergencia = False
    iter = 0
    """Precalculamos la inversa de H, R * H_inv * R.T, y  (inv(R_Hinv_R_T)) @ (R@l_opt - b) en 
inv(R_Hinv_R_T) @ R @ l_opt -
        -R_Hinv_R_T @ b y de aquí reducimos a inv(R_Hinv_R_T)_R @ l_opt - inv(R_Hinv_R_T)_b.
Reducimos una multiplicación"""
    H_inv = np.zeros([N, N])
    for k in range(0, N): #Así se calcula la inversa de una matriz diagonal mucho más rápido
        H_inv[k, k] = 1 / H[k, k]

    inv_R_Hinv_R_T = np.linalg.inv(R @ H_inv @ R.T)
    inv_R_Hinv_R_T_R = inv_R_Hinv_R_T @ R
    inv_R_Hinv_R_T_b = inv_R_Hinv_R_T @ b
    """ALGORITMO FISTA"""
    # I) El parametro de tolerancia viene definido por max_error
    # II) Inicialización de Beta, del vector sol lambda y de [xk, xk-1]. kk = 0.
    B = np.zeros([M, 1])
    l_opt_B = np.zeros([N, 1]) #Lambda optima para una beta dada.
    xk_1 = x_ini  # xk-1
    xk = x_ini  # xk
    """III) """
    tk_1 = 0
    tk = 0

    for kk in range(0, max_iter):
        # IV) Cálculo  del nuevo tiempo de integración
        tk = 0.5 * (1 + np.sqrt(1 + 4*tk_1**2))
        # V) Cálculo de Beta en función de [tk, tk_1] y [xk, xk_1]
        B = xk + ((tk_1 - 1) / tk) * (xk - xk_1)
        # VI) Cálculo del labmda óptimo en función de Beta
        l_opt_B = lambda_opt(gamma, R.T @ B, H)
        # VIII) Si no se supera el error máximo --> Exit. Este paso va a antes de VII) por 
eficiencia.
        iter = kk

        Rlopt_b = R @ l_opt_B - b
        norm_Rlopt_b = np.linalg.norm(Rlopt_b)

        if norm_Rlopt_b < max_error:
            convergencia = True
            break
        # VII) Recálculo de xk y xk_1 realizando una única multiplicación de matrices
        xk_1 = xk
        tk_1 = tk
        xk = B + 2*(inv_R_Hinv_R_T_R @ l_opt_B - inv_R_Hinv_R_T_b)

    J = l_opt_B.T @ H @ l_opt_B + gamma @ np.abs(l_opt_B)

    return [l_opt_B, convergencia, iter, J, xk]
```

6.8 Fista_Based_Restart.py

Como hemos visto a lo largo del TFG, el método FISTA puede tener algunos problemas prácticos de oscilación. Si identificamos que nuestro problema está mal condicionado o verificamos que la solución oscila, deberíamos importar la librería de *Fista_Based_Restart.py* ya que implementa el mismo algoritmo FISTA con la estrategia de reinicio definida en **3.5.3 Problema práctico *Rippling*.**

6.8.1 Importaciones necesarias

```
import numpy as np
from math import e
from time import time
```

6.8.2 *lambda_opt*

```
def lambda_opt(gamma, c, h):
    # VI) Cálculo de Lambda en función de Beta. Problema desacoplado
    N = c[:, 0].size
```

```
    L = np.zeros([N, 1])
    gam = 0
    for kk in range(0, N):
        gam = gamma[0, kk]
        if (c[kk, 0] <= gam) and (c[kk, 0] >= -gam):
            L[kk, 0] = 0
        elif c[kk, 0] > 0:
            L[kk, 0] = 0.5*(gam - c[kk, 0]) / h[kk, kk]
        else:
            L[kk, 0] = -0.5*(gam + c[kk, 0]) / h[kk, kk]
    return L
```

6.8.3 *MaxCotInf*

```
def MaxCotInf(s, H, R, inv_R_Hinv_R_T_R, inv_R_Hinv_R_T_b, gamma):
    c = R.T @ s
    l_opt = lambda_opt(gamma, c, H)
    delta_S = 2*(inv_R_Hinv_R_T_R @ l_opt - inv_R_Hinv_R_T_b)
    return delta_S, l_opt
```

6.8.4 *algorithm*

```
def algorithm(R, b, H, gamma, x_ini, max_iter, max_error):
    N = R[0, :].size
    M = R[:, 0].size
    if not isinstance(gamma, (list, np.ndarray)): #Convertimos gamma en array si no lo es
        gamma = np.ones([1, N]) * gamma
    convergencia = False
    iter = 0
    """Precalculamos la inversa de H, R * H_inv * R.T, y  (inv(R_Hinv_R_T)) @ (R@l_opt - b) en
inv(R_Hinv_R_T) @ R @ l_opt -
        -R_Hinv_R_T @ b y de aquí reducimos a inv(R_Hinv_R_T)_R @ l_opt - inv(R_Hinv_R_T)_b.
Reducimos una multiplicación"""
    H_inv = np.zeros([N, N])
    for k in range(0, N): #Así se calcula la inversa de una matriz diagonal mucho más rápido
        H_inv[k, k] = 1 / H[k, k]

    inv_R_Hinv_R_T = np.linalg.inv(R @ H_inv @ R.T)
    inv_R_Hinv_R_T_R = inv_R_Hinv_R_T @ R
    inv_R_Hinv_R_T_b = inv_R_Hinv_R_T @ b
    """ALGORITMO FISTA"""
    # I) El parametro de tolerancia viene definido por max_error
    # II) Inicialización de Beta, del vector sol lambda y de [xk, xk-1]. kk = 0.
    B = np.zeros([M, 1])
    l_opt_B = np.zeros([N, 1]) #Lambda optima para una beta dada.

    """ II) """
    tk_1 = 1
    j = 0
    z0 = x_ini
    """ III) """
    delta_Z, l_opt_z = MaxCotInf(z0, H, R, inv_R_Hinv_R_T_R, inv_R_Hinv_R_T_b, gamma)
    """ IV) """
    pj = np.linalg.norm(R @ l_opt_z - b)
    """ V) """
    xk_1 = z0 + delta_Z
    B = z0 + delta_Z
    """ VI) """
    error_acumulado = np.zeros([1, max_iter])
    t_acumulado = np.zeros([1, max_iter])
    t_ini = time()
    for kk in range(0, max_iter):
        """ VIII) """
        delta_B, l_opt_B = MaxCotInf(B, H, R, inv_R_Hinv_R_T_R, inv_R_Hinv_R_T_b, gamma)
        """ IX) """
        xk = B + delta_B
        """ X) """
        tk = 0.5 * (1 + np.sqrt(1 + 4*tk_1**2))
        """ XI) """
        B = xk + ((tk_1 - 1) / tk) * (xk - xk_1)*1
        """ XII) """
        if np.linalg.norm(R @ l_opt_B - b) <= (pj/e):
            j = j + 1 #Por si interesa saber cuantas veces se ha reiniciado
            zj = B
            tk_1 = 1
            delta_Z, l_opt_z = MaxCotInf(zj, H, R, inv_R_Hinv_R_T_R, inv_R_Hinv_R_T_b, gamma)
            pj = np.linalg.norm(R @ l_opt_z - b)
```

```
            xk_1 = zj + delta_Z
            B = xk_1
        else:
            xk_1 = xk
            tk_1 = tk
        iter = kk
        error_acumulado[0, kk] = np.linalg.norm(R @ l_opt_B - b)
        if pj < max_error:
            convergencia = True
            break
        t_acumulado[0, kk] = time() - t_ini

    J = l_opt_z.T @ H @ l_opt_z + gamma @ np.abs(l_opt_z)

    return [l_opt_z, convergencia, iter, J, B]
```

6.9 Momentum.py

Este archivo es de laboratorio. Es decir, sirve para poder probar suposiciones sobre FISTA, la relación entre las oscilaciones y la variación del *momentum* y en definitiva devolver una gráfica con hasta 8 *mometums* distintos para un mismo algoritmo y problema.

6.9.1 Importaciones Necesarias

```
import numpy as np
import matplotlib.pyplot as plt
import FISTA_v5 as fista
from time import time
```

6.9.2 Cuerpo del Código

```
def algorithm(R, b, H, gamma, x_ini, max_iter, max_error, mod_momentum):
    N = R[0, :].size
    M = R[:, 0].size
    convergencia = False
    iter = 0
    """Precalculamos la inversa de H, R * H_inv * R.T, y  (inv(R_Hinv_R_T)) @ (R@l_opt - b) en
inv(R_Hinv_R_T) @ R @ l_opt -
        -R_Hinv_R_T @ b y de aquí reducimos a inv(R_Hinv_R_T)_R @ l_opt - inv(R_Hinv_R_T)_b.
Reducimos una multiplicación"""
    H_inv = np.zeros([N, N])
    for k in range(0, N): #Así se calcula la inversa de una matriz diagonal mucho más rápido
        H_inv[k, k] = 1 / H[k, k]

    inv_R_Hinv_R_T = np.linalg.inv(R @ H_inv @ R.T)
    inv_R_Hinv_R_T_R = inv_R_Hinv_R_T @ R
    inv_R_Hinv_R_T_b = inv_R_Hinv_R_T @ b
    """ALGORITMO FISTA"""
    # I) El parametro de tolerancia viene definido por max_error
    # II) Inicialización de Beta, del vector sol lambda y de [xk, xk-1]. kk = 0.
    B = np.zeros([M, 1])
    l_opt_B = np.zeros([N, 1]) #Lambda optima para una beta dada.
    xk_1 = x_ini  # xk-1
    xk = x_ini  # xk

    """III) Momentum"""
    tk_1 = 2.194
    gamma_w = np.ones([1, N])
    error_acumulado = np.zeros([1, max_iter])
    t_acumulado = np.zeros([1, max_iter])
    t_ini = time()
    for kk in range(0, max_iter):
        # IV) Cálculo  del nuevo tiempo de integración
        tk = 0.5 * (1 + np.sqrt(1 + 4*tk_1**2))
        thetak = (tk_1 - 1) / tk
        if mod_momentum == 2:
            thetak = 0.5
        # V) Cálculo de Beta en función de [tk, tk_1] y [xk, xk_1] probar con 0.902
        B = xk + thetak * (xk - xk_1)*mod_momentum
        # VI) Cálculo del labmda óptimo en función de Beta
        l_opt_B = fista.lambda_opt(gamma*gamma_w, R.T @ B, H)
```

```
        # VIII) Si no se supera el error máximo --> Exit. Este paso va a antes de VII) por
eficiencia.
        iter = kk
        Rlopt_bAux = np.linalg.norm(R @ l_opt_B - b)
        Rlopt_b = Rlopt_bAux
        error_acumulado[0, kk] = Rlopt_b
        # VII) Recálculo de xk y xk_1 realizando una única multiplicación de matrices
        tk_1 = tk
        xk_1 = xk
        xk = B + inv_R_Hinv_R_T_R @ l_opt_B - inv_R_Hinv_R_T_b
        t_acumulado[0, kk] = time() - t_ini
    return [l_opt_B, convergencia, iter, error_acumulado, t_acumulado]


SETxini = np.array([[...]
,[...]
,[...]])

SETx = np.log(SETxini)
# SETx = np.array([np.arange(0, 200 + 0.15, 0.15)])
# N = 200
# SETx = np.array(np.random.rand(1, N))**2

N = SETx[0, :].size

R = np.append(SETx, np.ones([1, N]), axis=0)
b = np.array([[0.91574], [5.6987], [4.8765]])
#b = np.array([[0.91574]])
r = np.append(b, np.ones([1, 1]), axis=0)
print(f"N: {N}")

H = np.eye(N)*0.5

gamma = 1
max_iter = 100000
max_error = 1e-9

max_error = 1e-20
max_iter = 150
error_acumulado = np.zeros([8, max_iter])
momentum = [2, 1, 0.8, 0.62, 0.5, 0.3, 0]
[_, _, _, error_acumulado[0, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[0])

[_, _, _, error_acumulado[1, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[1])

[_, _, _, error_acumulado[2, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[2])

[_, _, _, error_acumulado[3, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[3])

[_, _, _, error_acumulado[4, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[4])

[_, _, _, error_acumulado[5, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[5])

[_, _, _, error_acumulado[6, :], _] = algorithm(R, r, H, gamma, np.zeros([4, 1]), max_iter,
max_error, momentum[6])

plt.figure(figsize=(15, 7))
plt.figure(1)
plt.title('Variación del mometum')
iter_fis = np.arange(0, max_iter)
y0 = plt.plot(iter_fis, error_acumulado[0, :], 'b-')[0]
y1 = plt.plot(iter_fis, error_acumulado[1, :], 'g-')[0]
y2 = plt.plot(iter_fis, error_acumulado[2, :], 'r-')[0]
y3 = plt.plot(iter_fis, error_acumulado[3, :], 'c-')[0]
y4 = plt.plot(iter_fis, error_acumulado[4, :], 'm-')[0]
y5 = plt.plot(iter_fis, error_acumulado[5, :], 'y-')[0]
y6 = plt.plot(iter_fis, error_acumulado[6, :], 'k-')[0]


plt.yscale('log')
plt.ylim([10**-11, 2*10**1])
plt.xlabel("Iteraciones")
plt.ylabel("||f(xk) - fopt||")
```

```
plt.legend([y0, y1, y2, y3, y4, y5, y6], ["1    ", str(momentum[1])+"       · thetak_1 ",
str(momentum[2])+"    · thetak_1", str(momentum[3])+" · thetak_1", str(momentum[4])+"    ·
thetak_1", str(momentum[5])+"    · thetak_1",  "0"])
plt.show()
```

REFERENCIAS

[1] A. D. Carnerero, D. R. Ramirez y T. Alamo, «Probabilistic interval predictor based on dissimilarity functions,» *arXiv preprint arXiv:2010.15530,* 2020.

[2] T. Alamo, J. M. Bravo y E. F. Camacho, «Guaranteed state estimation by zonotopes,» *Automatica,* vol. 41, nº 6, pp. 1035-1043, 2005.

[3] T. Alamo, J. M. Bravo, M. J. Redondo y E. F. Camacho, «A set membership state estimation algorithm based on DC programming,» *Automatica,* vol. 44, nº 1, pp. 216-224, 2008.

[4] R. E. H. Thabet, T. Raïssi, C. Combastel, D. Efimov y A. Zolghadri, «An effective method to interval observer design for time-varying systems,» *Automatica,* vol. 50, nº 10, pp. 2677-2684, 2014.

[5] S. Chebotarev, D. Efimov, T. Raïsse y A. Zolghadri, «Interval observers for continuos-time LPV systems with l1/l2 performace,» *Automatica,* vol. 58, pp. 82-89, 2015.

[6] J. Roll, A. Nazin y L. Ljung, «Non-linear System Identification Via Direct Weight Optimization,» *Automatica,* vol. 41, nº 3, pp. 475-490, 2005.

[7] J. M. Bravo, M. Vasallo, M. E. Gegundez y D. Marin, «Combined stochastic and deterministic interval predictor for time-varying systems,» de *23th Mediterranean Conference on Control and Automation (MED)*, 2015, pp. 833-839.

[8] A. Beck y M. Teboulle, «A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems,» *SIAM Journal on Imaging Sciences,* vol. 2, nº 1, pp. 183-202, 2009.

[9] T. Alamo, P. Krupa y D. Limon, «Gradient Based Restart FISTA,» de *2019 IEEE 58th Conference on Decision and Control (CDC)*, 2019, pp. 3936-3941, doi: 10.1109/CDC40024.2019.9029983.

[10] «Red Eléctrica de España,» [En línea]. Available: https://www.ree.es/es/datos/publicaciones/series-estadisticas-nacionales. [Último acceso: 14 08 2021].

[11] Datadista, «GitHub,» [En línea]. Available: https://github.com/datadista/datasets/tree/master/COVID%2019. [Último acceso: 09 2021].

[12] Y. Chen, *Accelerated gradient methods,* Princeton, 2019.

[13] U. Von Luxburg, *Statistical learning with similarity and dissimilarity functions,* Berlin, 2004.

[14] L. Vandenberghe, «Fast Proximal Gradient Method,» 2014. [En línea]. Available: http://www.seas.ucla.edu/~vandenbe/236C/lectures/fista.pdf. [Último acceso: 08 2021].