Práctica 1 - Memoria

Sergio Quijano Rey

5 de abril de 2021

Índice

1. Ejercicio 1 - Búsqueda iterativa de óptimos

1.1. Apartado 1

En este apartado se nos pide implementar el algoritmo de Gradiente Descendente. Esta función toma como parámetros de entrada:

- starting_point : punto inicial del que parte la búsqueda
- loss_function : función de error que buscamos minimizar. En nuestro caso concreto, es una función real de dos variables
- gradient: función vectorial 2-dimensional que representa el gradiente de loss_function
- learning_rate : la tasa de aprendizaje. Es el parámetro crítico, pues en esta sección nos centramos en estudiar el comportamiento del gradiente descendente en base a este parámetro (y también del starting_point)
- max_iterations: número máximo de iteraciones. Así tenemos una condición sufiente para saber que el algoritmo va a parar
- target_error : error debajo del cual paramos de iterar
- verbose: si es True, la función devuelve el error de cada solución de las iteraciones

Esta función devuelve el vector de soluciones que hemos ido construyendo. Si queremos obtener la solución final, lo único que tenemos que hacer es acceder a la última posición. Así podemos generar las gráficas en las que plasmamos tanto la función de error como los puntos que vamos construyendo. También, si tenemos activado el parámetro Verbose, devolvemos el error en cada iteración, lo que es necesario para hacer las gráficas de evolución de error que se nos piden.

El procedimiento del algoritmo es muy simple. Partiendo de la solución inicial, avanzamos el punto en la dirección opuesta al gradiente, puesto que el gradiente apunta en la dirección de máximo ascenso, y nosotros estamos haciendo minimización. Además, cada paso marcado por el gradiente lo ponderamos por el learning_rate . Todo esto se resume en la fórmula:

$$w_{new} = w_{old} - \eta * \nabla Err(w_{old})$$

donde ∇Err es la función vectorial gradiente, w_{new}, w_{old} son los vectores peso que definen la solución y η es el learning_rate.

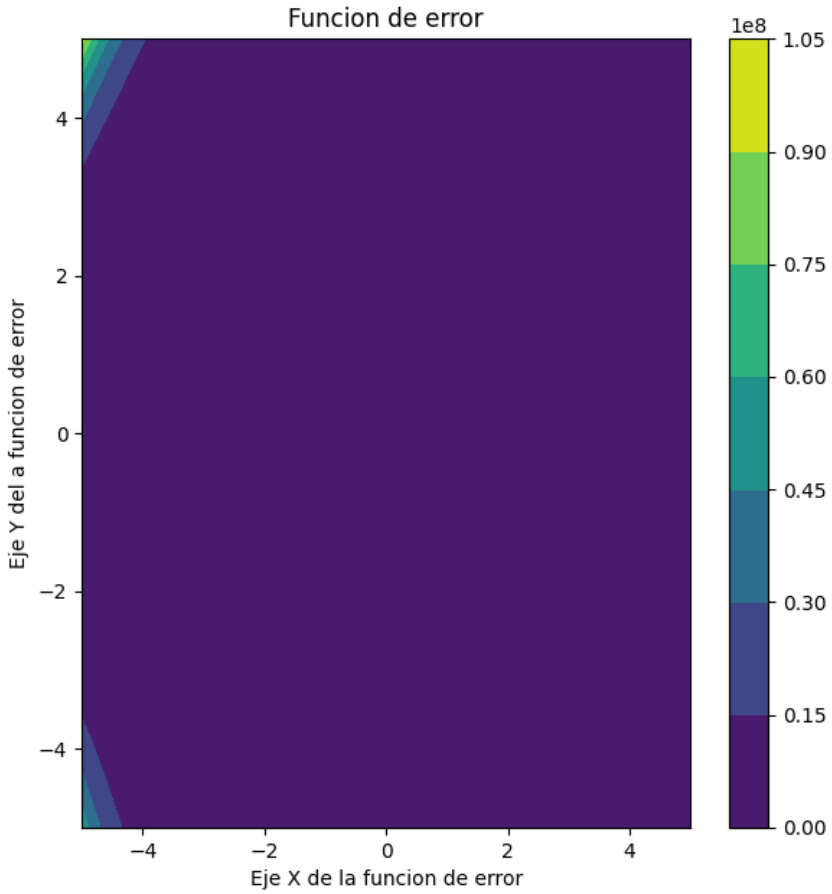
1.2. Apartado 2

Consideramos la función de error dada por:

$$E(u, v) := (u^3 * e^{v-2} - 2v^2 e^{-u})^2$$

Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u, v) = (1, 1)$ y usando una tasa de aprendizaje $\eta = 0, 1$.

Podemos mostrar la función de error en una vista de pájaro (la vista tridimensional solo será necesaria en casos en los que la vista de pájaro no sea suficiente). Preferimos esta vista bidimensional porque es más rápida de computar y en muchos casos más fácil de interpretar.



Los colores más azulados y oscuros indican los valores más bajos del error, mientras que valores verdosos y claros indican valores altos del error (la leyenda de la gráfica ya indica esto). Así que nuestros puntos deberán ir acercándose a zonas oscuras de la gráfica, si el comportamiento de los algoritmos es bueno.

1.2.1. Subapartado a

Calculamos analíticamente la expresión de las derivadas parciales:

$$\frac{\partial E}{\partial u} = 2 * (u^3 * e^{v-2} - 2v^2 e^{-u}) * (3u^2 e^{v-2} + 2v^2 e^{-u})$$

$$\frac{\partial E}{\partial v} = 2 * (u^3 * e^{v-2} - 2v^2 e^{-u}) * (u^3 * e^{v-2} - 4v e^{-u})$$

$$\nabla E = (\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v})$$

Esta es la expresión del gradiente que usamos en el código. Expresión que mostramos por pantalla, como se indica en el guión de la práctica.

1.2.2. Subapartados b y c

Se manda explícitamente que usemos flotantes de 64 bits, para lo cual usamos la orden np.float64 para devolver todas nuestras funciones que representan los errores y las derivadas parciales.

Como se muestra en el código, solo necesitamos 10 iteraciones para quedarnos por debajo de un error (o valor de $E(u, v)$, que es lo que estamos considerando como función de error a minimizar) de valor 10^{-14}. En el código indicamos que la primera iteración por debajo del error es la iteración 9, pero hay que tener en cuenta que empezamos a contar desde el cero.

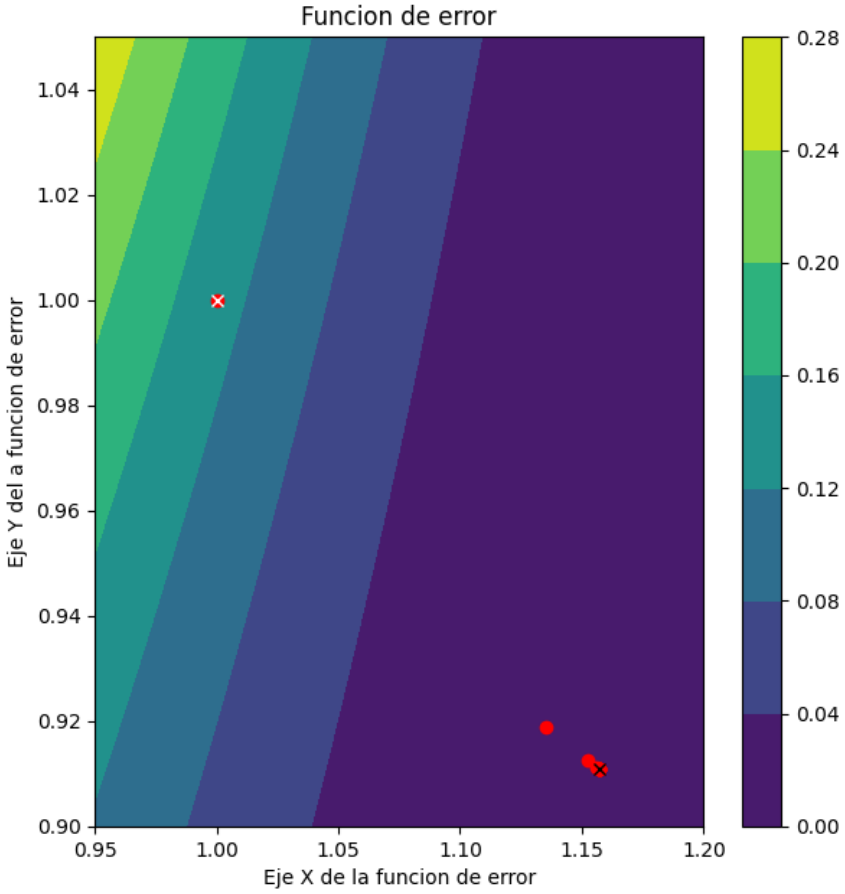
Los resultados que mostramos por pantalla nos indican que alcanzamos la solución:

```
Numero de iteraciones: 10
Pesos encontrados: [1.15728885 0.91083837]
```

Estos son los resultados vinculados a la onceava iteración (recordar que empezamos a contar desde el cero). Los resultados asociados a la décima iteración, en los que ya estamos por debajo del error buscado, son:

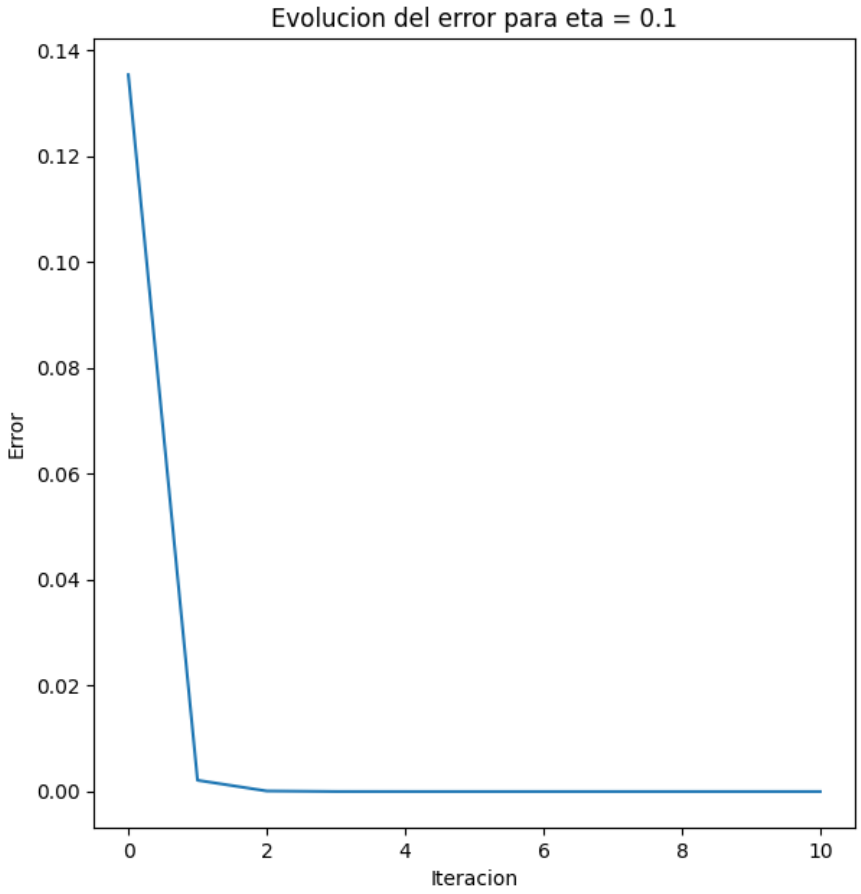
```
Primera iteracion por debajo de 10e-14: 9 (contando desde cero)
Las primeras coordenadas por debajo del error: [1.15728875 0.9108384 ]
```

Todo esto se muestra en las salidas por pantalla de nuestro programa. Además, podemos mostrar cómo avanzan nuestras soluciones sobre la superficie que representa la función de error:



El punto con una cruz blanca es el punto inicial. El punto con una cruz negra es el punto final de la búsqueda. Viendo la gráfica de las soluciones generadas es claro que podríamos seguir avanzando con la búsqueda, pero decidimos parar porque estamos por debajo de la cota de error pedida.

La gráfica del error para los parámetros que se nos han indicado en 1.2 es la siguiente:


Evolucion del error para eta = 0.1
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Error
0
2
4
6
8
10
Iteracion

1.3. Apartado 3

Se considera ahora la función:

$$f(x,y) := (x+2)^2 + 2(y-2)^2 + 2sin(2\pi x)sin(2\pi y)$$

Como vamos a usar la técnica del gradiente descendente, necesitamos calcular analíticamente la expresión del gradiente:

$$\frac{\partial f(x,y)}{\partial x} = 2(x+2) + 4\pi sin(2\pi y)cos(2\pi x)$$

$$\frac{\partial f(x,y)}{\partial y} = 4(y-2) + 4\pi sin(2\pi x)cos(2\pi y)$$

$$\nabla f(x,y) = (\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y})$$

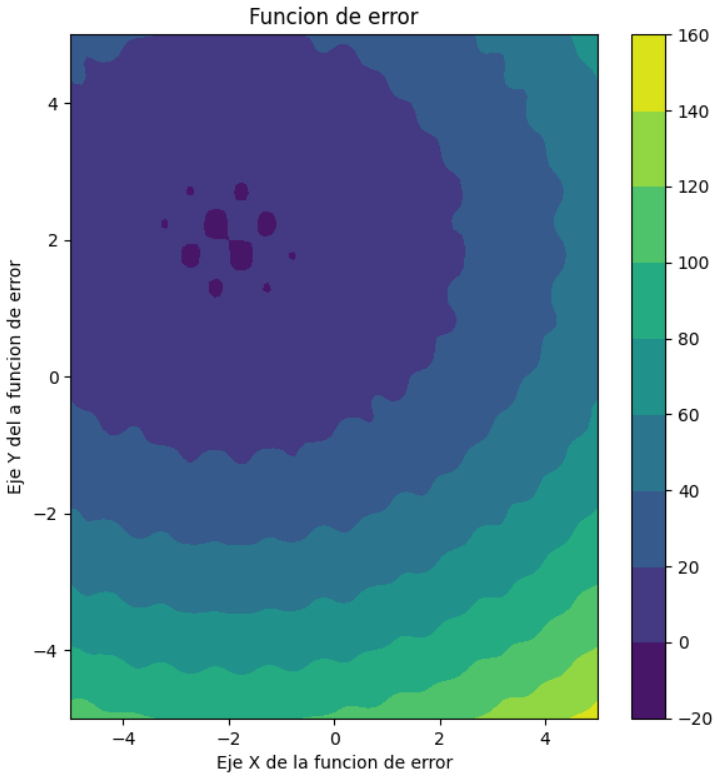
1.3.1. Subapartado a

Se nos pide usar gradiente descendiente para minimizar la función de error f. Además, se nos pide que usemos como parámetros del gradiente descendente:

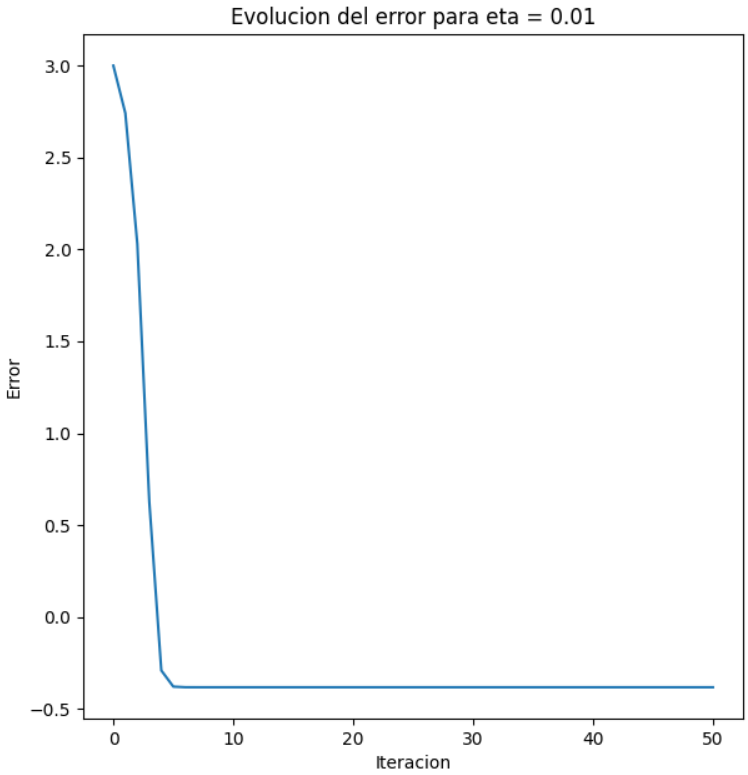
- $\eta = 0{,}01$
- $x_0 = -1, y_0 = 1$
- Máximo 50 iteraciones

Y que repitamos el experimento con un valor de $\eta = 0{,}1$.

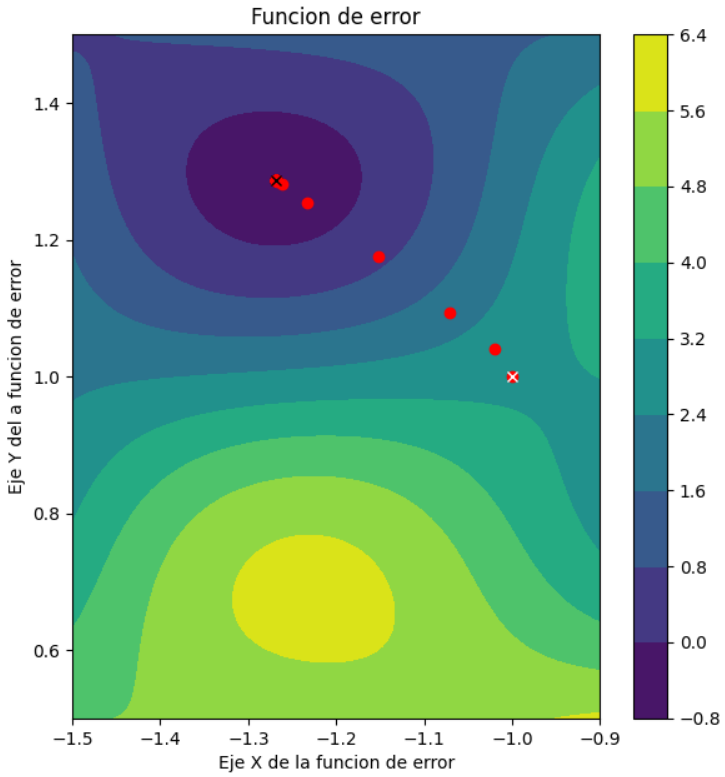
Lo primero que hacemos es mostrar una gráfica en dos dimensiones de la función de error que queremos minimizar:



Realizando gradiente descendente con los primeros parámetros dados obtenemos la siguiente gráfica, en la que se muestra cómo evoluciona el valor del error según avanzan las iteraciones del algoritmo.

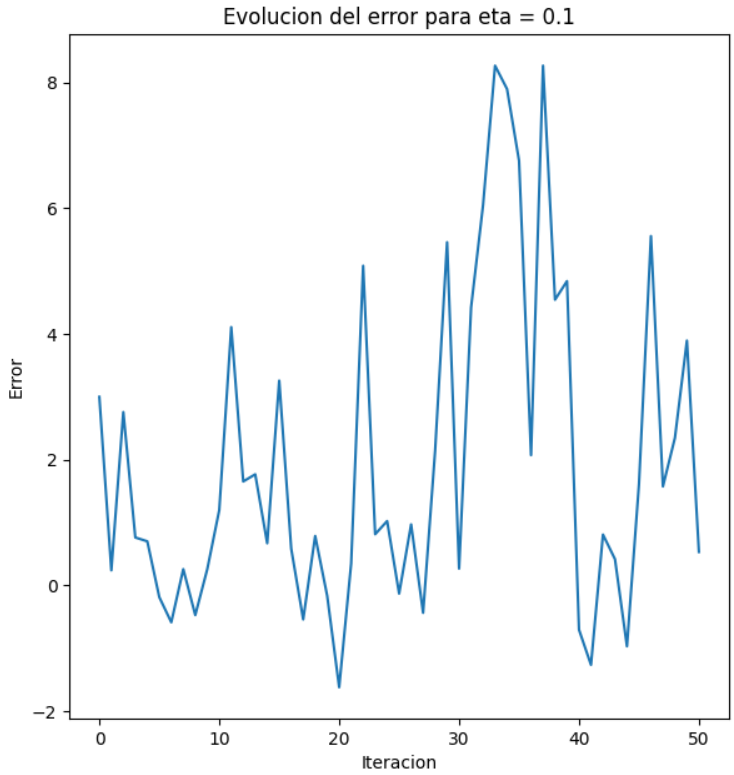


Mostramos, aunque no se nos pida en el guión, la traza del algoritmo (soluciones plasmadas sobre la gráfica bidimensional del error):

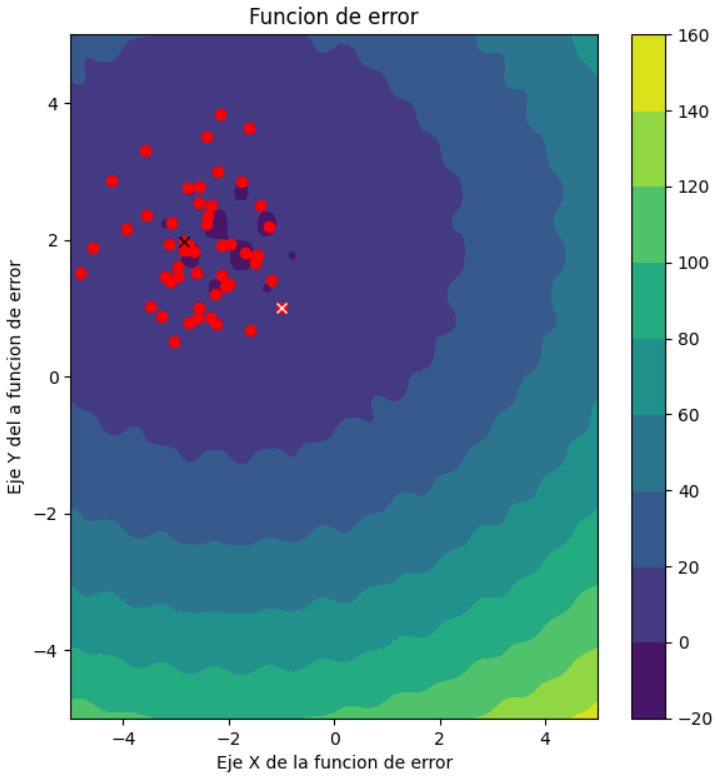


En esta gráfica se puede apreciar con facilidad que el algoritmo tiene el comportamiento deseado, pues cae con facilidad en un óptimo local.

Ahora realizamos el mismo proceso pero cambiando a $\eta = 0{,}1$. Obtenemos la siguiente gráfica de evolución del error:



De esta gráfica ya podemos ver que el comportamiento del algoritmo no es el deseado. Lo que esperaríamos es que el error disminuyese monótamente en cada iteración. Sin embargo, lo que nos encontramos es con que el error oscila con grandes saltos. De lo estudiado en teoría, sabemos que el motivo sea seguramente un valor de η demasiado grande. Con esto, cuando estamos cerca de una solución local, damos un salto muy grande en cierta dirección, saltando el óptimo local y yendo a una zona de alto error. Del mismo modo, de un salto desde una zona de alto error podemos acabar en una zona de muy bajo error. Es claro que este comportamiento errático no es deseable. Mostramos la traza de las soluciones en cada iteración para reforzar esta hipótesis sobre el mal comportamiento:



Con este gráfico, vemos que en la última iteración (del gráfico de evolución de error se deduce ya que se han consumido el total de iteraciones) hemos acabado, por buena suerte, en una buena zona de la función del error.

Con este ejemplo, podemos ver que es importante explorar con distintos valores de η, pues estos influyen mucho en el buen o mal comportamiento del gradiente descendiente. Es más, ya podemos intuir que un ajuste dinámico del valor de η puede ser muy interesante, pues como hemos visto en teoría, nos interesa avanzar con *"pasos de gigante"* al principio con un η grande, y disminuir el valor de η al acercarnos a un óptimo local para tener más precisión en las iteraciones.

En estos dos ejemplos no muestro los valores numéricos de las soluciones obtenidas ni del error alcanzado. Considero que es mucho más interesante

e informativo visualizar las gráficas, tanto del error como de la traza, pues estamos estudiando el comportamiento del algoritmo, no una solución concreta que no nos interesa al no ser un problema que realmente queramos resolver.

1.3.2. Subapartado b

Ahora repetiremos el mismo experimento pero tomando distintos valores para las soluciones iniciales. Y en este caso se pide que generemos una tabla con los valores obtenidos. Como no se especifica nada, tomamos $\eta \in 0{,}01, 0{,}1$ y un máximo de 1000 iteraciones. Dejamos la cota de error a None para que no se hagan comprobaciones de cotas. Además de la tabla, mostraremos las gráficas de la evolución del error y la traza de soluciones en los casos en los que sea interesante comentarlas (no en todas pues resultaría repetitivo).

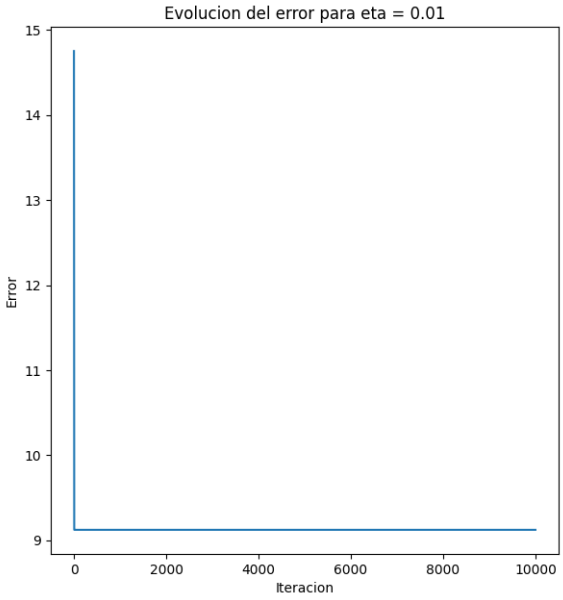
La tabla con los datos generados es:

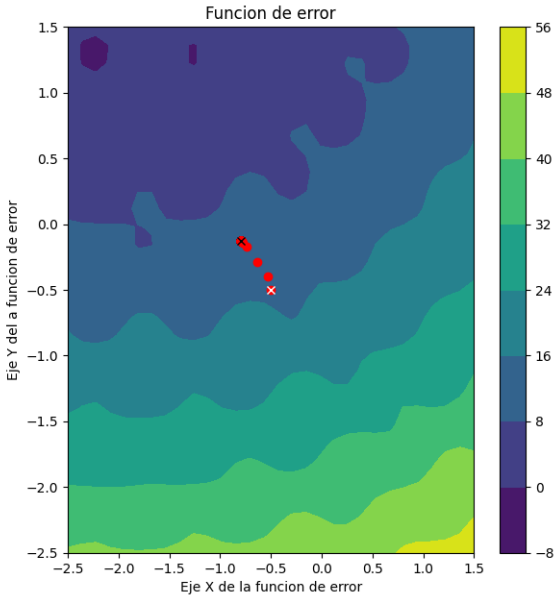
Punto Inicial	Learning Rate η	Solución alcanzada	Error Alcanzado
(-0.5, -0.5)	0.01	(-0.7934, -0.1259)	9.1251
(-0.5, -0.5)	0.1	(-1.1000, 3.4598)	4.7785
(1, 1)	0.01	(0.6774, 1.2904)	6.4375
(1, 1)	0.1	(-2.3820, 2.0917)	-0.5734
(2, 1)	0.01	(1.6466, 1.2954)	12.7624
(2, 1)	0.1	(-1.8963, 2.1082)	0.7969
(-2, 1)	0.01	(-2.2436, 1.2864)	-0.8685
(-2, 1)	0.1	(-0.8079, 2.0814)	2.3492
(-3, 3)	0.01	(-2.7309, 2.7132)	-0.3812
(-3, 3)	0.1	(-2.1558, 3.0127)	1.9427
(-2, 2)	0.01	(-2, 2)	$-4{,}7992 * 10^{-31}$
(-2, 2)	0.1	(-1.3420, 0.6155)	5.3779

Esta tabla ha sido obtenida a partir de los datos mostrados por pantalla. El programa no genera está tabla formateada, pero muestra por pantalla todos los valores que esta recoge.

1.3.3. Comentarios sobre los resultados obtenidos

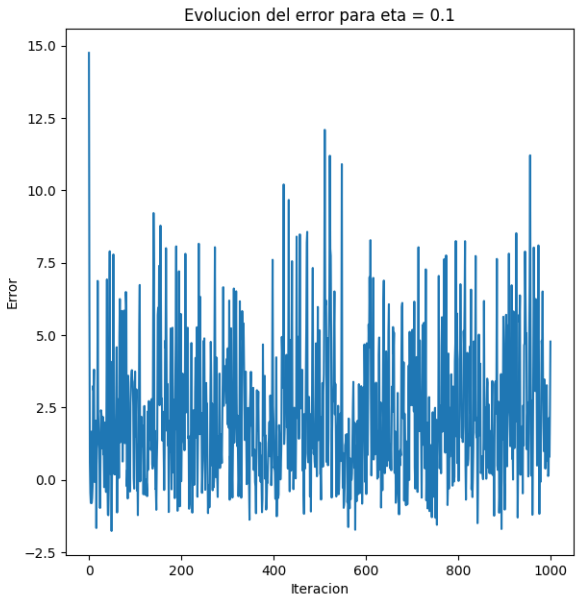
Para los primeros valores de la tabla tenemos las siguientes dos gráficas:





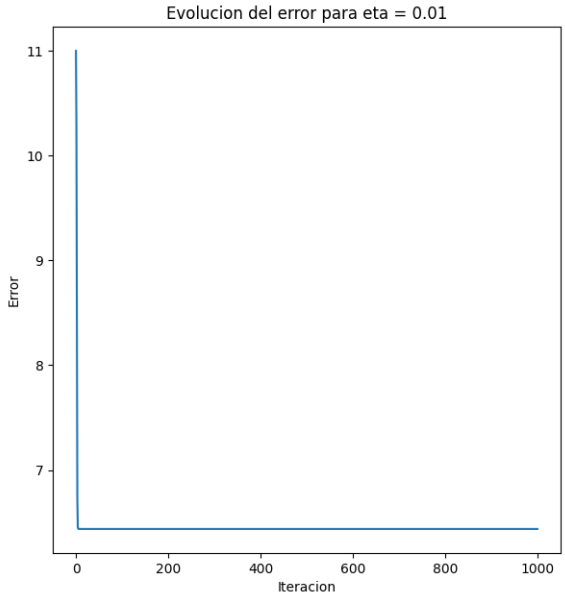
Vemos que damos un primer salto grande en el descenso del error, pero luego nuestro valor de η es demasiado pequeño para converger a un óptimo local. La gráfica en tres dimensiones que generamos no aporta ninguna información de interés.

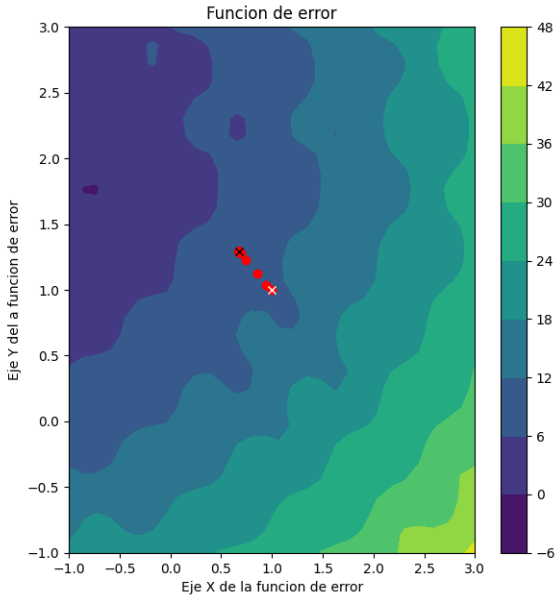
Para la segunda fila de la tabla, tenemos la siguiente gráfica de evolución del error:



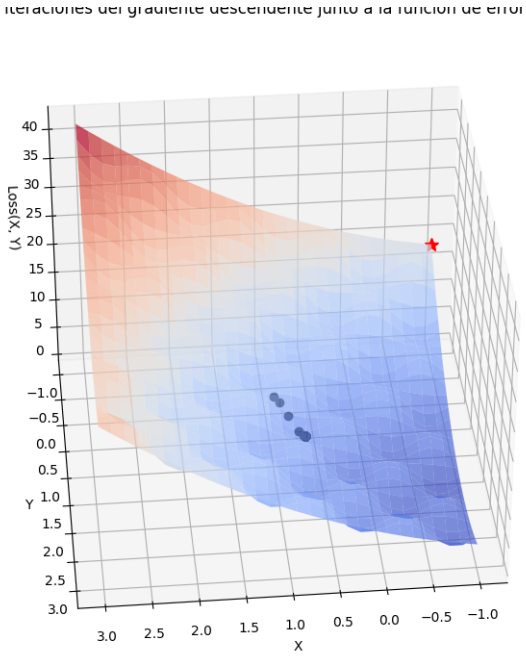
El valor de η es demasiado grande, por lo tanto, el error fluctúa tanto. No es un valor apropiado para η, por lo que hemos comentado en apartados anteriores. Y a pesar de este comportamiento errático, por casualidad, acabamos con un error menor que con el valor apropiado para η que aparece en la primera fila de la tabla.

Para la tercera fila de la tabla, parece que nos vuelve a pasar lo mismo que para la primera fila, a vista de las siguientes dos gráficas:


Evolucion del error para eta = 0.01
Error
Iteracion
11
10
9
8
7
0
200
400
600
800
1000


Funcion de error
Eje Y del a funcion de error
Eje X de la funcion de error
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
48
42
36
30
24
18
12
6
0
−6

Pero si nos fijamos en la siguiente gráfica:



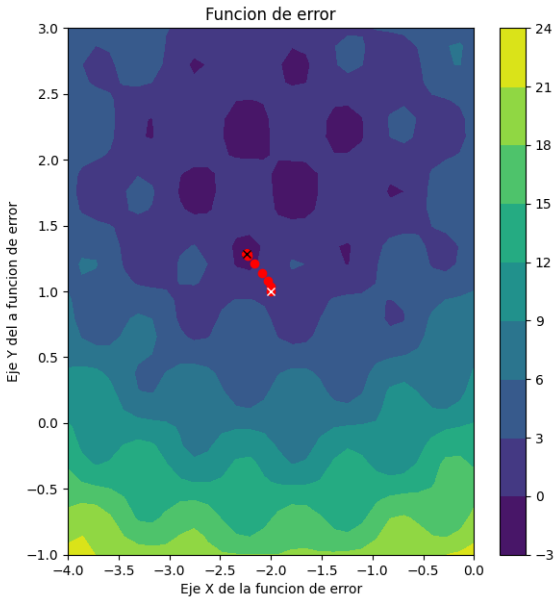
Lo que podemos ver es que nos quedamos en una especie de *bache* sobre el terreno. La conclusión es la misma, el valor de η tan pequeño nos deja atascado, pero ahora es más claro el motivo. No es que simplemente los 1000 *pasos* que damos sean muy cortos, es que llegamos a un óptimo local que con el pequeño valor de η no conseguimos superar.

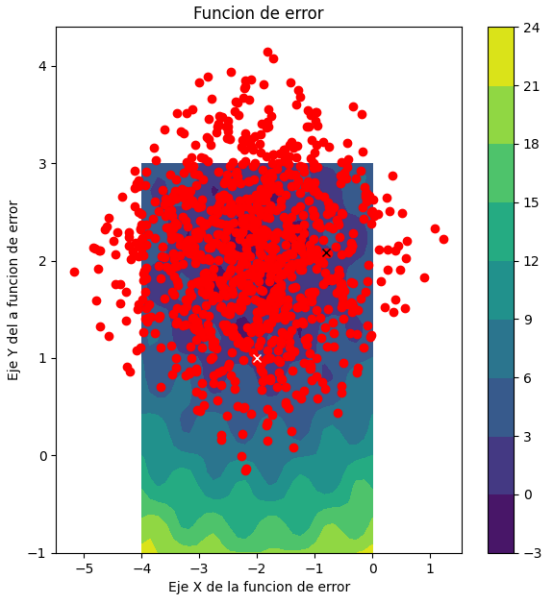
Para la cuarta fila, con un $\eta = 0{,}1$, tenemos un valor demasiado grande, lo que produce un comportamiento errático. Y de nuevo, por pura casualidad, alcanzamos un mejor valor de del error que usando un valor de η adecuado. No incluyo gráficas pues son muy parecidas a las ya mostradas cuando ocurre esto, para evitar ser repetitivo.

Para la quinta fila, tenemos el mismo comportamiento comentado en el caso en el que las soluciones se quedaban atascadas en un *bache*. Mientras que para la sexta fila volvemos a tener un comportamiento errático, que de nuevo, vuelve a dar mejores resultados por casualidad.

En la séptima fila, vemos como las soluciones convergen correctamente

hacia un óptimo local para un valor de $\eta = 0{,}01$. Mientras que para $\eta = 0{,}1$, tenemos un comportamiento errático. Esto se muestra con claridad en las siguientes dos gráfica de la traza de las soluciones (en la leyenda se muestran los valores de *eta*)

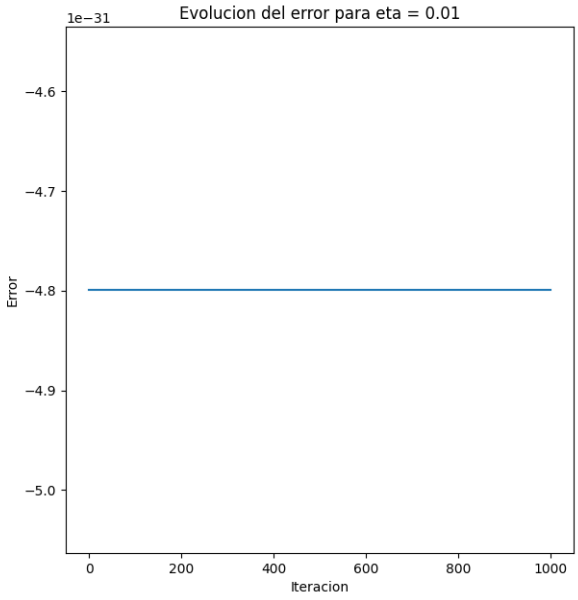


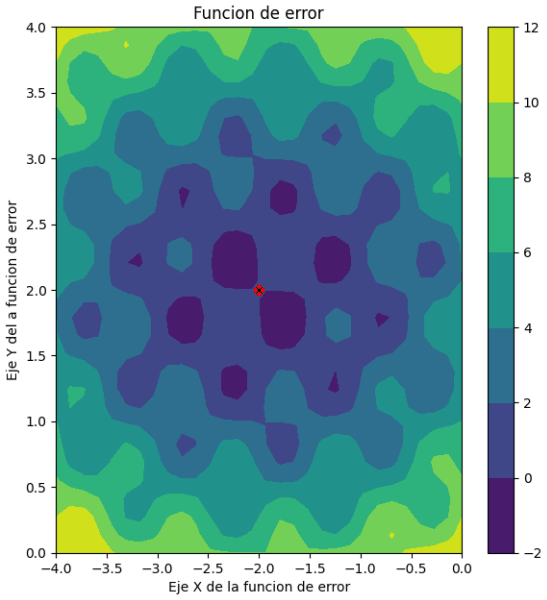


Además, en este caso, un valor apropiado de η consigue mejores resultados que con un valor que produce comportamiento errático.

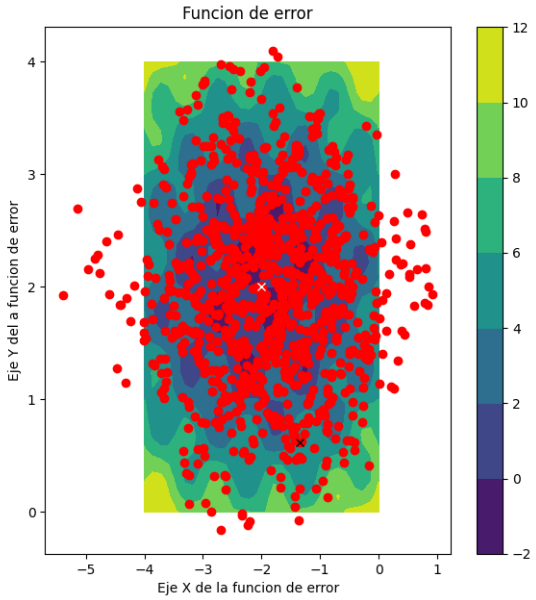
En la novena fila tenemos un muy buen comportamiento que da convergencia hacia un óptimo local (en la gráfica de la traza de la solución que muestra el programa esto se ve claramente. No añado esta gráfica pues ya se ha mostrado una muy parecida, para evitar ser repetitivo). Mientras que en la décima fila (η demasiado grande) tenemos el claro caso de comportamiento errático, que en este caso produce peor solución que un adecuado valor de η.

En la undécima fila tenemos un comportamiento muy curioso. La solución inicial y final coinciden. Esto se ve claramente en las siguientes dos gráficas:


Evolucion del error para eta = 0.01
1e−31
−4.6
−4.7
−4.8
−4.9
−5.0
Error
0
200
400
600
800
1000
Iteracion


Funcion de error
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Eje Y del a funcion de error
−4.0
−3.5
−3.0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
Eje X de la funcion de error
12
10
8
6
4
2
0
−2

Vemos como estamos en el filo de dos *agujeros* a los que nos interesaría caer. Estamos en un equilibrio muy inestable en lo que podría llamarse un punto de silla. En este caso en concreto, queremos que se rompa dicho equilibrio pues nos haría caer en uno de los dos óptimos locales. Notar también, que consumimos las mil iteraciones sin modificar la solución. Esto nos asegura que estamos en un punto de silla, pues el que no se mueva implica que el gradiente es cero. Gradiente nulo en un punto que no es óptimo indica con seguridad que es un punto de silla. Sin embargo, para un valor más grande de η nos salimos del equilibrio, pero produciendo un comportamiento errático (como se muestra en la siguiente gráfica) que nos lleva a una peor solución:



1.4. Apartado 4

La búsqueda de un mínimo global de una función arbitraria presenta varias dificultades, como hemos podido comprobar en estos casos de estudio.

Para comenzar, no conocemos a priori si la función estudiada tiene un mínimo global. Tenemos resultados como el que nos asegura que en una función convexa, todo mínimo local es un mínimo global (análogo con funciones cóncavas y máximos).

Los métodos analíticos para calcular puntos críticos son poco útiles en la práctica en espacios paramétricos de gran dimensionalidad. O con funciones para las que no existen soluciones analíticas, a pesar de que las funciones que definen el problema sean suficientemente regulares (continuas, diferenciables, C^∞, ...), con lo que estas deben ser aproximadas por métodos numéricos. Por tanto, para muchos casos de uso, estas técnicas no van a ser factibles.

Y en otros muchos casos, vamos a trabajar con funciones de error que directamente no van a ser analíticamente tratables, por lo que quedan directamente descartados estos métodos analíticos. Es aquí donde entran en juego otros métodos, como búsquedas heurísticas o el método del gradiente descendente (que en este caso, se apoya en una propiedad analítica como es la existencia del gradiente, o lo que es lo mismo, que sea al menos diferenciable una vez).

Este método en concreto, que es el que estudiamos en este apartado, tiene las problemáticas usuales de los métodos iterativos para la búsqueda de óptimos locales. Dependen de parámetros como la solución inicial escogida, condiciones de parada, tamaño de la población de búsqueda (en caso de las técnicas vistas hasta ahora, solo tenemos una solución en la población de soluciones sobre las que realizamos la búsqueda).

Pero en el caso concreto del gradiente descendente, dependemos de forma crítica de elegir un valor adecuado para η, pues tendrá un gran impacto a la hora de decidir si la búsqueda del óptimo local tiene un comportamiento bueno (descenso del error monótono a un buen ritmo) o un mal comportamiento (evolución del error no monótono, con grandes picos, a un ritmo elevado, o por otro lado, descenso monótono pero a un ritmo demasiado lento, lo que hace que se atasque en malos óptimos o que no consiga llegar al óptimo local al agotar las iteraciones máximas permitidas).

La elección de este valor de η además depende en gran medida de la función de error que queremos minimizar, y el valor de la solución de la que parte la búsqueda iterativa. Por tanto, y como ya se ha comentado, sería interesante disponer de técnicas para ajustar de forma dinámica el valor del *learning_rate*, pues nos interesa tener un valor alto al inicio para alejarnos rápidamente de zonas de alto error, y descender el valor de η para tener más precisión a la hora de acercarnos al óptimo local. Y todo esto, en cada caso concreto con el que nos encontremos (hasta ahora, dependiendo de la solución inicial y de la función concreta que estemos minimizando).

Además, en las dos funciones de error estudiadas, hemos tenido que calcular analíticamente la expresión del gradiente para poder operar. Aunque esto no debería ser un gran problema con otras funciones, puesto que existen paquetes de cálculo simbólico como sympy o aproximaciones numéricas del gradiente en un punto (como por ejemplo, diferencias finitas).

2. Ejercicio 2 - Regresión Lineal

2.1. Descripción del problema

Buscamos ajustar modelos de regresión a vectores de características extraídos de imágenes de dígitos manuscritos. Estas características son:

- Intensidad: valor medio del nivel de gris
- Simetría: respecto al eje vertical

Solo trabajaremos con los dígitos 1 y 5.

Cargaremos los datos, tanto de entrenamiento como de testing, de ficheros dados por los profesores de prácticas. La función que usamos para leer los datos, readData, también ha sido dada por los profesores.

Es importante tener en cuenta que la matriz de datos que cargamos del fichero ya tiene una primera columna de unos. Esta columna es necesaria porque representa el sumando del término independiente en las ecuaciones lineales con las que estamos trabajando de forma matricial. De no ser así, tendríamos que añadir esta columna de unos para poder operar cómodamente con la matriz. En el experimento con características no lineales, añadimos columnas a la matriz con la orden np.insert(X, nueva_columna)

Además, en este ejercicio, usaremos regresión lineal con la intención de clasificar los datos. Es decir, para un dato de entrada, devolvemos una solución real en un intervalo dado (regresión lineal). Después, tomamos como valor de clasificación la etiqueta cuyo valor numérico está más cercana a nuestra predicción de regresión. En el caso concreto de este ejercicio, las etiquetas son -1 y 1, luego es suficiente con quedarnos con el signo de la predicción de la regresión como valor de clasificación.

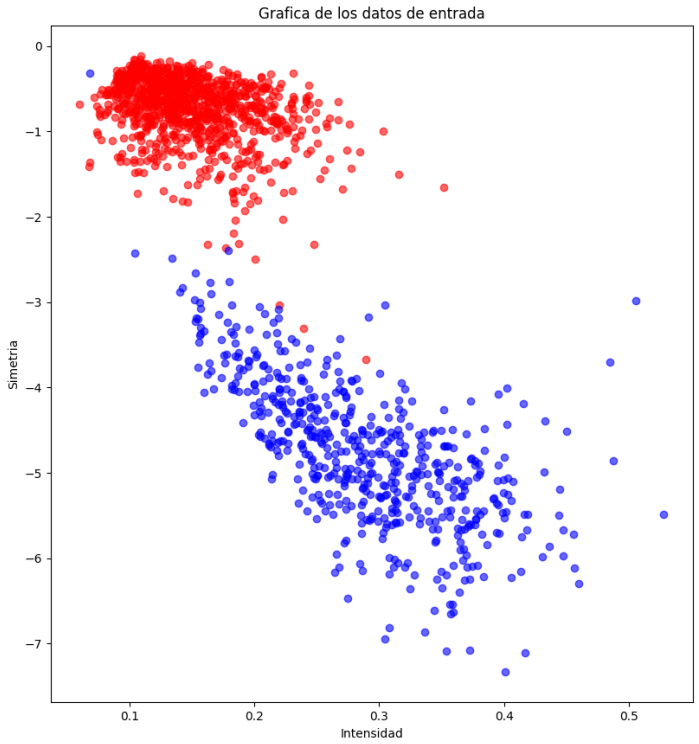
A diferencia del ejercicio anterior, en los que no teníamos datos, sino simplemente una función de error a minimizar, dispondremos tanto de datos de entrenamiento como datos de testing. Y con ellos, podremos calcular los valores de error en la muestra E_in como error fuera de la muestra E_out

2.2. Apartado 1

Se nos pide estimar un modelo lineal a partir de los datos proporcionados, usando tanto el algoritmo de la pseudo-inversa como el algoritmo de gradiente descendente estocástico. Las etiquetas de los datos serán:

- 1: para los dígitos cinco
- -1: para los dígitos uno

Una vez cargados los datos, estos pueden ser visualizados en un *scatter plot*:



Los puntos azules se corresponden con los unos, mientras que los rojos se corresponden con los cincos. Además, añadimos transparencias a los puntos para que sean más apreciables las zonas en las que hay mas concentración de puntos.

2.2.1. Pseudo-Inversa

Los resultados que mostramos por pantalla una vez ejecutado el algoritmo de la pseudo-inversa es:

```
Los pesos obtenidos son: [-1.11588016 -1.24859546 -0.49753165]
El error de clasficacion en la muestra Ein es: 2.2408193670794097
El error de clasificacion fuera de la muestra Eout es: 2.439522584578059
El error cuadratico medio en la muestra Ein es: 0.07918658628900388
El error cuadratico medio fuera de la muestra Eout es: 0.13095383720052575
El error porcentual en la muestra Ein es: 0.5124919923126201%
El error porcentual fuera de la muestra Eout es: 1.650943396226415%
```

Este algoritmo calcula una solución cerrada en un paso, pues está usando el despeje matemático del sistema de ecuaciones que se nos plantea. Así que así hemos calculado la mejor solución, la que menor error va a cometer, al menos en los datos de entrenamiento y con el error cuadrático medio.

El error porcentual obtenido es muy bueno, tanto en la muestra de entrenamiento (un 0,5125 %) como en la muestra de testing (un 1.6509 %). Como al usar la pseudo-inversa obtenemos la mejor solución, la información que esto nos aporta es que los datos con los que estamos trabajando están altamente correlados de forma lineal.

El error cuadrático es menos interpretable que el porcentual, pues depende de la escala de los datos con los que estemos trabajando, de las unidades con las que estemos trabajando.... Además, al ser una media de suma de errores al cuadrado es más complicado que lo interpretemos sin tener otras referencias. Este error cuadrático medio nos servirá para hacer comparaciones con los errores cuadráticos de los métodos iterativos, pues como ya se ha mencionado, la pseudo-inversa obtiene el error mínimo para este problema, al menos para el error cuadrático medio. Y también es interesante tener un control sobre el error cuadrático medio pues es la función de error que los métodos iterativos van a tratar de minimizar (de nuevo, este algoritmo no

es un método iterativo, da una solución cerrada en un paso).

El error de clasificación es el error que aparece en la teoría, en el contexto del algoritmo *PLA*. El profesor ya nos comentó que estos métodos serán tratados en la siguiente práctica, pero hemos pensado que no vendría mal tener una referencia sobre esa medida de error.

Un inconveniente de este algoritmo es que para matrices mucho más grande que esta, es más lento que métodos iterativos como los que vamos a explorar a continuación. Sin embargo, no tenemos que escoger un valor adecuado para el learning_rate. Por otro lado, solo es aplicable a regresión lineal, mientras que las distintas técnicas que usan el gradiente descendente son aplicables a distintos problemas.

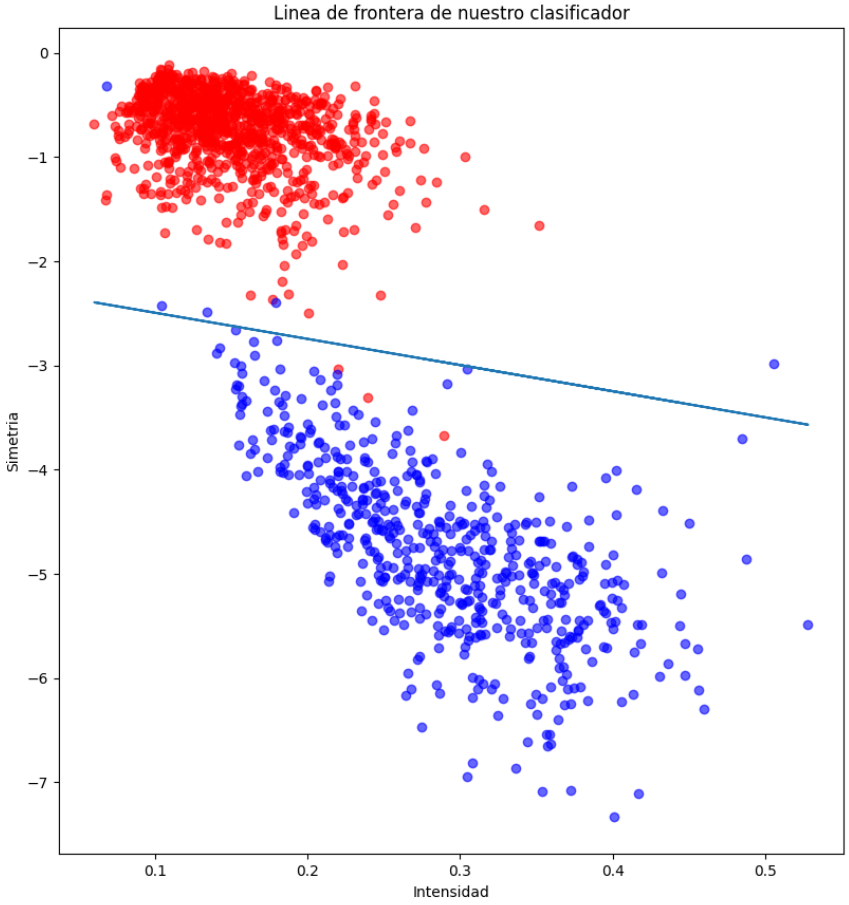
Ahora mostramos la recta que separa los datos en los dos grupos en los que clasificamos. Es lo que podríamos llamar la recta frontera de la clasificación. Un dato lo clasificamos de una u otra forma según el signo de la predicción, por tanto, la frontera de clasificación será la recta que prediga los datos a un valor de cero, y con ello hacemos el siguiente desarrollo:

$$y = w_0 + w_1 x_1 + w_2 x_2$$

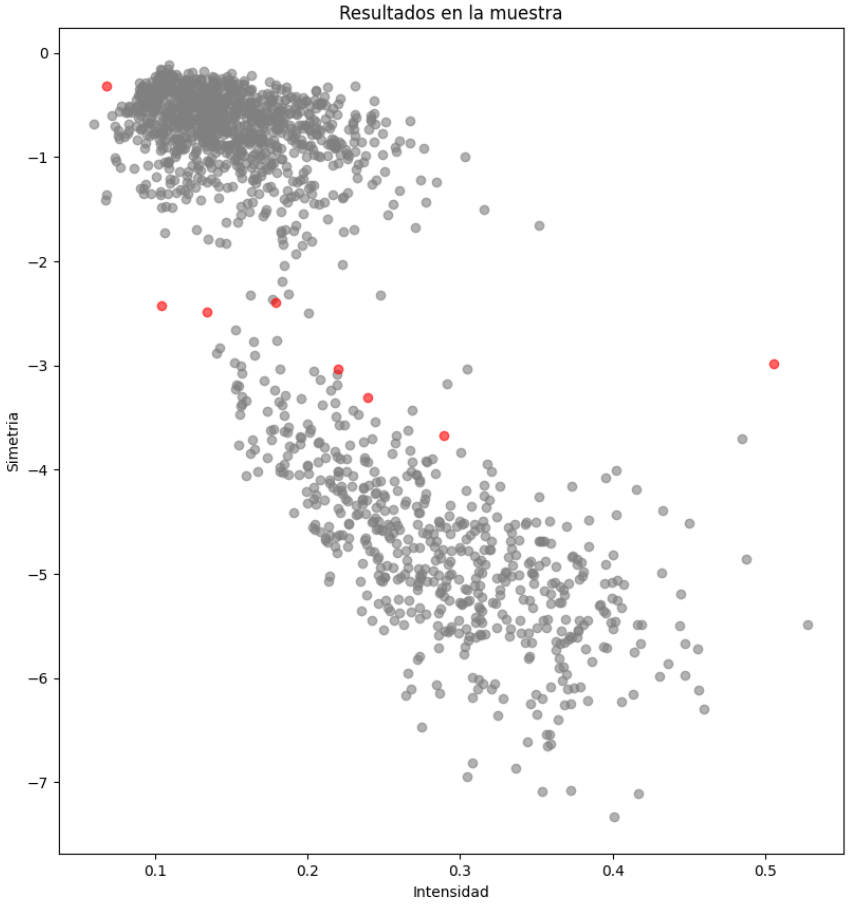
La frontera será por tanto los valores para los que tengamos la predicción $y = 0$ y por tanto:

$$0 = w_0 + w_1 x_1 + w_2 x_2 \Longrightarrow x_2 = \frac{1}{w_2}(-w_0 - w_1 x_1)$$

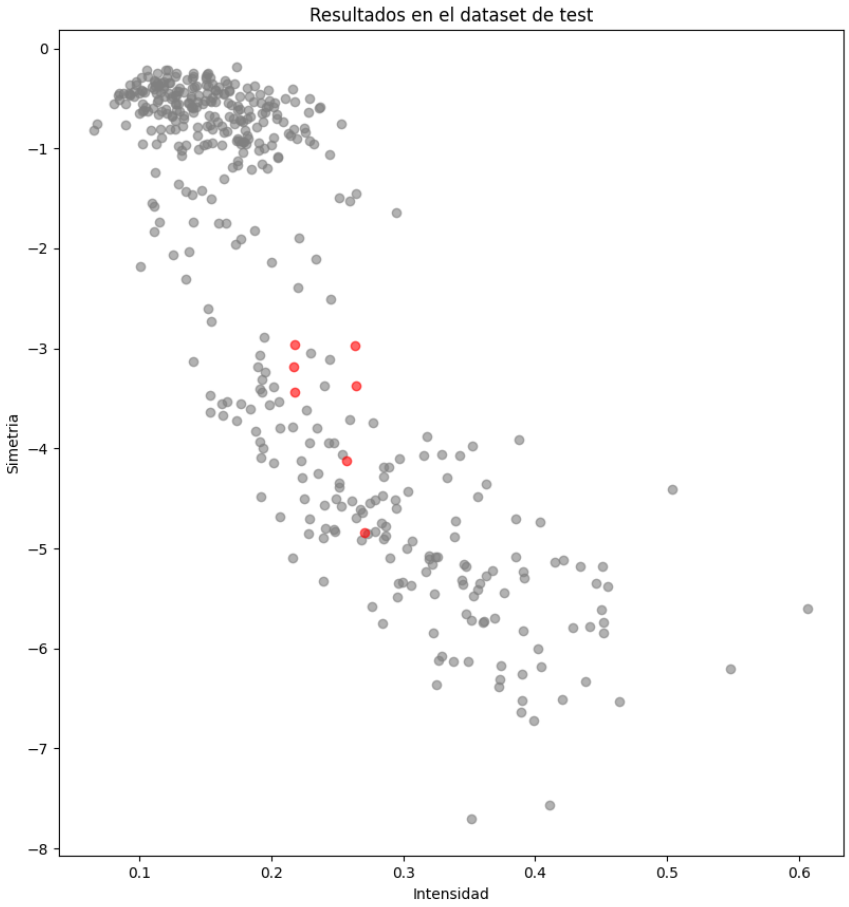
y con esta fórmula ya podemos representar la recta frontera:



Esta recta frontera está representada con los datos de entrenamiento en el fondo. Además de esta gráfica, voy a mostrar una gráfica para ver dónde fallamos. En esta gráfica, los datos bien clasificados se mostrarán en gris, mientras que los datos mal clasificados se mostrarán en rojo. Dicha gráfica sobre los datos de entrenamiento es:



Y la misma gráfica sobre los datos de test es:



Por tanto, podemos concluir que la muestra de datos está bien correlada linealmente y por tanto esta técnica tiene buenos resultados, a vista de los resultados reflejados en E_in y E_out. Además como no tenemos una gran cantidad de datos, el algoritmo devuelve la solución en un tiempo razonable.

2.2.2. Stochastic Gradient Descent

Este algoritmo depende de generar valores aleatorios. Así que, para trabajar con resultados reproducibles, fijamos la semilla aleatoria con np.random.seed(123456789).

Stochastic Gradient Descent, en su formulación clásica, trabaja con batch size $= 1$, pero como nos indica el profesor de prácticas, es mejor explorar distintos valores de *batch size*, así que realmente estamos trabajando con *MiniBatch Stochastic Gradient Descent*.

Probaremos con un $batch\_size \in \{1, 32, data\_size\}$, donde $data\_size$ es el tamaño de la matriz de datos, con lo que tenemos, según el valor del $batch\_size$:

- 1: Stochastic Gradient Descent clásico
- 32: Minibatch con un valor del $batch\_size$ convencional
- 64: Minibatch con otro valor del $batch\_size$ convencional
- $data\_size$: Batch Gradient Descent, algoritmo que hemos usado en el Ejercicio 1

En estas pruebas solo nos quedaremos con los errores porcentuales, al ser los más fácilmente interpretables, y una vez escogido el tamaño del $batch\_size$, mostraremos los resultados completos. Además, siguiendo la indicación de nuestro profesor de prácticas, con 200 iteraciones sobre los minibatches generados, debería ser suficiente. Por otro lado, el valor del *learning rate* lo dejaremos a 0,01 tras haber hecho una exploración de posibles valores. Esta exploración no la incluimos también por no hacer demasiado extensa la memoria de esta práctica.

Tamaño del minibatch	Error Porcentual en la muestra	Error porcentual fuera de la muestra
1	0.4484 %	1.8867 %
32	0.4484 %	1.6509 %
64	0.4484 %	1.6509 %
$data\_size$	0.4484 %	1.6509

Todos los algoritmos corren en poco tiempo, salvo cuando usamos $data\_size$, que tarda algo más. Todos los errores, cuando miramos cuatro cifras decimales, son los mismos, salvo cuando usamos $batch\_size = 1$, que fuera de la muestra obtiene peores resultados. Por tanto, decidimos quedarnos con $batch\_size = 32$, por ser un valor bastante convencional, que produce buenos resultados y que tarda poco.

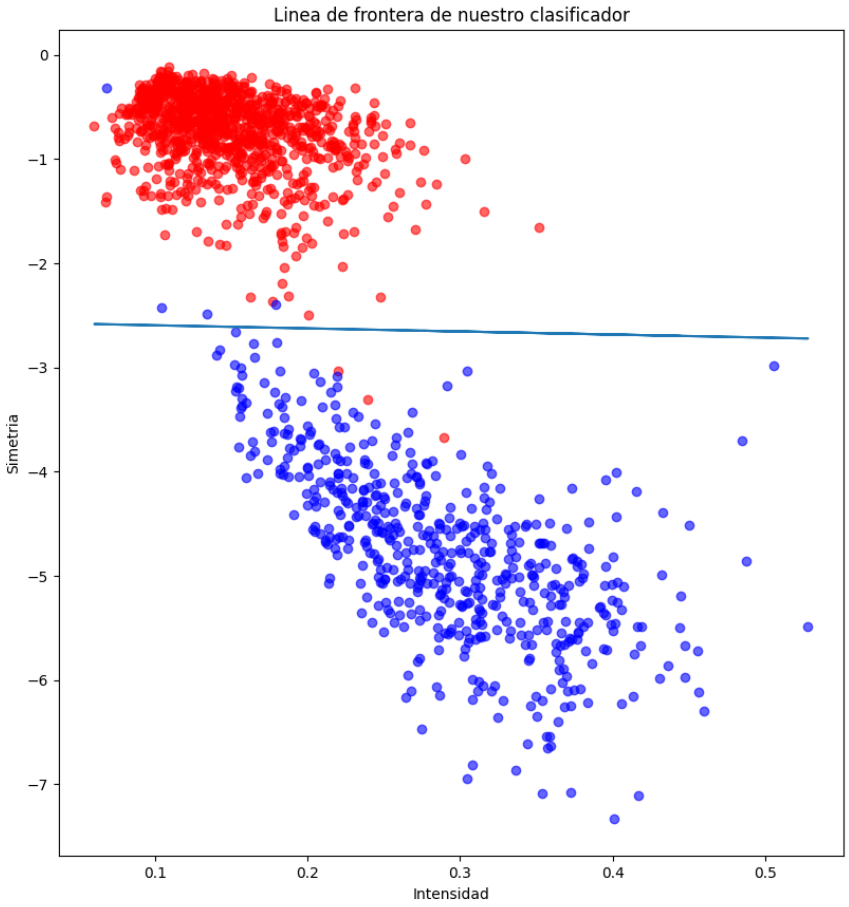
Así que corriendo el algoritmo con un valor de $batch\_size = 32$ obtenemos los siguientes resultados que mostramos por pantalla:

```
Los pesos obtenidos son: [-1.0312233  -0.11892069 -0.40199688]
El error de clasficacion en la muestra Ein es: 1.959996706581955
El error de clasificacion fuera de la muestra Eout es: 2.5841736601550416
El error cuadratico medio en la muestra Ein es: 0.10448833536429288
El error cuadratico medio fuera de la muestra Eout es: 0.1533263011697918
El error porcentual en la muestra Ein es: 0.4484304932735426%
El error porcentual fuera de la muestra Eout es: 1.650943396226415%
```

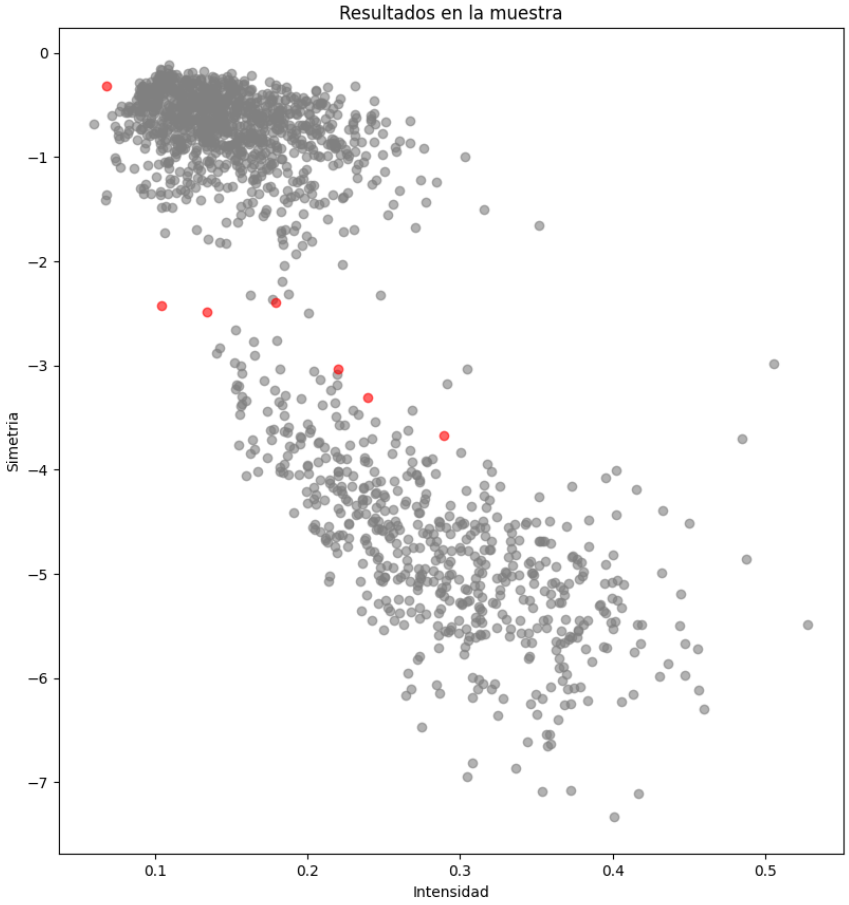
Tanto fuera como dentro de la muestra, obtenemos errores porcentuales por debajo de los errores porcentuales de los errores porcentuales del algoritmo de la pseudo-inversa. Esto puede ocurrir al estar minimizando el error cuadrático medio y no un error porcentual. Sin embargo, el error cuadrático medio es más alto con este algoritmo que con la pseudo-inversa, lo que era de esperar.

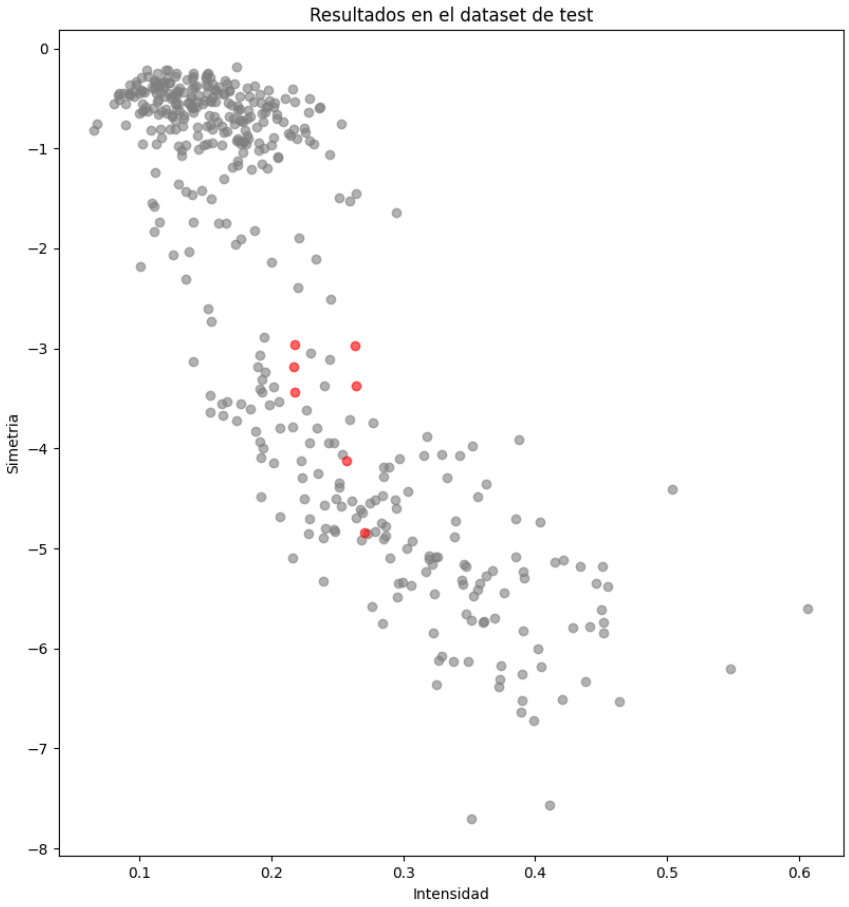
El error porcentual solo tiene en cuenta si clasificamos bien un punto o no. Mientras que el error cuadrático medio mide también cómo de lejos está la predicción a través de regresión del valor de la etiqueta que queremos alcanzar. Por ejemplo, podemos predecir para un dato de entrada con verdadera etiqueta 1 la salida 0.57. Respecto al error porcentual, no estamos cometiendo error, pues la clasificación que se deriva es correcta. Pero para el error cuadrático medio se incluye en la sumatoria de la media un sumando $(1 - 0{,}57)^2$.

Mostramos la recta frontera obtenida con este algoritmo:



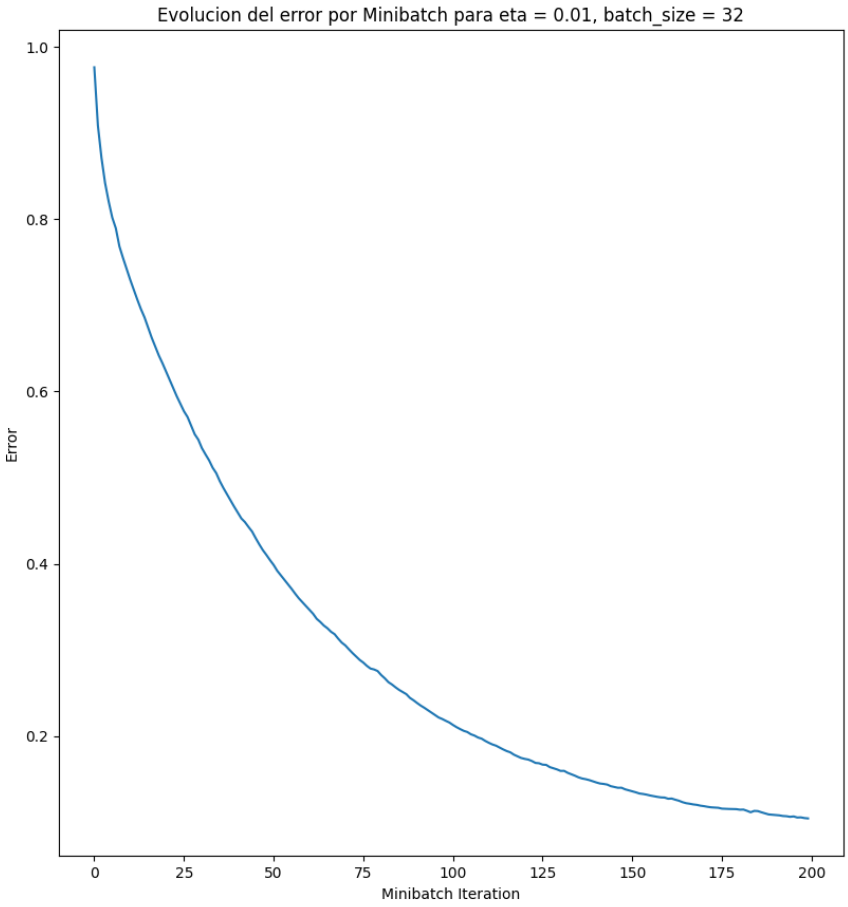
También mostramos las gráficas de puntos errados, tanto en la muestra de entrenamiento como en la muestra de test:


Resultados en la muestra
Simetria
Intensidad
0
−1
−2
−3
−4
−5
−6
−7
0.1
0.2
0.3
0.4
0.5



En esta gráfica, podemos ver que por ejemplo, el punto que está el segundo más a la derecha, está bien clasificado, mientras que con la pseudo-inversa se clasifica mal. Esto puede ocurrir porque para minimizar el error cuadrático medio, seguramente convenga que ese punto se clasifique mal pero que en contrapartida el error cuadrático medio global sea menor.

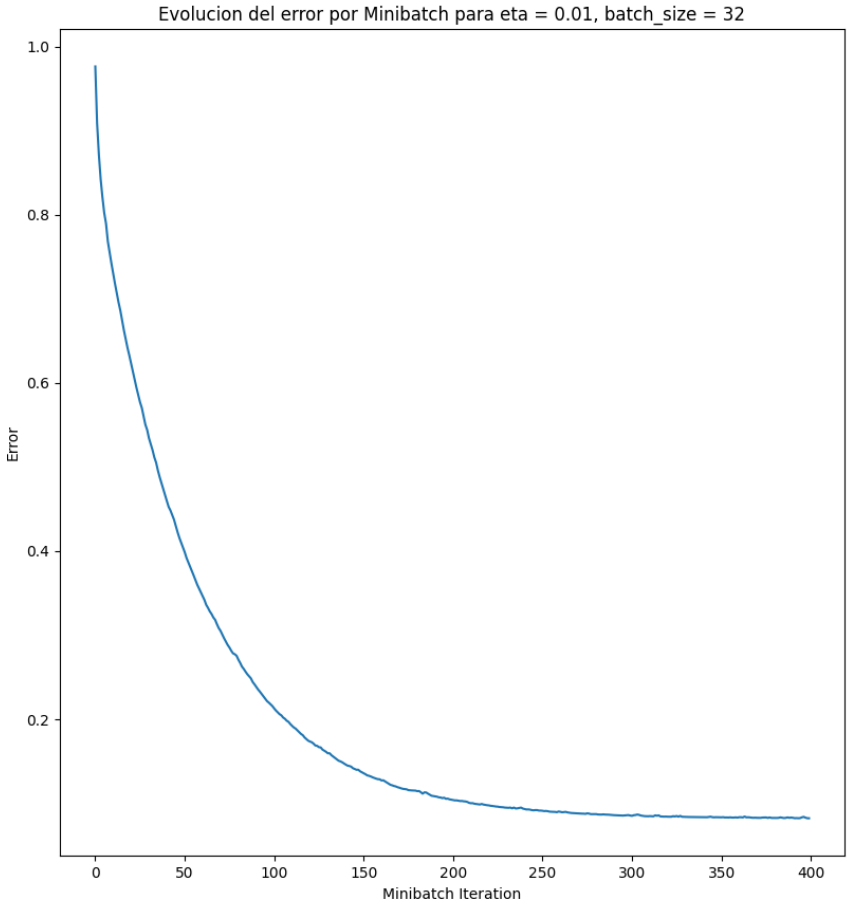
Al estar usando un proceso iterativo, podemos mostrar cómo avanza el error en cada iteración sobre cada uno de los minibatches:



Notar que hay iteraciones en las que el error asciende en vez de descender de forma monótona, como buscábamos en Batch Gradient Descent. Sin embargo, ahora este comportamiento no es indeseable como lo era antes. Para empezar, el ascenso del error es pequeño, no como en Batch Gradient Descent en los que teníamos picos demasiado bruscos. Además, la tendencia general es claramente descendente. Y en ocasiones puede ser interesante poder tener iteraciones que aumenten el error. Ya en el ejercicio 1 nos habría venido bien, tanto para salir de esos pequeños baches como para caernos de un punto de silla hacia un mínimo local.

Estos pequeños aumentos del error se deben a que, como no estamos usando toda la información de la muestra de datos, ciertas iteraciones pueden ir en una dirección confundida que indica el cálculo del gradiente con ese *subset* de los datos.

También puede parecer que las doscientas iteraciones se quedan cortas a la hora de seguir disminuyendo el error. Así que pruebo a poner un máximo de 400 iteraciones para ver si se puede seguir disminuyendo el valor del error, obteniendo la gráfica siguiente:



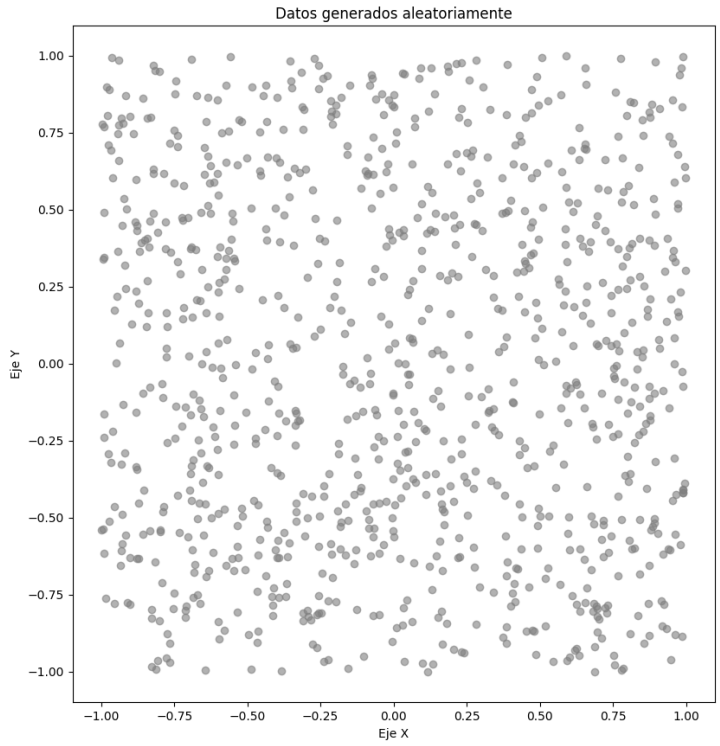
Podemos ver que a partir de las doscientas iteraciones apenas se reduce el error. Alcanzamos una cota para el error que podemos minimizar. Esto puede ser bien porque los datos con los que trabajamos no permiten reducir más el error (hemos alcanzado un mínimo local), o porque a partir de las doscientas iteraciones convendría emplear un *learning_rate* más bajo para poder mejorar el error con más precisión.

2.3. Apartado 2

Usamos la función de generación de datos simula_unif(N, 2, size) que nos devuelve N coordenadas bidimesionales en el cuadrado $[-size, size]^2$

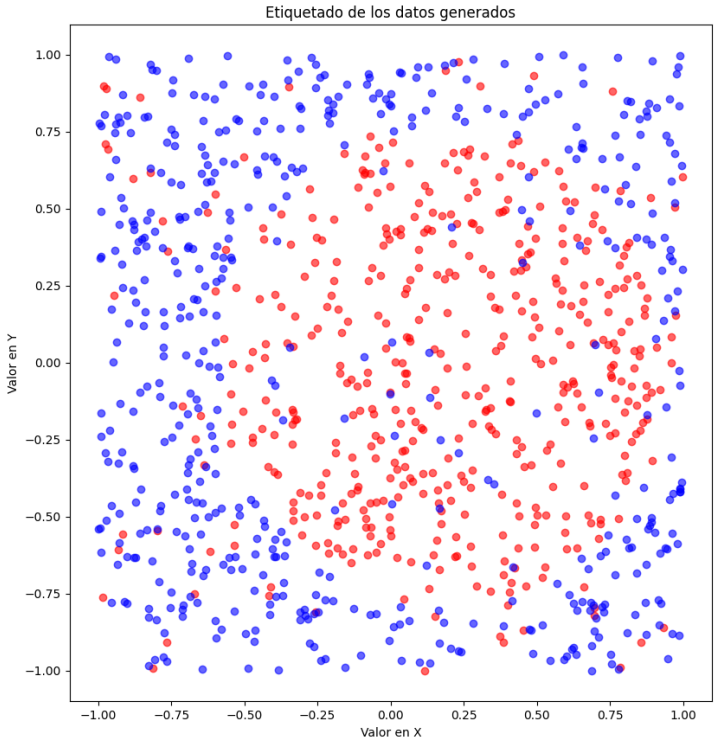
2.3.1. Subapartado a

Usando la anterior función generamos los 1000 puntos en el cuadrado $[-1, 1]^2$ usando simula_unif(1000, 2, 1), y generamos el siguiente gráfico:



2.3.2. Subapartado b

Consideramos la función $f(x_1, x_2) = sign((x_1 - 0{,}2)^2 + x_2^2 - 0{,}6)$. Usamos dicha función para etiquetar los datos. Introducimos ruido sobre el 10 % de las etiquetas, es decir, cambiamos el signo del 10 % de las etiquetas. Para ello, generamos los índices $\{0, 1, \ldots, 999\}$, los mezclamos y cambiamos el signo de los datos datos correspondientes a los primeros 100 indices. Por tener más generalidad, trabajamos con el 10 % de los indices generados. Así obtenemos la siguiente gráfica:



2.3.3. Subapartado c

Ajustamos un modelo lineal usando el vector de características $(1, x_1, x_2)$ usando Gradiente Descendiente Estocástico.

Necesitamos añadir una columna de unos a la matriz de datos generadas. Para ello usamos las instrucciones:

```
number_of_rows = int(np.shape(X)[0])
new_column = np.ones(number_of_rows)
X = np.insert(X, 0, new_column, axis = 1)
```

Para lanzar *SGD* establecemos los siguientes parámetros:

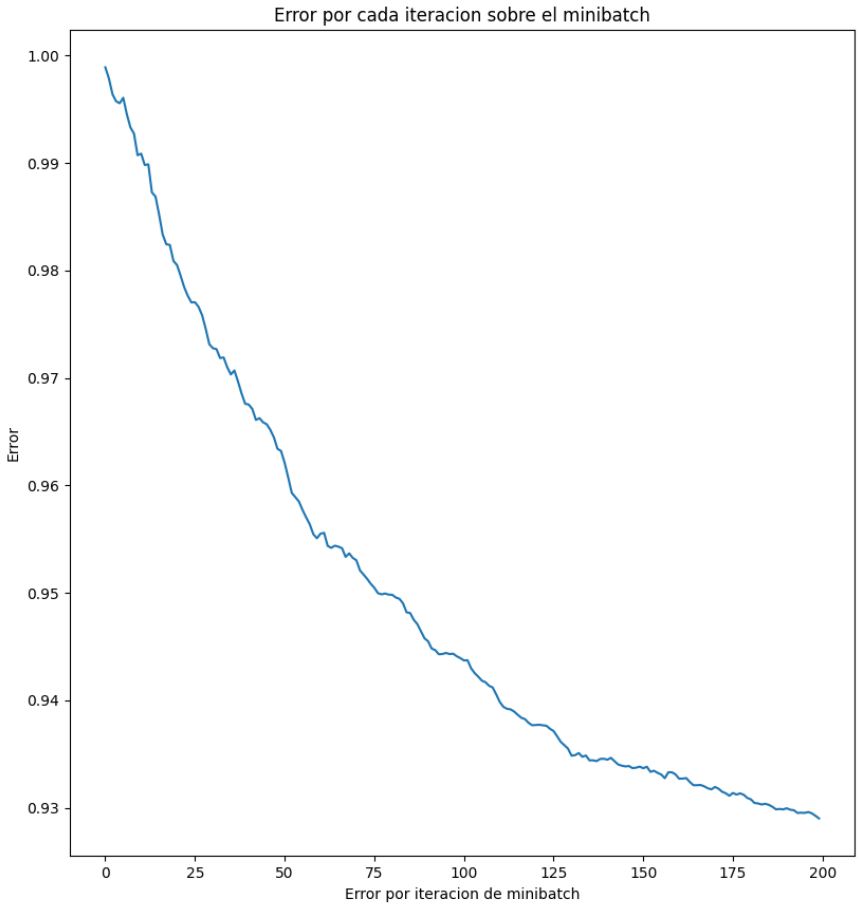
- Solución inicial: vector de ceros con la dimensionalidad adecuada
- *batch_size*: 32, por lo bien que nos ha funcionado en el ejercicio anterior
- Número máximo de iteraciones: 200, el profesor nos comentó que deberían ser suficientes. Y en el apartado anterior vemos que con más iteraciones no mejoramos demasiado el resultado
- η: 0.01 pues hemos explorado otros valores para este problema y este nos ha funcionado bien

El algoritmo nos devuelve el siguiente resultado:

```
Los pesos obtenidos son: [ 0.0495547  -0.3450253   0.03473961]
El error de clasficacion en la muestra Ein es: 61.57961277110754
El error cuadratico medio en la muestra Ein es: 0.9290128091406921
El error porcentual en la muestra Ein es: 39.800000000000004%
```

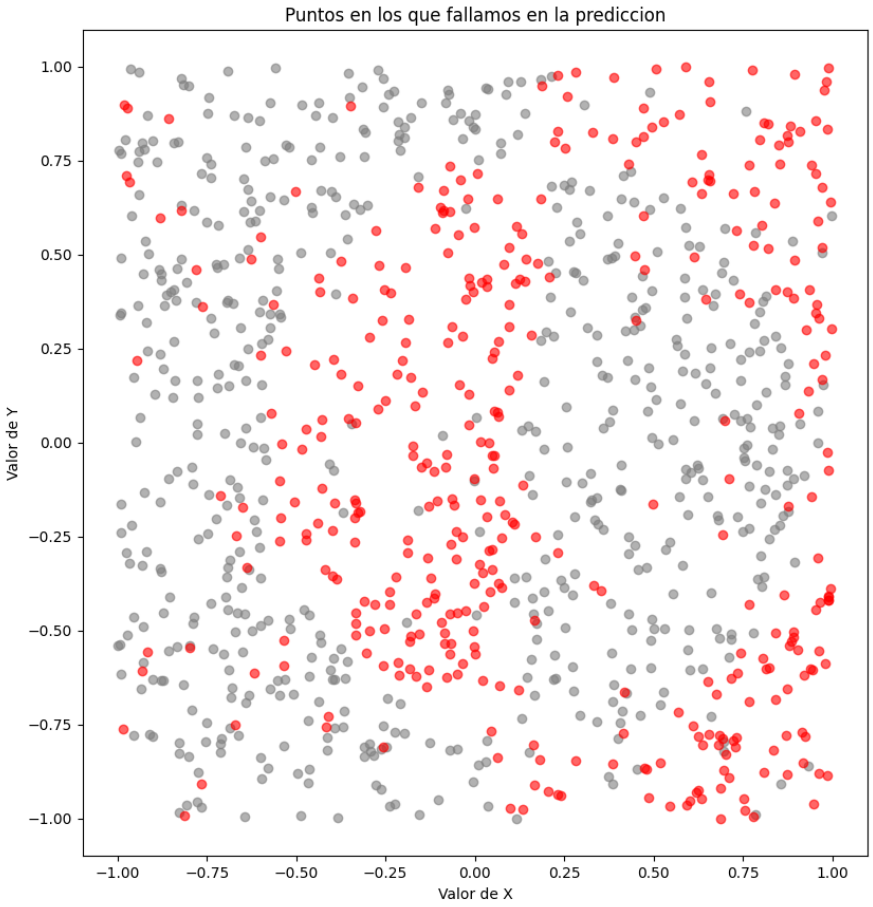
Fallamos en prácticamente el 40 % de los datos. Por tanto, no estamos consiguiendo buenos resultados. En este caso, podemos afirmar que los datos no son buenamente tratables con regresión lineal, al menos con el vector de características que estamos trabajando. Esto ya lo podíamos saber fácilmente viendo la función que hemos usado para etiquetar los datos.

Mostramos cómo avanza el error sobre las iteraciones en el minibatch:



Como en el apartado anterior, vemos que con 200 iteraciones la tendencia de la gráfica es a aplanarse, por lo tanto, con unos resultados tan malos no merece la pena estudiar si con más iteraciones mejoraríamos el resultado.

Y los puntos en los que falla la clasificación:



2.3.4. Subapartado d

Repetimos el experimento 1000 veces. En cada una de estas repeticiones, generamos una nueva muestra aleatoria de 1000 puntos para el training, y otra muestra aleatoria de 1000 puntos para testing, así que ahora podremos computar el valor tanto de E_in como de E_out. Los resultados obtenidos son:

```
Error medio porcentual DENTRO de la muestra: 40.41820000000005%
Error medio porcentual FUERA de la muestra: 40.74019999999995%
Error medio cuadratico medio DENTRO de la muestra: 0.9310675034757584
Error medio cuadratico medio FUERA de la muestra: 0.9363078116801644
```

Así que repitiendo mil veces el experimento, vemos que la clasificación usando regresión lineal es muy mala. Además, se clasifica prácticamente igual de mal dentro de la muestra de entrenamiento como en la muestra de testing.

2.3.5. Subapartado e

Por tanto, como ya hemos comentado, no es adecuado el uso de regresión lineal para clasificar este conjunto de datos. Un 40 % de error no es admisible a la hora de clasificar los datos. Esto era previsible desde el momento que hemos usado una función de etiquetado $f(x, y)$ no lineal, pero un vector de características lineal para realizar las predicciones.

Todo esto es motivación suficiente para emplear el vector de características no lineales del siguiente ejercicio.

2.4. Repetir el apartado, pero usando otro vector de características

Vamos a realizar el mismo experimento que en el apartado anterior, pero usando un vector de características no lineales. Este vector viene dado por $\phi(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$

Para ello, tenemos que añadir nuevas columnas en nuestra matriz de datos, al igual que hicimos para añadir la columna de unos en la primera posición. Usando numpy calculamos las nuevas columnas, para después añadirlas a la matriz. Todo el código de modificación de la matriz queda tal que:

```
# Añadimos la columna de unos al training
number_of_rows = int(np.shape(X)[0])
new_column = np.ones(number_of_rows)
X = np.insert(X, 0, new_column, axis = 1)
```

```
# Añadimos la tercera columna x1 * x2
third_col = X[:, 1] * X[:, 2]
X = np.insert(X, 3, third_col, axis = 1)

# Añadimos la cuarta columna x1**2
fourth_col = X[:, 1] * X[:, 1]
X = np.insert(X, 4, fourth_col, axis = 1)

# Añadimos la quinta columna x2**2
last_col = X[:, 2] * X[:, 2]
X = np.insert(X, 5, last_col, axis = 1)
```

Este es el único cambio que tenemos que hacer para correr el algoritmo *SGD* adaptado a estas características no lineales, pues todas las operaciones son sobre vectores y matrices usando numpy.

2.4.1. Lanzar 1000 veces el experimento

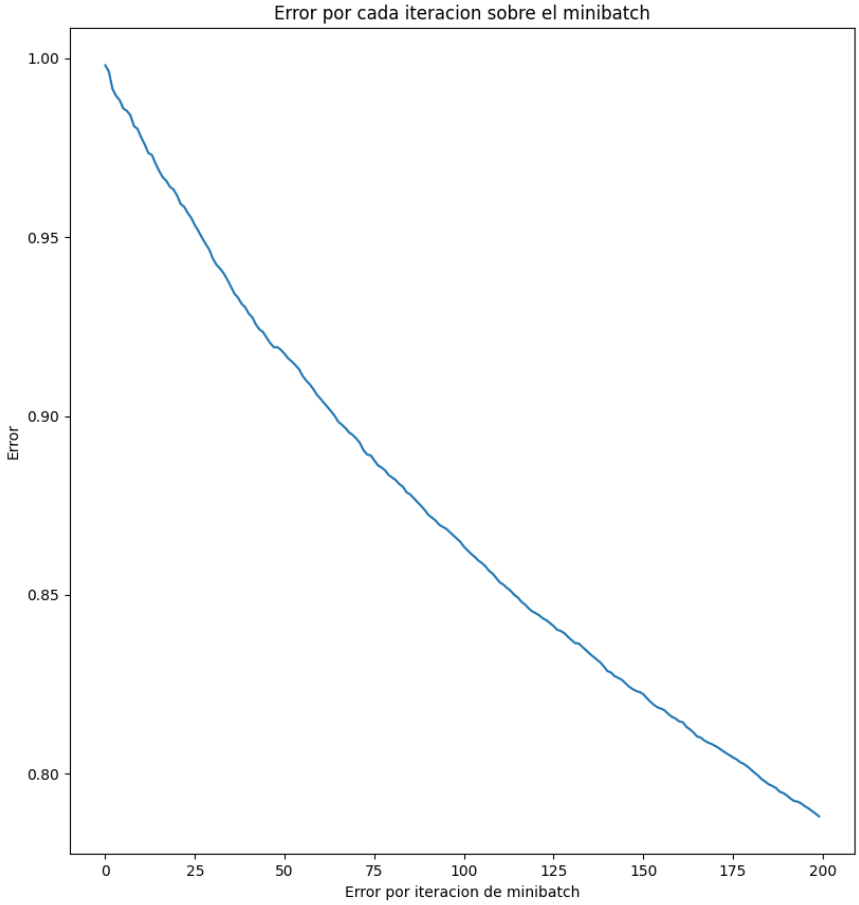
Los resultados tras lanzar 1000 veces el experimento y promediando los valores son:

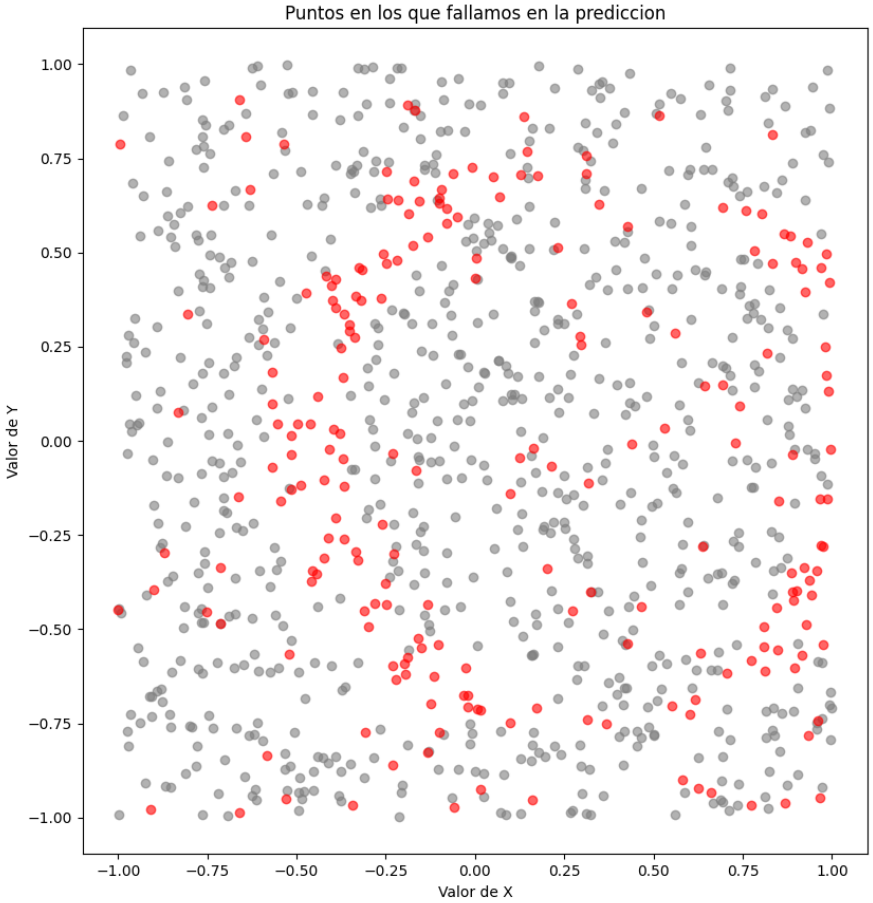
```
Error medio porcentual DENTRO de la muestra: 22.85820000000001%
Error medio porcentual FUERA de la muestra: 23.287299999999977%
Error medio cuadratico medio DENTRO de la muestra: 0.7737067204159451
Error medio cuadratico medio FUERA de la muestra: 0.7786926343602749
```

2.4.2. Mostramos los resultados para un experimento concreto

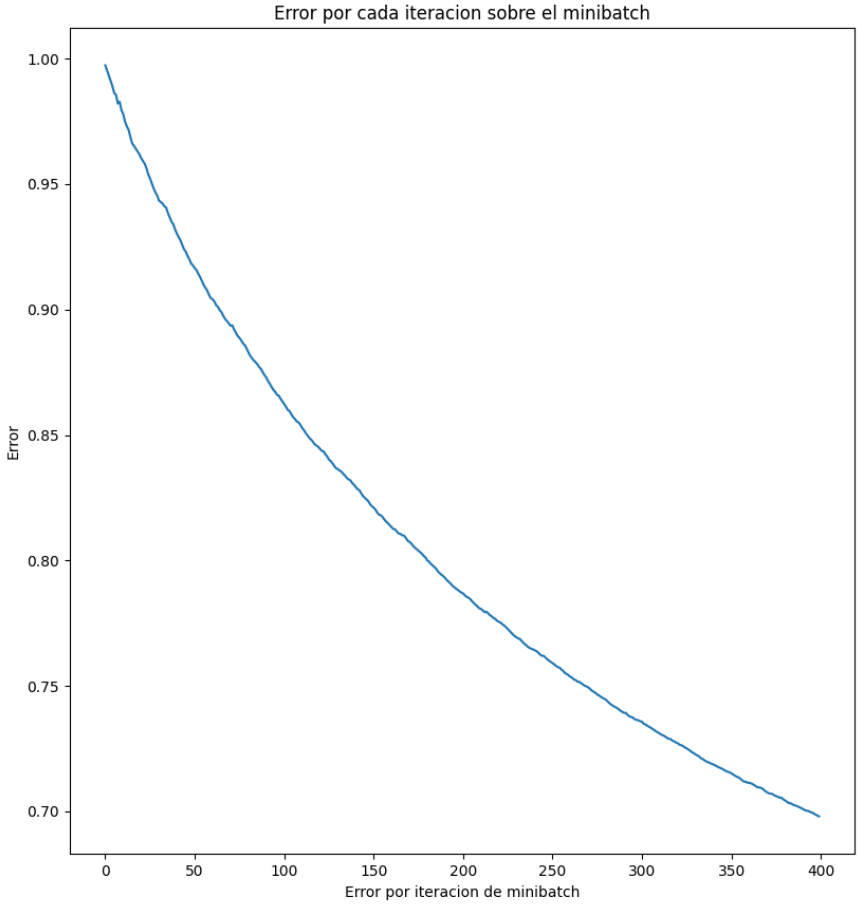
En este caso, no vamos a mostrar los valores de los distintos errores, pues son más representativos aquellos calculados a partir de mil repeticiones. En este caso, nos interesa ver las gráficas que nos van a dar mucha información sobre el comportamiento del algoritmo una vez que hemos añadido características no lineales.

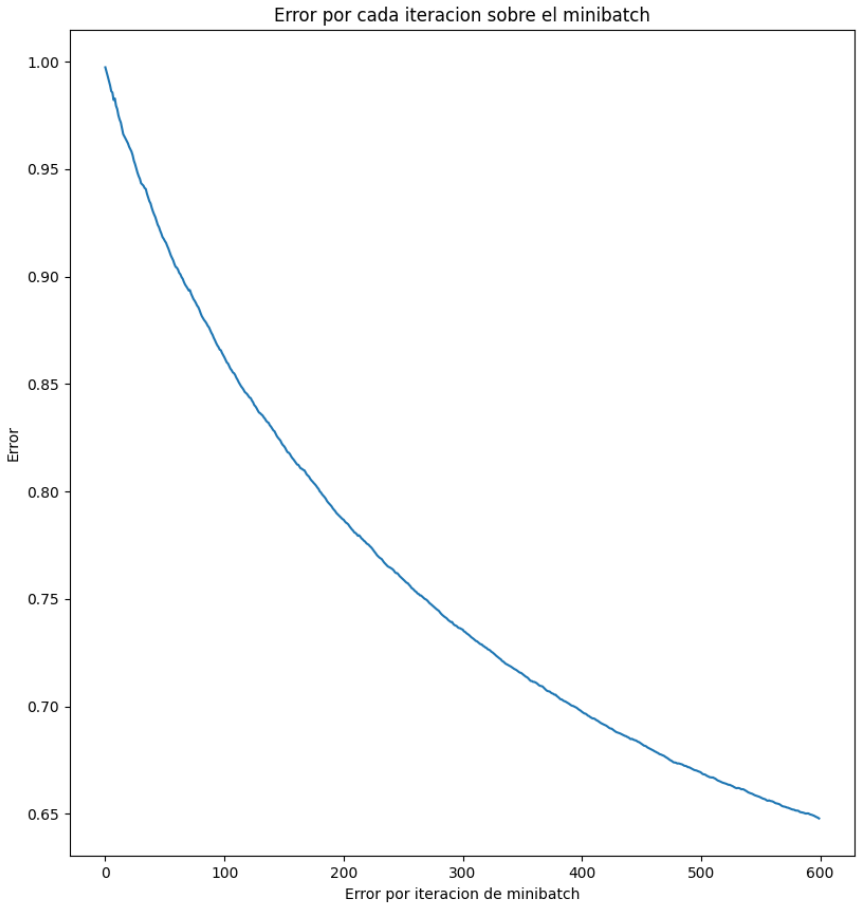
Al usar un valor de max_iterations de 200 obtenemos la siguiente gráfica de evolución de error y puntos mal clasificados:

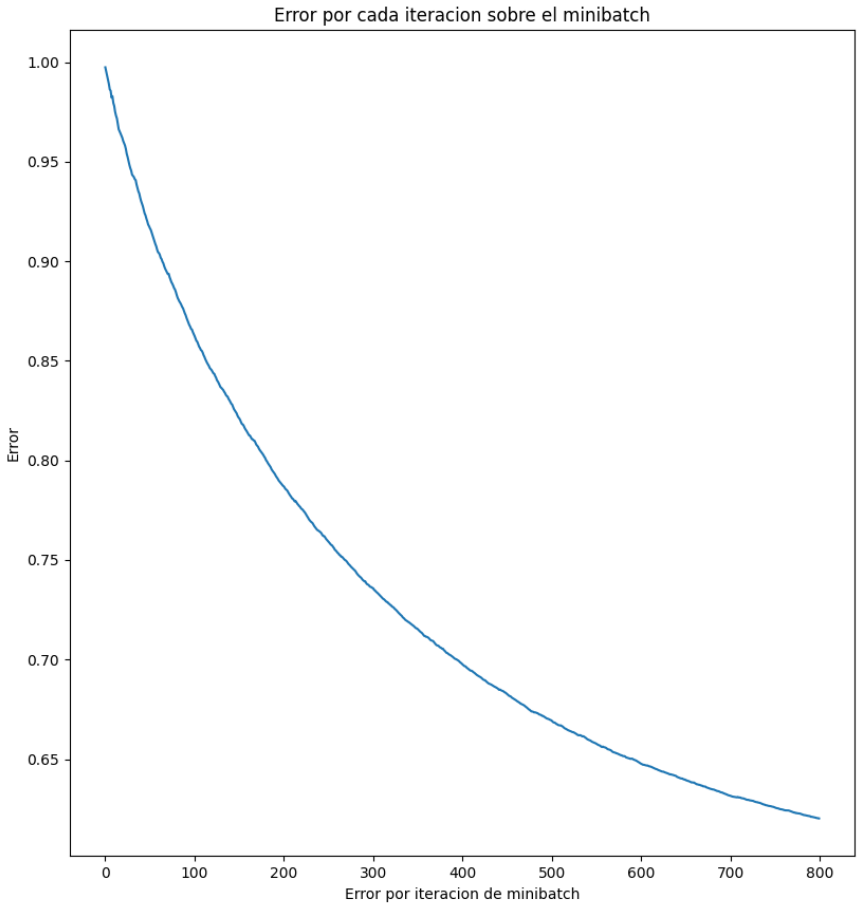

Error por cada iteracion sobre el minibatch
1.00
0.95
0.90
0.85
0.80
Error
0
25
50
75
100
125
150
175
200
Error por iteracion de minibatch



Según esta gráfica de error, parece que podemos seguir mejorando el error conforme avancen más iteraciones, así que pruebo con 400, 600 y 800 iteraciones, obteniendo las siguientes gráficas de errores:


Error por cada iteracion sobre el minibatch
1.00
0.95
0.90
0.85
0.80
0.75
0.70
Error
0
50
100
150
200
250
300
350
400
Error por iteracion de minibatch


Error por cada iteracion sobre el minibatch
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Error
0
100
200
300
400
500
600
Error por iteracion de minibatch



Efectivamente, con más iteraciones hemos podido mejorar aún más el error. A partir de 800 iteraciones, el error se estabiliza. Así que gracias a este análisis, repito las mil iteraciones del experimento pero con un valor de max_iterations de 1000

2.4.3. Lanzar 1000 veces el experimento, cambiando max_iterations

Los resultados obtenidos en el experimento, tras haber cambiado este parámetro en el código, son:

```
Error medio porcentual DENTRO de la muestra: 13.98220000000001%
Error medio porcentual FUERA de la muestra: 14.315200000000006%
Error medio cuadratico medio DENTRO de la muestra: 0.6134839356488047
Error medio cuadratico medio FUERA de la muestra: 0.6203435366656959
```

2.4.4. Conclusiones

Claramente añadiendo las características no lineales hemos mejorado enormemente el valor del error, tanto en las muestras de entrenamiento como en las muestras de testing. Hemos pasado de un 40 % de error aproximadamente a un 14 % de error porcentual. Pero para ello, ha sido necesario aumentar el número de iteraciones máximas, es decir, que el proceso de aprendizaje es más lento.

Este cambio hace que las mil iteraciones tarden mucho en ejecutarse, pero a cambio pasamos de unos errores porcentuales, tanto dentro como fuera de la muestra, de entre 22 % y 23 %, a unos errores porcentuales de entre el 13 % y 14 %.

Por tanto, considero que es más adecuado el modelo con características no lineales, pues reduce en gran medida los errores cometidos. Aunque requiera mucho más tiempo de ejecución, este tiempo no es demasiado grande como para que el algoritmo no sea aplicable, mientras que un error del 40 % no lo podemos considerar admisible a la hora de ser aplicado a un problema de clasificación.

2.5. Consideraciones sobre la implementación de Minibatch Gradient Descent

Para generar los minibatches de forma eficiente y cómoda, lo que hemos hecho ha sido generar todos los índices de los puntos con los que trabajamos: $\{0, 1, 2, \ldots, N\}$ y los hemos mezclado aleatoriamente. Una vez hecho, devolvemos el vector de grupos de tamaño *batch_size*. Con esto conseguimos dos cosas:

- No mezclamos la matriz de datos, que al ser muy grande, podría llegar a ser costoso dependiendo de las implementaciones de los métodos que usemos para ello
- Es muy sencillo realizar exactamente la misma mezcla y agrupación a las etiquetas, pues solo hay que aplicar los mismos índices y grupos generados para la matriz de datos

Esta funcionalidad se implementa en la función get_minibatches(), que como ya hemos comentado devuelve un vector de vectores (vector con las agrupaciones de índices de cada minibatch). Mostramos el código a continuación:

```
def get_minibatches(data, batch_size: int):
    """
    Dados unos datos de entrada, mezcla los datos y los devuelve en subcon
    de batch_size elementos. Realmente devolvemos un array de conjuntos de
    que representan este mezclado y empaquetado, pues asi es mas facil de
    (no alteramos los datos de entrada y no tenemos que considerar como qu
    ordenadas las etiquetas asociadas a los datos) y mas eficiente (trabaja
    con indices, no con datos multidimensionales)

    Paramters:
    ==========
    data: matriz de datos de entrada que queremos mezclar y agrupar
    batch_size: tamaño de los paquetes en los que agrupamos los datos

    Returns:
    ========
    indixes: array de conjuntos de indices (array tambien) que representa
```

```
            descrita anteriormente
    """

    # Los indices de todos los datos de entrada
    # Me quedo con las filas porque indican el numero de datos con los que
    # Las columnas indican el numero de caracteristicas de cada dato
    all_indixes = np.arange(start = 0, stop = np.shape(data)[0])

    # Mezclo estos indices antes de dividirlos en minibatches
    np.random.shuffle(all_indixes)

    # Array de conjuntos de indices (array de arrays)
    grouped_indixes = []

    # Agrupamos los indices que ya han sido mezclados en los minibatches
    last_group = []
    for value in all_indixes:

        # El ultimo minibatch no esta completo, podemos añadir un nuevo pu
        if len(last_group) < batch_size:
            last_group.append(value)

        # El minibatch esta completo, asi que hay que hay que añadirlo al
        # de minibatches y reiniciar el grupo
        if len(last_group) == batch_size:
            grouped_indixes.append(last_group)
            last_group = []

    return np.array(grouped_indixes)
```

Una vez que tenemos generados los grupos de índices, es muy fácil tomar los datos y etiquetas de un minibatch, gracias a que estamos trabajando con numpy, el cual nos permite usar el operador corchete con un vector de índices. Por ejemplo, el siguiente código que se usa en la función stochastic_gradient_descent:

```
# Iteramos en los minibatches
for mini_batches_indixes in mini_batches_index_groups:
    # Tomo los datos y etiquetas asociadas a los indices de este minibatch
    minibatch_data = data[mini_batches_indixes]
```

```
minibatch_labels = labels[mini_batches_indixes]
```

Además, ahora que trabajamos con datos podemos aplicar la fórmula para el gradiente siguiente, que solo trabaja con un *subset* de los datos, y no con la muestra completa:

$$\nabla E_{in}(w) = \frac{2}{M} \sum_{n=1}^{M} x_n(h(x_n) - y_n)$$

Esta es la fórmula que usamos en la función calculate_gradient_from_data()