BÚSQUEDA ITERATIVA DE ÓPTIMOS Y REGRESIÓN LINEAL
MEMORIA P1 DE APRENDIZAJE AUTOMÁTICO

Ignacio Garach Vélez

2 de abril de 2022

Índice

1. Introducción **1**

2. Ejercicio sobre búsqueda iterativa de óptimos **2**
2.1. Algoritmo de Gradiente Descendente 2
2.2. Ejercicio 1.2 : Función $E(u,v) = (uve^{u^2v^2})^2$ 3
2.3. Ejercicio 1.3 : Función $f(x,y) = x^2 + 2y^2 + 2sen(2\pi x)sen(\pi y)$ 5

3. Ejercicio sobre Regresión Lineal **8**
3.1. Algoritmo de Gradiente Descendente Estocástico SGD 9
3.2. Algoritmo de Pseudoinversa 10
3.3. Aplicación : Clasificación de dígitos manuscritos 10
3.4. Transformación de errores al aumentar la complejidad del modelo 13

4. Algoritmo de Newton **16**
4.1. Aplicación a la función f 17

5. Bibliografía **18**

1. Introducción

En esta práctica vamos a abordar la implementación y aplicación de varios algoritmos fundamentales y básicos para el Aprendizaje automático.
En primer lugar se implementa el algoritmo de gradiente descendiente, se desarrolla su base teórica y para su aplicación nos alejamos por un momento de los problemas del aprendizaje automático y lo usamos para minimizar funciones arbitrarias en lugar de errores de ajuste. Entre otros asuntos, se tratará la importancia del punto inicial del algoritmo y del parámetro *learning rate* .

Para el segundo ejercicio, regresamos al ámbito del problema de aprendizaje. En este caso trataremos de ajustar modelos de regresión a vectores de características extraídos a partir de imágenes de dígitos manuscritos. Para ello se utilizarán tanto el algoritmo de gradiente descendente estocástico (mejora del implementado en el ejercicio anterior, que suele lograr

evitar óptimos locales) como la pseudoinversa, método de un paso. Estudiaremos también sus ventajas y limitaciones.

Continuaremos tratando de ajustar una nube de puntos generada aleatoriamente en un recinto acotado, con clase binaria asignada por una función dada. Para ello usaremos el modelo anterior y lo comparamos con otro modelo que utiliza características no lineales.

Finalmente, se estudia el algoritmo de Newton que extiende el algoritmo de gradiente descendente, utilizando la información que aportan las derivadas de segundo orden en el desarrollo en serie de la función de error. Comparamos su desenvoltura con los ejemplos del primer ejercicio.

2. Ejercicio sobre búsqueda iterativa de óptimos

2.1. Algoritmo de Gradiente Descendente

Este algoritmo generaliza una idea matemática básica, se sabe que la pendiente de una función coincide con su derivada en cada punto y esta marca el crecimiento o decrecimiento de la misma. Por tanto si avanzamos en la dirección opuesta a la derivada, nos acercaremos a un mínimo local (por tanto es crucial el punto de inicio). Para una función con una cantidad superior a 1 de variables, el gradiente generaliza el concepto de derivada, no es más que el vector de derivadas parciales con respecto a cada variable:

$$\nabla f(x_1, ..., x_n) = (\frac{\partial f(x_1, ..., x_n)}{\partial x_1}, ..., \frac{\partial f(x_1, ..., x_n)}{\partial x_n})$$

Por tanto, en este caso general, la idea es avanzar en una determinada proporción en la dirección opuesta al vector de dirección que determina el gradiente. Esta proporción será un parámetro del algoritmo al que llamamos tasa de aprendizaje, *learning rate*. En definitiva, a partir de una elección de punto de inicio(en principio arbitraria, aunque muy importante) en cada iteración vamos restando al punto el gradiente de la función en dicho punto multiplicado por la tasa de aprendizaje. Tenemos que considerar cuando terminar de iterar, ponemos como criterio, o bien alcanzar un número determinado de iteraciones o llegar a un error admisible.

Atendiendo a estas características, tendríamos el siguiente pseudocódigo:

GRADIENTE DESCENDENTE(Winicial, LearningRate, Gradiente, Función, Error Admisible, Máx. Iteraciones)

nº iteraciones= 0
$W = Winicial$
$Error = Función(W)$
Mientras Error > Error Admisible y no se alcanze el máximo de iteraciones:

Descender en la dirección negativa del gradiente y actualizar el error cometido.

$$W = W - LearningRate\dot{G}radiente(W)$$
$$Error = Función(W)$$
$$Iteraciones++$$

Fin-Mientras
Devolver W

Para después poder mostrar la evolución del punto solución, podemos almacenar en memoria el punto obtenido en cada iteración, en nuestro código lo llamaremos traza. De igual forma se puede ir almacenando como disminuye el error (en el contexto de este ejercicio, el valor de la función). Como desventajas de este algoritmo, se ha estudiado en teoría que el uso de todos los puntos para calcular el gradiente puede dificultar el alcance de un buen óptimo, así como ser costoso computacionalmente, utilizando muchas iteraciones. Además, puede darse el caso de que el entorno del punto de partida determine un gradiente que lleve al algoritmo a un mínimo local que puede ser muy lejano al mínimo global de la función.
En funciones convexas está garantizado por técnicas de análisis en varias variables que si se encuentrá un mínimo local, este es el mínimo global. Puede darse el caso si estamos utilizando el algoritmo para minimizar funciones pero en el caso del problema de aprendizaje la función del error en general no va a ser convexa, o al menos no tendremos forma de garantizarlo.

2.2. Ejercicio 1.2 : Función $E(u, v) = (uve^{u^2v^2})^2$

Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u, v) = (0.5, -0.5)$ y usando una tasa de aprendizaje $\eta = 0, 1$.

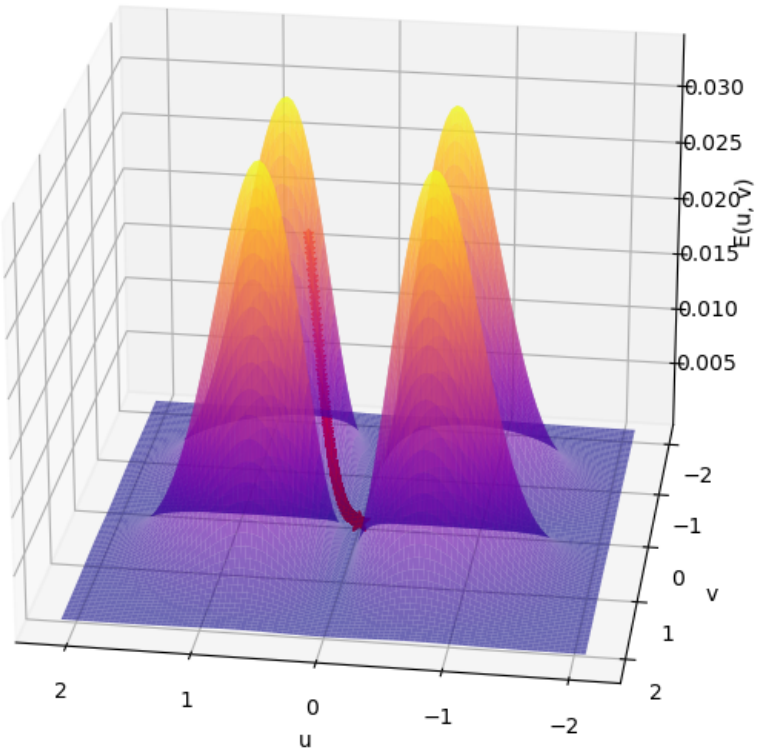
$$E(u, v) = (uve^{-u^2-v^2})^2$$

Calculamos, mediantes la regla de la cadena el gradiente de la función f, es decir, sus derivadas parciales, obteniéndose:

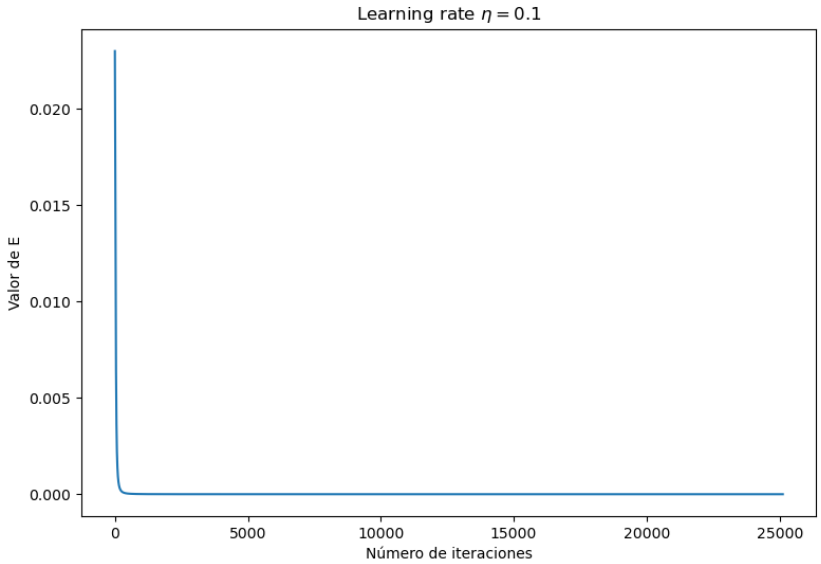
$$\nabla E(u, v) = (-2u(2u^2 - 1)v^2e^{-2(u^2+v^2)}, -2v(2v^2 - 1)u^2e^{-2(u^2+v^2)})$$

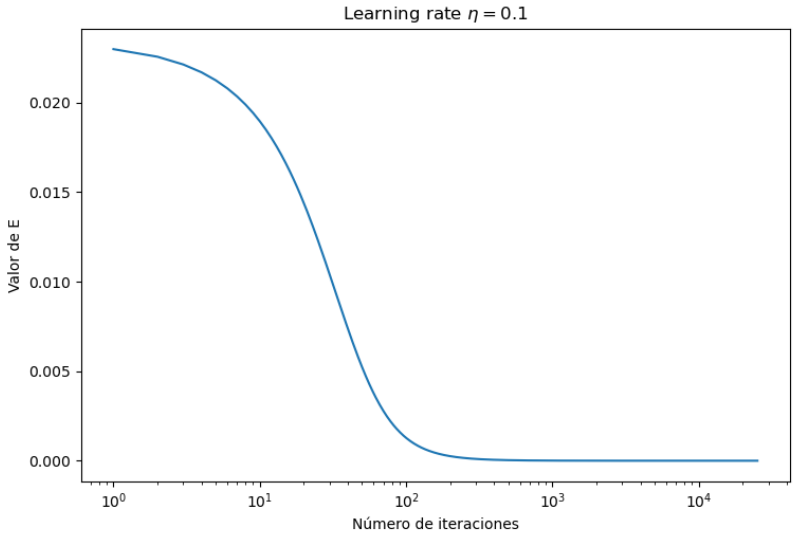
Lanzamos el algoritmo con dichas funciones definidas, una tasa de aprendizaje de 0.1 y el punto inicial $(0.5, -0.5)$ y obtenemos como salida garantizando un error máximo de 10^{-8} trás 25117 iteraciones el punto: $(0.010000842574554563, -0.010000842574554563)$

Podemos observar como va avanzando el algoritmo a través de la superficie en dirección al punto mínimo obtenido:

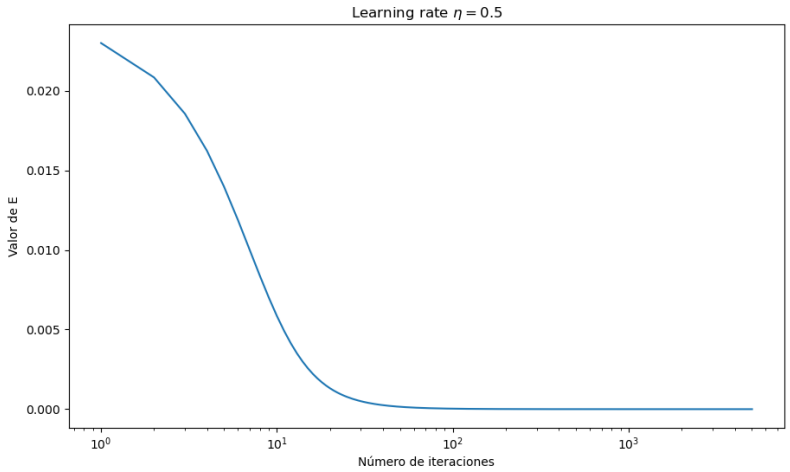


De igual forma, podemos mostrar como disminuye el valor de la función con las iteraciones, al comienzo es más clara la disminución, pero al final se mejora muy poco a poco, hasta alcanzar el error épsilon solicitado en el ejercicio, es más apreciable la disminución al final en la segunda gráfica, en la cuál se pone el eje de las iteraciones en escala logarítmica:





Podemos plantearnos ampliar la tasa de aprendizaje, y en este caso, por las características de localización del punto inicial y la superficie, esto es factible y mejora bastante la velocidad de convergencia a ese mismo punto mínimo. Para un $\eta = 0.5$, se alcanza el objetivo de error en 5020 iteraciones y a partir de la 50 el resultado ya es bastante prometedor.



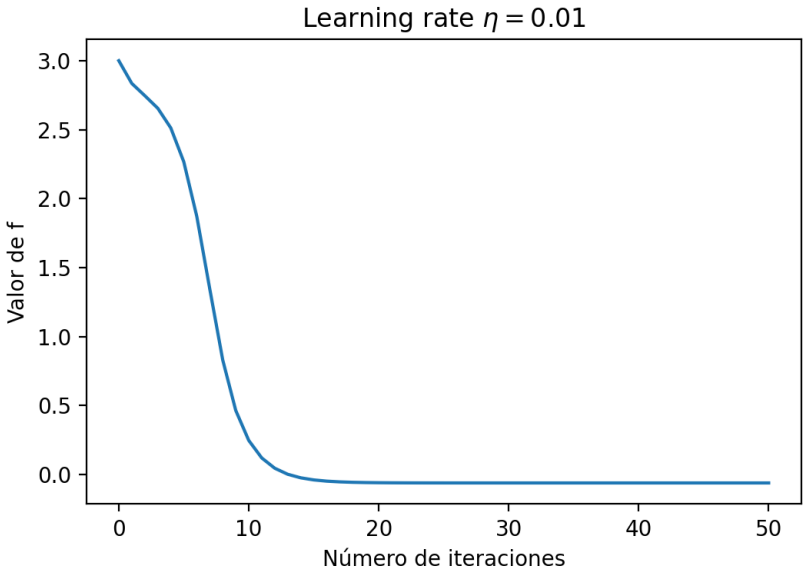
2.3. Ejercicio 1.3 : Función $f(x, y) = x^2 + 2y^2 + 2sen(2\pi x)sen(\pi y)$

Calculamos, mediantes la regla de la cadena el gradiente de la función f, es decir, sus derivadas parciales, obteniéndose:

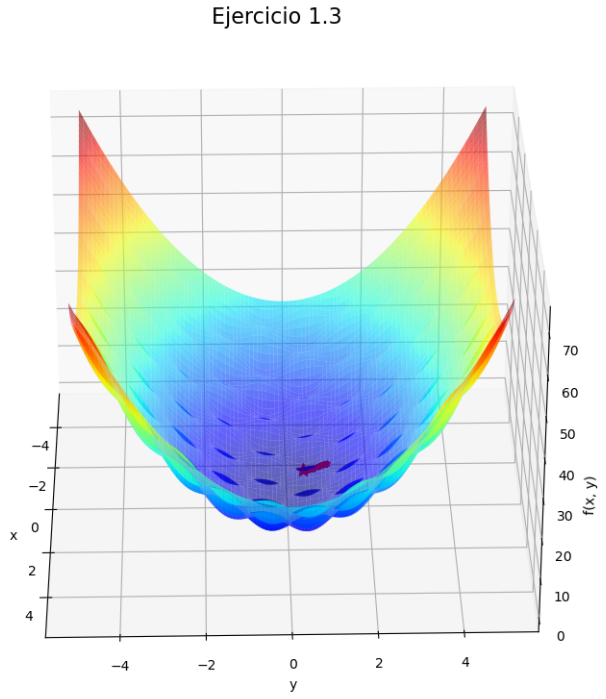
$$f(x, y) = x^2 + 2y^2 + 2sen(2\pi x)sen(\pi y)$$

$$\nabla f(x, y) = (2x + 4\pi sen(\pi y)cos(2\pi x), 4y + 2\pi sen(2\pi x)cos(\pi y))$$

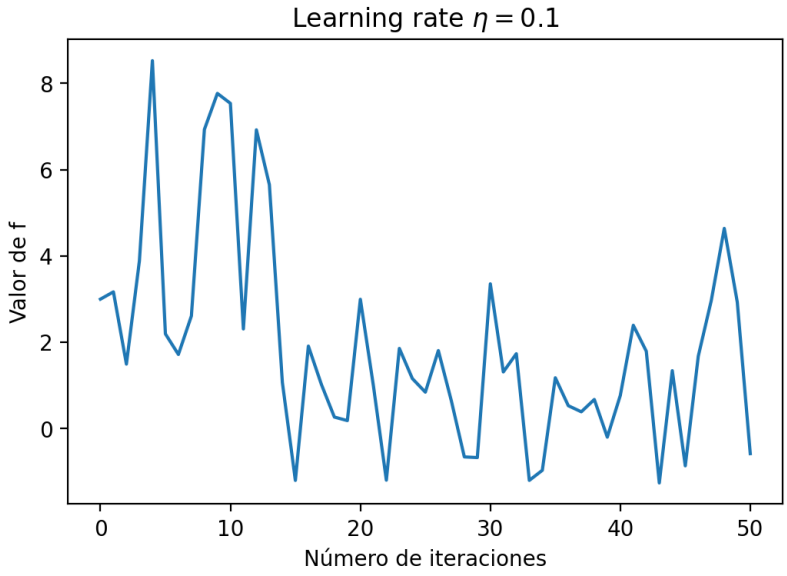
En este caso, lanzamos el algoritmo con las nuevas funciones, una tasa de aprendizaje de 0.01 y el punto $(-1, 1)$, obtenemos en 50 iteraciones el punto $(-1.2177, 0.4134)$. Podemos observar como va reduciendose el error conforme avanzan las iteraciones en el siguiente gráfico.



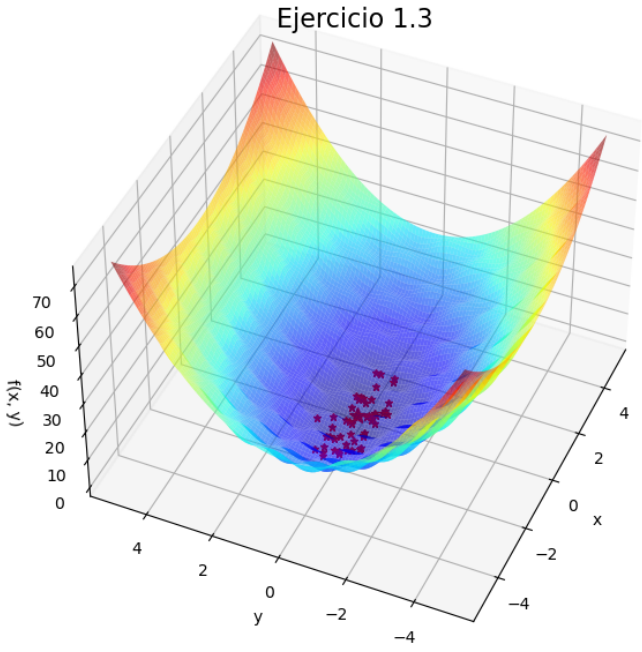
Y también comprobamos el desplazamiento del algoritmo a través de la superficie:



Si variamos el valor de la tasa de aprendizaje a $\eta = 0.1$, observamos en el gráfico correspondiente que el algoritmo empeora, pues en este caso el descenso no es monótono, de hecho, en torno a la décima iteración, se tiene un resultado más óptimo que el arrojado en la última iteración. Este hecho nos hace observar que la tasa de aprendizaje escogida es mala, pues avanza demasiado rápido en las direcciones que el gradiente va obteniendo y oscila mucho entre distintos mínimos.



El gráfico de avance sobre la superficie es por tanto mucho más caótico:



A continuación vamos a comparar el funcionamiento del algoritmo para distintos valores de inicio y con los 2 valores de la tasa de aprendizaje, se obtienen los siguientes valores:

Punto inicial	(-0.5, -0.5)	(1, 1)	(2.1, -2.1)	(-3, 3)	(-2, 2)
$\eta = 0.01$	$(-0.73, -0.414)$	$(-0.73, -0.414)$	$(1.66, -1.17)$	$(-2.188, 0.586)$	$(1.66, -1.17)$
$\eta = 0.1$	$(-1.60, 0.228)$	$(-0.625, 0.158)$	$(0.8, -0.33)$	$(-0.89, -0.13)$	$(-0.17, 0.286)$

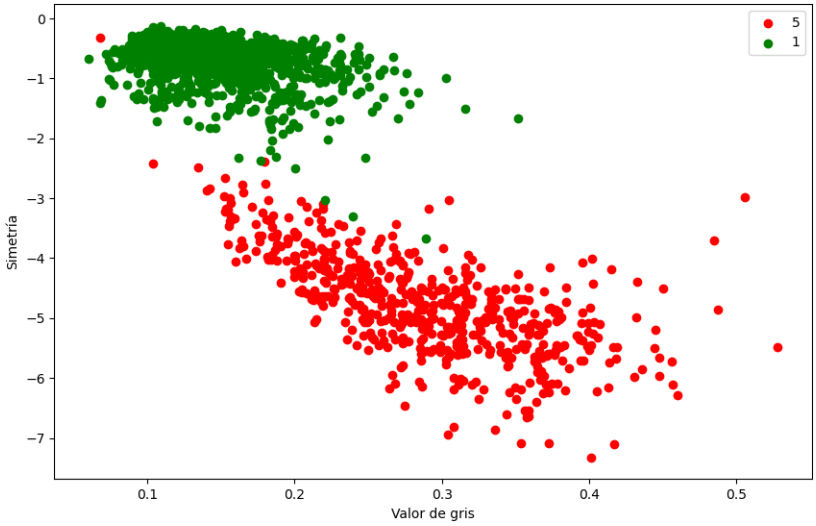
Punto inicial	(-0.5, -0.5)	(1, 1)	(2.1, -2.1)	(-3, 3)	(-2, 2)
$\eta = 0.01$	-1.0365	-1.0365	4.633	3.694	4.633
$\eta = 0.1$	3.511	1.12	2.498	0.326	-1.186

Estos resultados numéricamente no dicen mucho pero si que nos indican las desventajas de este algoritmo, la gran dependencia del punto inicial (obtenemos puntos muy distintos y mínimos mejores que otros). También observamos que en casos como el del punto inicial $(-3, 3)$, con la tasa pequeña se estanca en un mínimo local y con la mayor escapa a otro mejor. Aunque como hemos visto en el primer ejemplo de esta función aumentar la tasa también puede empeorar el resultado. Como conclusión obtenemos que para sacarle partido a este algoritmo es necesario tener cierta información sobre donde es mejor comenzar a iterar. Hemos visto también lo crucial que es la tasa de aprendizaje para conseguir un buen resultado (con distintas tasas se obtienen mínimos muy distintos). Además se confirma que no tenemos garantías de obtener el mínimo global si no hay condiciones de convexidad.

3. Ejercicio sobre Regresión Lineal

Este ejercicio ajusta modelos de regresión a vectores de características extraídos a partir de imágenes de dígitos manuscritos. En particular, se extraen dos características concretas que miden el valor medio del nivel de gris y la simetría del dígito respecto de su eje vertical. Solo se seleccionarán para este ejercicio las imágenes de los números 1 y 5. Despues en apartados posteriores, analizaremos el comportamiento al aumentar la complejidad de un modelo.

En primer lugar utilizaremos la función proporcionada para leer los datos desde fichero y vamos a obtener un gráfico de la distribución de los datos:



Podemos observar a priori que las dos clases están bastante delimitadas en 2 zonas del espacio, salvo algunos representantes de la clase 1 mas cercanos a la clase 5 y la situación opuesta en un punto aislado en la esquina superior izquierda. Podemos preveer que vamos a obtener un buen resultado con regresión lineal.

El problema de regresión lineal consiste en ajustar los coeficientes w para la clase de funciones lineales siguientes:

$$H = \{h : \mathbf{R}^{d+1} \to \mathbf{R} : h(x) = w^T x \qquad w \in \mathbf{R}^{d+1}\}$$

La solución que se utiliza de forma clásica consiste en minimizar la función de error cuadrático medio en la muestra, esto es:

$$E_{in}(w) = \frac{1}{N}\sum_{i=1}^{N}(w^T x_i - y_i)^2$$

Pasemos a explicar e implementar los algoritmos para esta sección:

3.1. Algoritmo de Gradiente Descendente Estocástico SGD

En este algoritmo vamos a tratar de disminuir las desventajas del algoritmo de gradiente descendiente básico, introduciendo un factor de aleatoriedad que nos permitirá evitar óptimos locales, a los que en el otro algoritmo podríamos haber estado destinados a caer por una mala elección de punto de partida. La idea es seleccionar un subconjunto de puntos (*mini-batch*) para calcular el gradiente del error en cada iteración. Esto nos aportará mayor variabilidad en el gradiente, pues se utilizan menos puntos en el promedio. Además es rápido de computar y se ha encontrado evidencia empírica de que en funciones no convexas se obtienen buenos mínimos locales.

La descripción en pseudocódigo del algoritmo es prácticamente la misma que antes, añadiendo la generación de los *mini-batches* y el cálculo del gradiente para estos.

Sin embargo, cambian los parámetros porque en este caso, se le pasarán directamente los datos y el algoritmo llamará a la función de error.

GRADIENTE DESCENDENTE ESTOCÁSTICO(X, Y, LearningRate, MiniBatchSize, Error Admisible, Máx. Iteraciones)

nº iteraciones= 0
Inicializo W con un vector de ceros
RandomOrder = Permutación aleatoria de tamaño el número de datos.
Inicio = 0
$Error = ErrorCuadráticoMedio(X, Y, W)$
Mientras Error > Error Admisible y no se alcanze el máximo de iteraciones:

Considero cada iteración el Minibatch de datos dados por las posiciones de RandomOrder:
MiniBatchPos = RandomOrder[Inicio hasta Inicio+MiniBatchSize]
Si no quedan datos para completar el tamaño de minibatch, se usan los que queden.
Inicio = Inicio + MiniBatchSize
Actualizo los pesos considerando sólo los puntos del MiniBatch para Calcular el Gradiente:
$W = W - LR * GradienteECM(X[MiniBatchPos], Y[MiniBatchPos], W)$
$Error = ErrorCuadráticoMedio(X, Y, W)$
Si se han considerado ya todos los datos:
Crear un nuevo RandomOrder.
Inicio = 0
$Iteraciones++$
Fin-Si

Fin-Mientras
Devolver W

3.2. Algoritmo de Pseudoinversa

Este algoritmo es directo, en el sentido que es de un sólo paso, la idea es obtener el vector de pesos w tal que el gradiente del error, se haga 0, es decir, en nuestro caso, que la función error cuadrático medio se minimice. Para ello, derivemos e igualemos a 0:

$$E_{in}(w) = \frac{1}{N}\|Xw - y\|^2$$

$$\nabla E_{in}(w) = \frac{2}{N}X^T(Xw - y) = 0 \Rightarrow X^T X w = X^T y$$

De donde el problema se reduce a calcular la inversa de $X^T X$ para despejar w, y si llamamos $X^\tau = (X^T X)^{-1} X^T$ a la pseudoinversa, se puede resumir el algoritmo en el despeje $w = X^\tau y$.

El vector de pesos resultado es la solución óptima desde el enfoque clásico del problema de regresión.

Consideramos para calcular la pseudoinversa que cualquier matriz admite una descomposición en valores singulares de la forma $X = UDV^T$ con U y V ortogonales Y D diagonal, por tanto $X^T X = VDDV^T$ y podemos calcular la pseudoinversa ahora de forma fácil usando la diagonal.

Explicado el fundamento teórico, la implementación mediante las librerías de cálculo numérico de Python es casi trivial con el siguiente pseudocódigo:

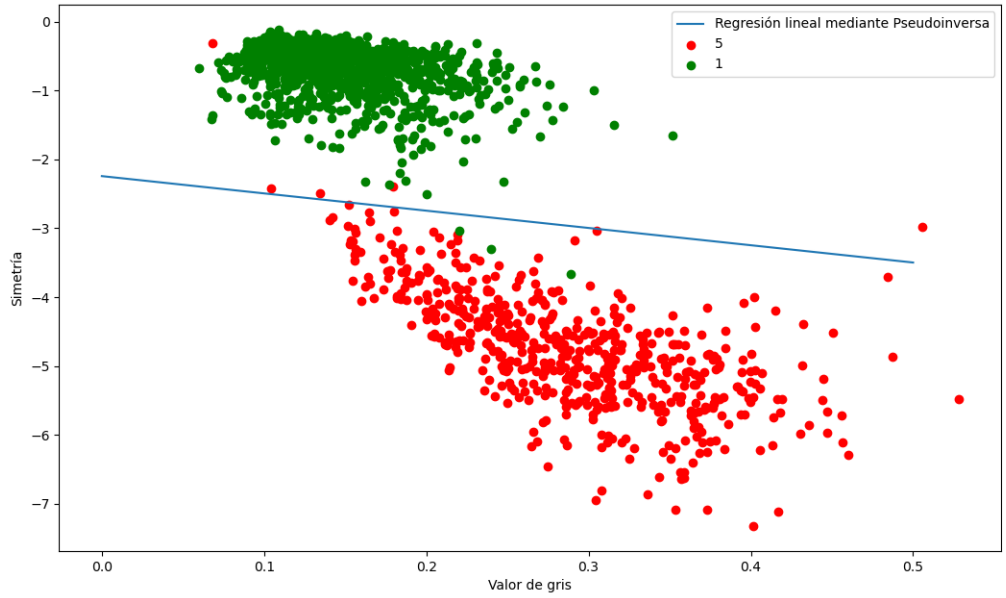
PSEUDOINVERSA(X, y)

Calcular descomposición en valores singulares.

Devolver $\mathrm{V}^T D^{-2} V X^T y$

3.3. Aplicación : Clasificación de dígitos manuscritos

A continuación vamos a ejecutar los algoritmos que acabamos de describir e implementar y comparar sus resultados, en primer lugar, lanzamos el algoritmo de pseudoinversa sobre los datos que debe arrojar el mejor resultado posible para este modelo lineal. Se obtiene el plano $y = -1.115880 - 1.248595x_1 - 0.497531x_2$ que si proyectamos en 2D haciendo y = 0 tiene este aspecto:



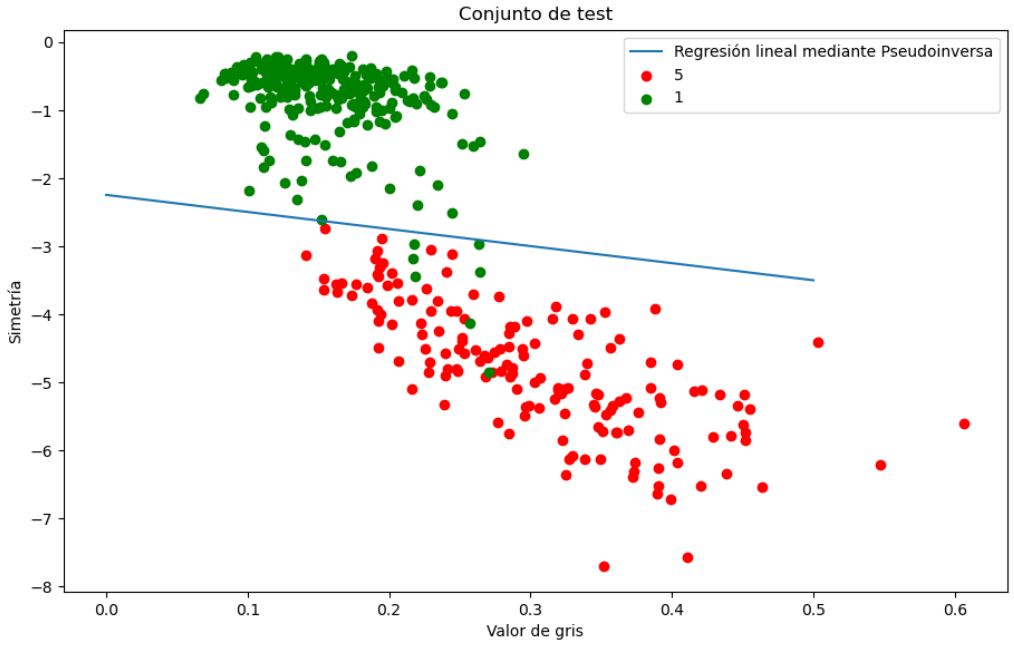
Calculamos además los errores en la muestra y fuera de ella haciendo uso del conjunto de test que se nos proporciona:

$$E_{in} = 0.079186$$

$$E_{out} = 0.130954$$

El error cuadrático medio mide la media de las diferencias de los valores esperados y los reales al cuadrado, por tanto cuanto más cerca este de 0, mejor.

Observamos que como era de esperar el error en la muestra es menor que en el conjunto de test, pues el modelo se ajustó con los primeros datos, sin embargo el resultado sobre el conjunto de prueba es razonablemente bueno. Mostramos a continuación los datos del conjunto de test junto con el modelo obtenido para observar que empeora ligeramente el error.



Vamos a introducir otra medida asociada al problema de clasificación binario que se está estudiando:

Exactitud (Accuracy): Porcentaje de casos en los que el modelo ha acertado.

Este algoritmo sobre el conjunto de test obtiene una exactitud del 98,34 % , como en el caso anterior dentro de la muestra este porcentaje se eleva al 99,48 %.

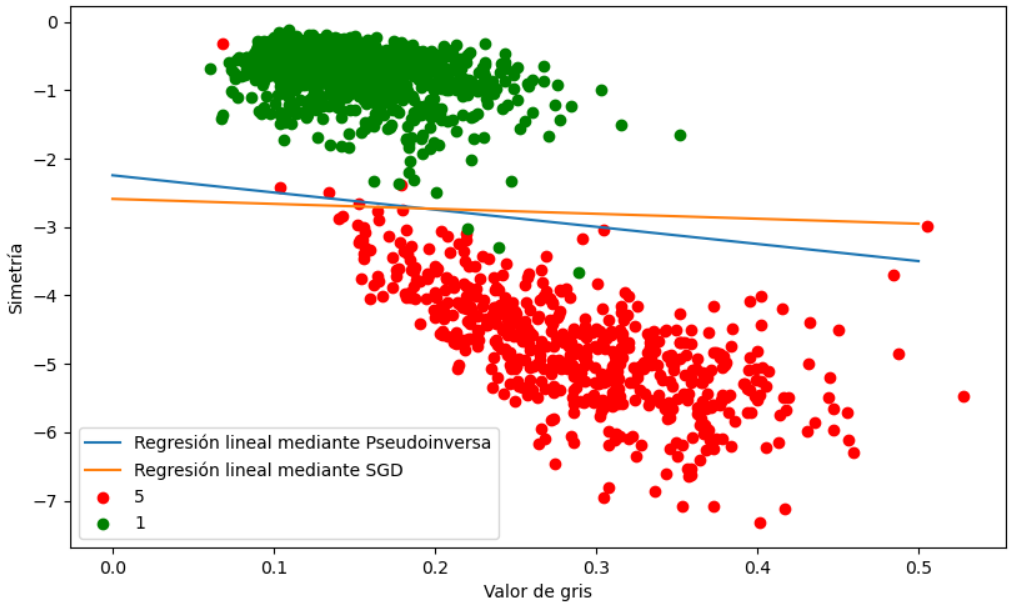
Pasamos ahora a utilizar gradiente descendente estocástico con distintos parámetros para ver como se comporta.

En primer lugar, utilizamos un $\eta = 0.01$ y tamaño de minibatch 32, obtenemos el plano $y = -1.22320489 - 0.34270032x_1 - 0.47264344x_2$ y los siguientes errores:

$$E_{in} = 0.0815$$

$$E_{out} = 0.1372$$

Aquí en el siguiente gráfico podemos ver la comparación de ambos algoritmos, la cual nos hace darnos cuenta que los resultados en cuanto a exactitud de la clasificación son exactamente los mismos que en pseudoinversa, hay 2 puntos en los que los clasificadores asociado a estas regresiones se confunden y aciertan complementariamente (uno en uno y otro en el otro):



Haciendo pruebas con diferente número de iteraciones, se comprueba que en torno a la iteración 70 ya converge a un clasificador igual de bueno para nuestro conjunto de test, aunque el error cuadrático medio de regresión es algo superior y sigue disminuyendo poco a poco.

Si lanzamos el algoritmo sobre minibatches de tamaño igual a toda la muestra (realmente estaríamos perdiendo la aleatoriedad del algoritmo y asumiendo la posibilidad de estancarnos en un mínimo local) conseguimos una regresión algo peor atendiendo a que aumentan los errores cuadráticos medios dentro y fuera de la muestra, pero el clasificador funciona igual de bien en nuestro conjunto de test.

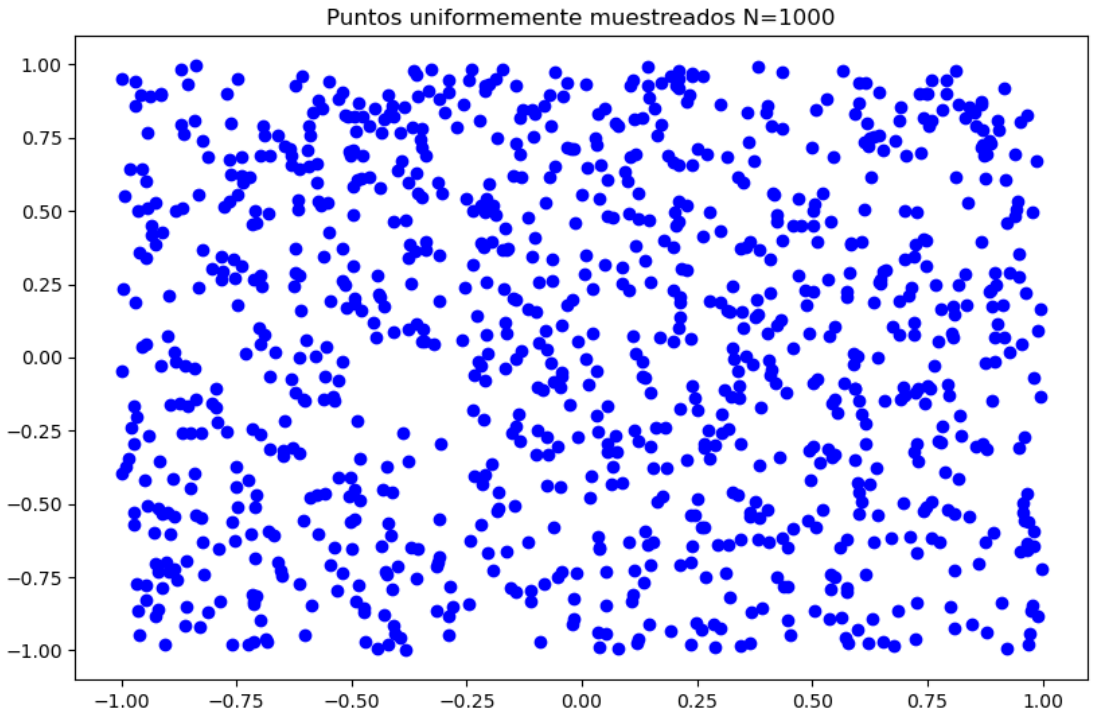
3.4. Transformación de errores al aumentar la complejidad del modelo

En este ejercicio vamos a utilizar una nube de puntos distribuidos uniformemente en el cuadrado $[-1, 1] \times [-1, 1]$ divididos en 2 clases distintas mediante la función siguiente:

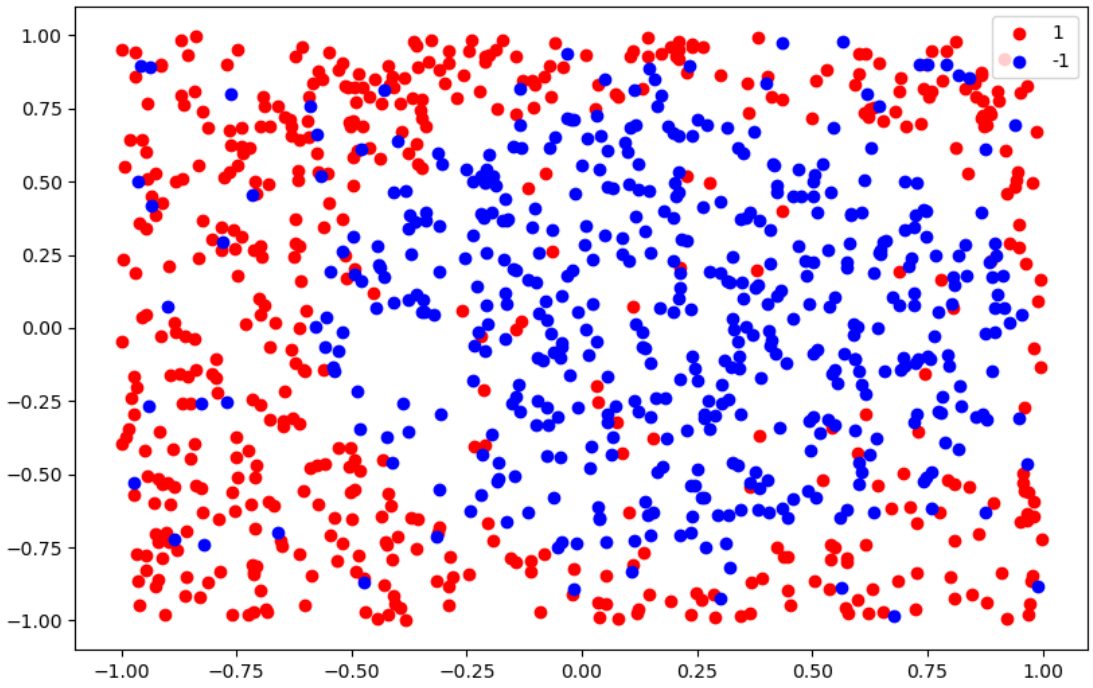
$$f(x_1, x_2) = sign((x_1 - 0.2)^2 + x_2^2 - 0.6)$$

Y el objetivo será medir la capacidad de aprender la función f de nuestros algoritmos con distintos modelos, sin embargo vamos a introducir un porcentaje de ruido del 10 % para complicar esta tarea, es decir con una probabilidad del 10 % modificaremos el signo de las etiquetas.

Utilizando la función *simulaunif* que se vale de la librería Numpy para generar la distribución uniforme, generamos la nube de puntos:



Aplicamos ahora vectorialmente la función f sobre los puntos uniformemente distribuidos para calcular las clases e introducimos el ruido en un 10 por ciento aleatorio de ellos, este es el resultado con las clases generadas:

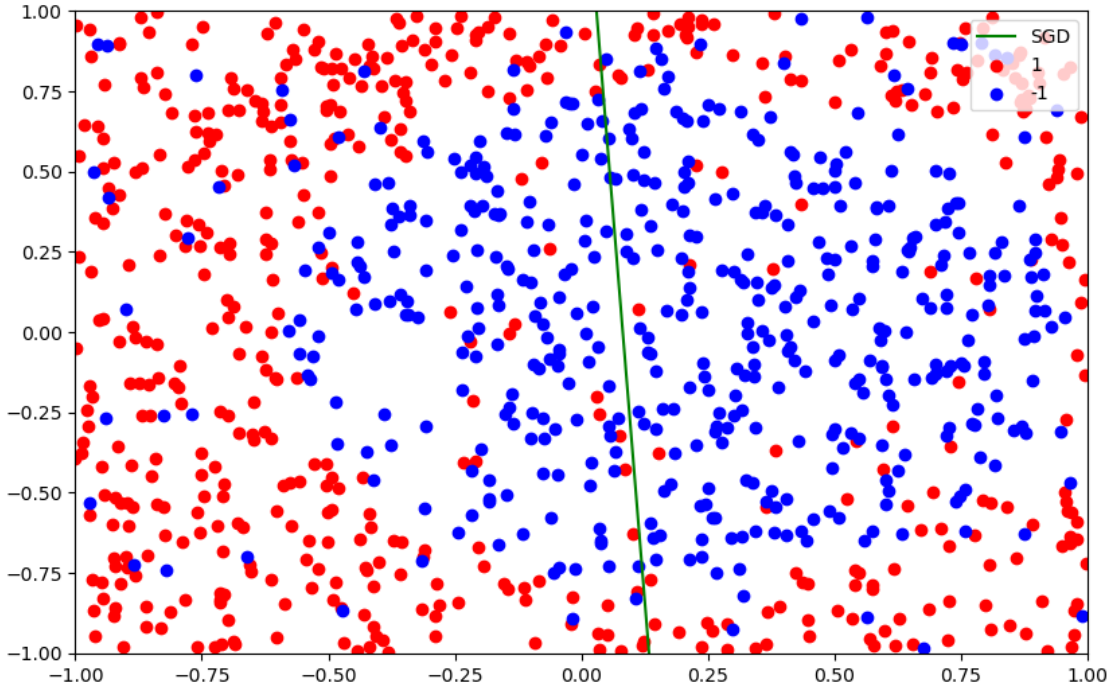


Aplicamos ahora gradiente descendente estocástico:
Con minibatches de tamaño 32 y una tasa de aprendizaje $\eta = 0.01$ se obtienen los siguientes valores:

$$E_{in} = 0.8946 \qquad\qquad Accuracy : 0.644$$

$$w = (0.04591997, -0.56060707, -0.02910647)$$

que nos muestran que el aprendizaje es muy malo, el *accuracy* está cercano al 0.5 que sabemos por teoría es el peor valor posible para un clasificador (valores cercanos a 1 son los mejores, pero si fueran cercanos al 0 bastaría invertir la predicción y los tendríamos cercanos al 1). Esto unido a la gráfica siguiente nos hace darnos cuenta que el modelo lineal elegido no es válido, los datos no son linealmente separables.



Para asegurarnos de que hemos obtenido resultados correctos y representativos vamos a repetir el experimento 1000 veces, para ello se utilizará la función *generaMuestraNueva* que automatiza la generación de la distribución uniiforme, el cálculo de las clases y la introducción de ruido. Finalmente introduce la columna de unos para el término afín que es necesaria por como está diseñado el algoritmo.
Para realizar estas ejecuciones usamos un bucle que genera 2 muestras una para entrenamiento y otra para test, ajusta los pesos con la primera y calcula los errores tanto con ella(E_{in}) como con la muestra de test(E_{out}).

Los resultados son los siguientes en promedio:

$E_{in} = 0.9268$ $E_{out} = 0.9324$ $Accuracy_{in} : 0.6028$ $Accuracy_{out} : 0.6003$

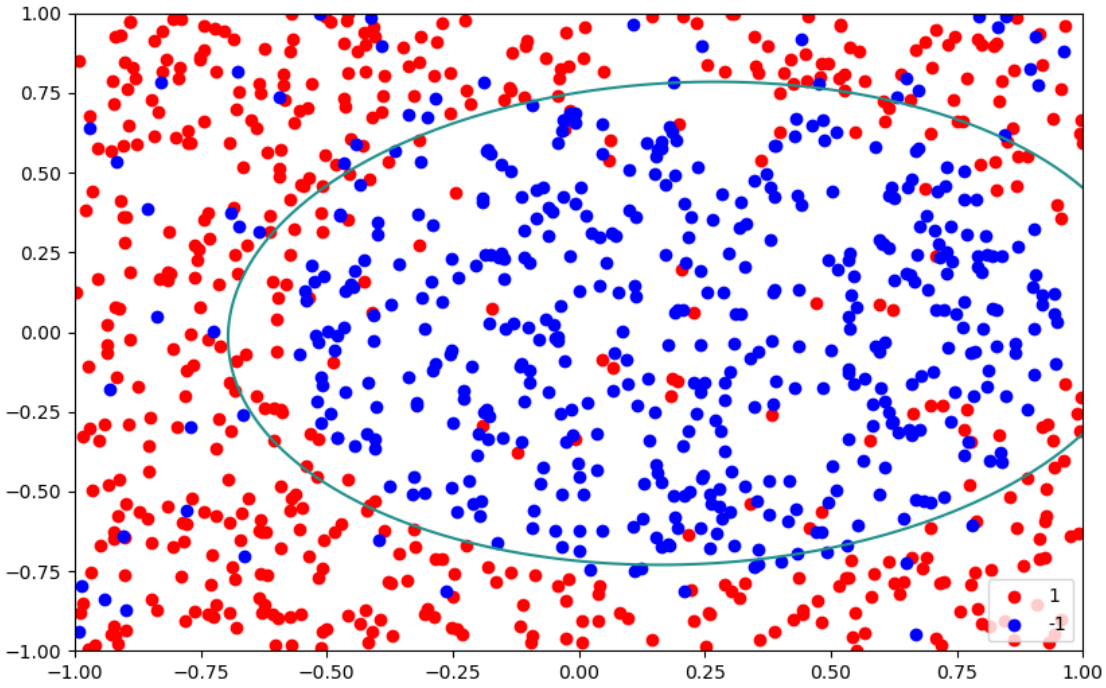
Estos datos reafirman lo que vimos con la primera ejecución, el modelo lineal no se comporta correctamente con estos datos, los errores son muy altos tanto dentro como fuera de la muestra y la exactitud es muy mala también. Es necesario buscar otro tipo de modelos que realicen mejor la labor de clasificación para nuestros datos. De hecho, ahora trabajaremos con un modelo que utiliza un vector con algunas características no lineales.

Cambiamos el enfoque, ahora usaremos el vector de características siguientes:

$$\phi(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$$

El algoritmo no necesita ser adaptado para estas características, bastará por tanto generar las muestras como antes y agregarles las nuevas características, en mi código utilizo la función *generaMuestraYNoLineales* que llama a la utilizada al ejercicio anterior, calcula las columnas con el producto de las características y sus cuadrados y devuelve los datos preparados para que el algoritmo pueda utilizarlos.

Aplicando SGD con minibatches de tamaño 32, obtenemos los siguientes resultados, que a simple vista muestran gran mejoría:



$E_{in} = 0.5840$ $Accuracy : 0.844$

Para dibujar la gráfica, hemos calculado los valores del "polinomio"que resulta de la regresión sobre un *linspace* y hacemos uso de la funcion *contour* para poder mostrarlo en el plano coordenado x1, x2.

Volvemos a repetir el experimento 1000 veces para obtener unos resultados más fiables, el sistema utilizado es el mismo que en el caso de regresión lineal:

$E_{in} = 0.5800$ $E_{out} = 0.5875$ $Accuracy_{in} : 0.8573$ $Accuracy_{out} : 0.8550$

Observamos que los errores tanto dentro como fuera de la muestra han descendido en gran medida al aumentar la complejidad a este modelo de regresión. Podemos por tanto concluir que para esta función este modelo es más indicado que el anteriormente aplicado. Por otro lado, observamos que no conseguimos obtener resultados extremadamente buenos, esto es debido al porcentaje de ruido que introdujimos al generar los datos, es puramente aleatorio, por tanto no logramos aprenderlo. La exactitud de la clasificación siempre va a estar limitada por este porcentaje de ruido.

4. Algoritmo de Newton

En este apartado final, se discute la implementación del método de Newton en el cual se utiliza información asociada a curvatura, es decir, usamos las derivadas de segundo orden. El método se resume en la siguiente ecuación que se usa para obtener ceros de funciones:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

que aplicado a funciones de varias variables resulta en:

$$x_{n+1} = x_n - H_f(x_n)^{-1} f'(x_n)$$

siendo H_f la matriz hessiana con las derivadas segundas de la función. Podemos añadirle también una tasa de aprendizaje multiplicando para ayudar a que converja. Por tanto, el algoritmo es igual que el de gradiente descendente simple pero cambiando la línea en la que se actualizan los pesos.

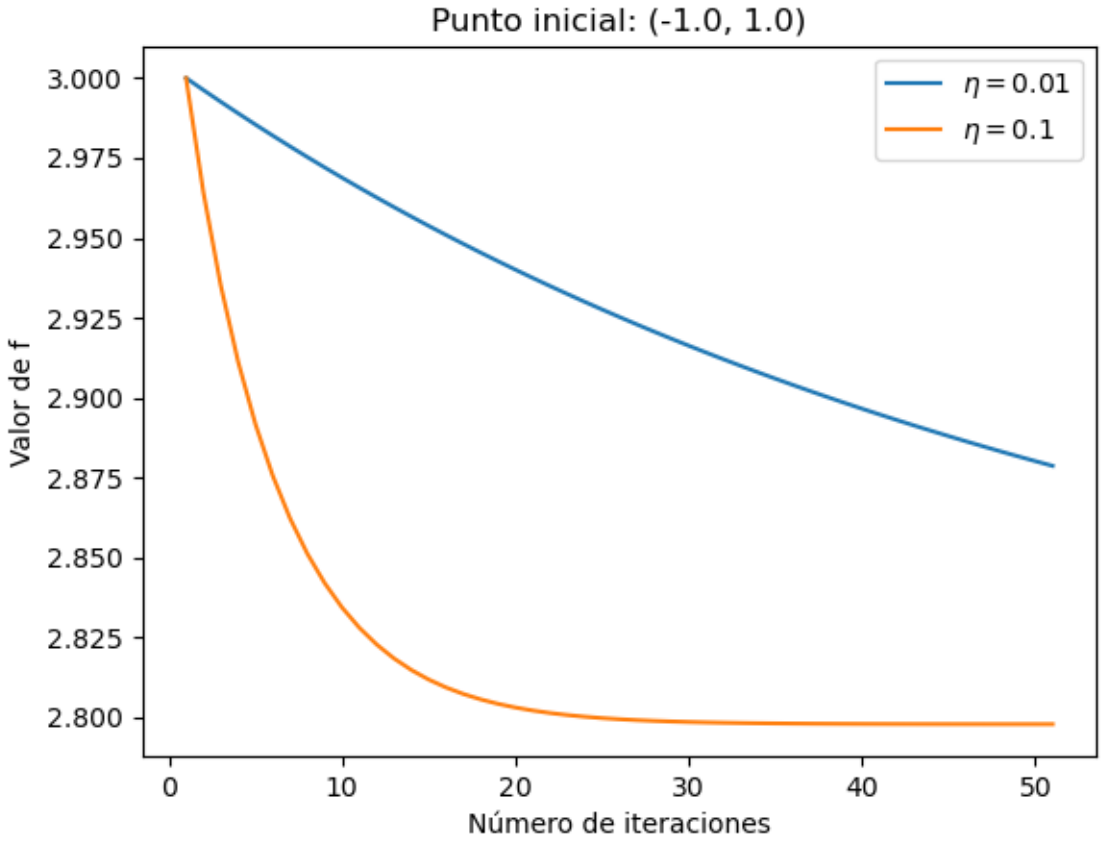
4.1. Aplicación a la función f

Para poder aplicarlo calculamos y definimos las derivadas segundas de f, obteniendo la siguiente matriz hessiana:

$$H_f(x,y) = \begin{pmatrix} \frac{\partial^2 f}{\partial^2 x} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial^2 y} \end{pmatrix} = \begin{pmatrix} 2 - 8\pi^2 sin(\pi y) sen(2\pi x) & 4\pi^2 cos(\pi y) cos(2\pi x) \\ 4\pi^2 cos(\pi y) cos(2\pi x) & 4 - 2\pi^2 sin(\pi y) sen(2\pi x) \end{pmatrix}$$

Podemos ya pasar a realizar los experimentos del ejercicio 1.

Partiendo del punto $(-1.0, 1.0)$ con distinta tasa de aprendizaje se obtienen las siguientes soluciones:
w = (1.2177, 0.4134)



Comparando entre ellos para este caso, parece que la segunda tasa es mejor porque consigue alcanzar el mínimo antes, con la tasa $\eta = 0.01$ si le damos más capacidad de iterar continua mejorando hasta estancarse en el mismo punto que la otra tasa.

Si entramos a comparar con el gradiente descendente, el resultado es peor pues con él se conseguía llegar a un mínimo bastante mejor, practicamente se alcanzaba el 0. Si ampliamos la tasa a $\eta = 0.1$, el algoritmo mejora ligeramente pero se estanca en un mínimo local con valor 2.8. Si imprimimos el hessiano durante las iteraciones, observamos que la matriz no es definida positiva, los determinantes totales son negativos, además la inversa toma valores cercanos al 0, que unidos al producto por la tasa de aprendizaje pueden hacer que el algoritmo se estanque.

Repetimos el experimento con distintos puntos iniciales y tasas obteniendo estos resultados:

Punto inicial	(-0.5, -0.5)	(1, 1)	(2.1, -2.1)	(-3, 3)	(-2, 2)
$\eta = 0.01$	-0.718	3.142	21.45	25.675	12.624
$\eta = 0.1$	4.236	3.246	254.5	10.742	12.471

Volvemos a mostrar los mínimos obtenidos por gradiente descendente para comparar:

Punto inicial	(-0.5, -0.5)	(1, 1)	(2.1, -2.1)	(-3, 3)	(-2, 2)
$\eta = 0.01$	-1.0365	-1.0365	4.633	3.694	4.633
$\eta = 0.1$	3.511	1.12	2.498	0.326	-1.186

Confirmamos nuestras suposiciones, en todos los puntos, mas unos que otros, el mínimo obtenido es peor con Newton que con gradiente descendente, esto es debido a que para la función no convexa que estamos considerando se dan las peores condiciones para el mismo, la hessiana no es definida positiva y los múltiples cambios de curvatura de la función hacen que el producto de la matriz por el gradiente se haga muy cercano a 0, no permitiendo al algoritmo avanzar bien. En conclusión, para este caso no es buena idea usar el método de Newton, pues ademas de encontrar malos mínimos, ocasiona más gasto computacional y requiere que la función sea derivable hasta el segundo orden.

5. Bibliografía

Fuentes de consulta on-line, en general documentación de las librerías:

1. Numpy https://numpy.org/doc/stable/reference/generated/numpy.hstack.html

2. https://www.tutorialspoint.com/matplotlib/matplotlib_contour_plot.html

3. https://numython.github.io/posts/2016/02/graficas-de-contorno-en-matplotlib/