UNIVERSIDAD DE GRANADA

Aprendizaje Automático

CURSO 2018/2019

Trabajo 1: Programación

Laura Rabadán Ortega

1. EJERCICIO SOBRE LA BÚSQUEDA ITERATIVA DE ÓPTIMOS

Gradiente Descendente

1. Implementar el algoritmo de gradiente descendente

Para implementar el algoritmo, se ha partido de la fórmula presentada en las diapositivas de teoría.

$$w_j := w_j - \eta \frac{\partial E_{in}(\mathbf{w})}{\partial w_j}$$

Ilustración 1 - Ecuación General

El gradiente descendente recibe como parámetros de entrada:

- La función que se quiere minimizar.
- El gradiente de la función anterior.
- Los valores de los variables de la función, los pesos, iniciales.
- El valor de la constante de aprendizaje.
- La cota que se quiere alcanzar, es decir, el valor máximo que se quiere conseguir resolviendo la función con los pesos. Si no se especifica este valor, se le asigna un valor muy bajo para que se gaste el número de iteraciones.
- El número máximo de iteraciones. Si no se especifica, se establece un máximo de 100 iteraciones.

Por otro lado, la función devuelve:

- El número de iteraciones gastadas.
- Los pesos finales, resultado de haber conseguido un valor menor a la cota o de haber gastado el número de iteraciones.
- Una matriz con:
    - El número de iteración, que servirá como identificador.
    - Los valores de los pesos resultados de esa iteración.
    - El valor de la función en cada iteración con los pesos conseguidos en esa iteración.

El algoritmo ejecuta la ecuación de la Ilustración 1 mientras el valor de la función sea mayor o igual que la cota o mientras queden iteraciones. Si se cumple alguna de las condiciones anteriores, se para el bucle y se devuelven los pesos definitivos y el resumen de cada iteración.

En cada iteración, se modifican los pesos como muestra la ecuación. Se llama al gradiente de la función con los pesos de esa iteración y se va minimizando el valor de la función. A su vez, se calcula el valor de la función con los nuevos pesos.

2. Considerar la función $E(u,v) = (u^2e^v - 2v^2e^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0.01$.
    a. Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u,v)$.

Partiendo de la función $E(u,v) = (u^2e^v - 2v^2e^{-u})^2$, el gradiente de $E(u,v)$ es:

$$\frac{dE(u,v)}{du} = 2(u^2e^v - 2v^2e^{-u})(2ue^v + 2v^2e^{-u})$$

$$\frac{dE(u,v)}{dv} = 2(u^2e^v - 2v^2e^{-u})(u^2e^v - 4ve^{-u})$$

$$\nabla E(u,v) = \left[\frac{dE(u,v)}{du} \quad \frac{dE(u,v)}{dv}\right]$$

    b. ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u,v)$ inferior a 10^{-14}. (Usar flotantes de 64 bits)

Se consigue por primera vez un valor menor que 10^{-14} tras 33 iteraciones. Exactamente, el valor que se consigue es de 6×10^{-15}, aproximadamente, como se muestra en la Ilustración 2.

    c. ¿En qué coordenadas (u,v) se alcanzó por primera vez un valor igual o menor a 10^{-14} en el apartado anterior?

Se consigue en un valor menor a 10^{-14} con $(u,v) = (0.619208, 0.968448)$.

```
        EJERCICIO 1 - Ejercicio sobre la búsqueda iterativa de óptimos

Apartado 2 - Función E(u,v):

        Valor conseguido:  5.997300838629042e-15

        Número de iteraciones requeridas:  33

        Coordenadas para el primer valor < 10^(-14): (0.619208,0.968448)
```

Ilustración 2 - Resultado de ejecución

Laura Rabadán Ortega

3. Considerar ahora la función $f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$
    a. Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = 0.1, y_0 = 0.1)$, (tasa de aprendizaje $\eta = 0.01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0.1$, comentar las diferencias y su dependencia de η.

Se genera un único grafo con los resultados de las dos iteraciones, Ilustración 3. En él, se observa que con una tasa de aprendizaje mayor, de 0.1 en este caso, el valor de la función sufre cambios bastante bruscos, es decir, pasa de valores muy bajos a valores muy altos. En cambio, con una constante de aprendizaje más pequeña, 0.01 en este ejercicio, se observa que el valor se mantiene constante tras un número de iteraciones.

Elegir la constante de aprendizaje es muy importante a la hora de intentar minimizar una función. Cuando el valor de esta constante es muy grande, como pasa en este ejemplo al aplicar una constante de 0.1, puede ocurrir que al ejecutar el Gradiente Descendente se salte el mínimo. Es decir, que se consigan valores mayores a los anteriores. Esto se puede ver en la gráfica. En cada iteración se pasa, en la mayoría de los casos, de un valor a otro mucho menor o mayor. Cuando se aplica una constante de aprendizaje grande, se recorre la función de forma más rápida, situación que podría interesar en momentos determinados, pero se corre el riesgo de saltarse el mínimo.

Por otro lado, elegir una constante de aprendizaje más pequeña, como 0.01, produce un desplazamiento más lento por la función, pero también más cuidadoso. Como puede verse en el gráfico, se acerca al mínimo y se mantiene en él. Este tipo de valores son interesantes para cuando se está cerca del mínimo.

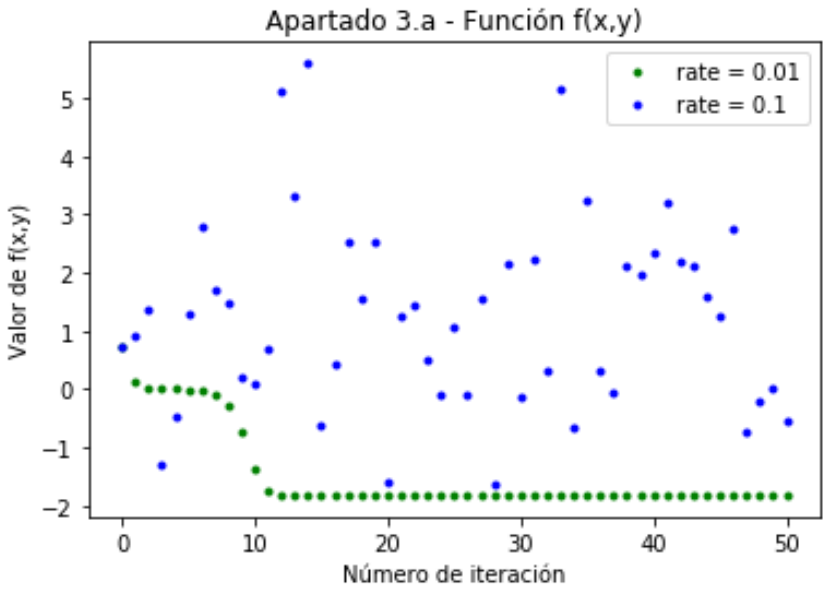

Ilustración 3 - Comparación de Learning Rate

Laura Rabadán Ortega

b. Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija: $(0.1, 0.1), (1, 1), (-0.5, -0.5), (-1, -1)$. Generar una tabla con los valores obtenidos.

La tabla de valores obtenidos, generada a partir de los resultados de ejecución y con una constante de aprendizaje de 0.01, es la siguiente:

Punto de Inicio	Valor Mínimo	Valores de (x, y)
(0.1, 0.1)	-1.820079	(0.243805, -0.237926)
(1, 1)	0.593269	(1.21807, 0.712812)
(-0.5, -0.5)	-1.332481	(-0.731377, -0.237855)
(-1, -1)	0.593269	(-1.21807, -0.712812)

Tabla 1

```
Valores obtenidos en el apartado 3.b (tabla en la memoria):
        -> ( 0.100000,  0.100000)      : -1.820079     : ( 0.243805, -0.237926)
        -> ( 1.000000,  1.000000)      : 0.593269      : ( 1.218070, 0.712812)
        -> ( -0.500000,  -0.500000)    : -1.332481     : ( -0.731377, -0.237855)
        -> ( -1.000000,  -1.000000)    : 0.593269      : ( -1.218070, -0.712812)
```

Ilustración 4 - Resultado de ejecución

4. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

En mi opinión, la dificultad a la hora de encontrar el mínimo global de una función es encontrar un buen punto de inicio y elegir una buena constante de aprendizaje.

El punto de inicio es importante, ya que dependiendo el punto que se elija, puede encontrarse el óptimo global o caer en un óptimo local, por lo que el resultado dependerá mucho de este valor inicial.

La constante de aprendizaje determina como moverse por la función. Si es un valor muy grande, cómo se comentó en un apartado 3.b, podría salirse de ese óptimo global.

Por tanto, la verdadera dificultad está en fijar un buen punto de inicio y controlar el desplazamiento por la función con una constante de aprendizaje correcta.

2. EJERCICIO SOBRE REGRESIÓN LINEAL

1. Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números (Intensidad promedio, Simetría) usando tanto el algoritmo de la pseudoinversa como Gradiente descendente estocástico (SGD). Las etiquetas serán$\{-1, 1\}$, una para cada vector de cada uno de los números. Pintar las soluciones obtenidas junto con los datos usados en el ajuste. Valorar la bondad del resultado usando E_{in} y E_{out} (para E_{out} calcular las predicciones usando los datos del fichero de test). (Usar $Regress\_Lin(datos, label)$ como llamada para la función (opcional)).

Para llevar a cabo este ejercicio, se leyeron los datos de entrenamiento y *test* de los ficheros *"features_train.npy"* y *"features_test.npy"*, respectivamente. Una vez se leen los datos, se modifican las etiquetas de los grupos, de manera que en vez de tener como etiquetas *{1, 5}*, se tiene *{-1, 1}.* Es decir, el grupo 1 pasa a ser ahora el grupo -1 y el grupo 5 el 1.

A parte de cambiar las etiquetas, otra modificación que se lleva a cabo es la de añadir un valor adicional a cada conjunto de datos. Este valor es 1, y se agrega para poder operar de manera correcta con los pesos, ya que hay uno que es constante, y es el que se multiplicaría por este valor. Esta columna de unos se añade al final de la matriz, es decir, al final de cada conjunto de datos. Por tanto, los pesos se presentan como $(x_1, x_2, 1)$.

Los errores producidos en el proceso de estimación de pesos, tanto dentro como fuera de la muestra, se llevan a cabo con las funciones E_{in} y E_{out}, respectivamente. Esas funciones son:

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

En un principio, estas funciones se implementaron de forma incorrecta, calculando el error respecto a la etiqueta. Es decir, en vez de calcular la diferencia de la etiqueta respecto a $w^T x_i$, se calculaba con respecto a la etiqueta que se le correspondía en función al resultado del producto anterior. Al corregir este error, se consiguieron unos errores peores que en el primer caso, pero el error conseguido dentro de la muestra, datos de entrenamiento, era mejor que el error en los datos fuera de ella, datos del test. Por esto, se mantiene la última versión, la correcta, pero se presentan en este documentos ambos resultados, identificando como *"Caso 1"* la primera implementación, la errónea, y como *"Caso 2"* los resultados de la implementación correcta.

Para llevar a cabo la predicción, una vez conseguidos los pesos, se multiplica las características del dato de entrada, modificado de la manera especificada en el segundo párrafo, por los pesos obtenidos. Dependiendo del valor conseguido, se le asignaba una clase u otra, en función del signo del resultado. Si era positivo, se le asignaba la clase 1 (5) y si era negativo, la clase -1 (1). Estos datos de entrada, como se explica en el párrafo anterior, están previamente etiquetados, por lo que se calcula el error medio obtenido a la hora de etiquetar. El objetivo es lograr un error mínimo tanto en los datos de entrenamiento como en los datos del test.

Para la implementación del algoritmo del Gradiente Descendente Estocástico, se parte de los conceptos estudiados en teoría: no se trabaja con toda la muestra al mismo tiempo sino que se divide en distintos *mini-batches* y se va trabajando con cada uno de ellos. En cada iteración se mezclan los datos y se escoge el primer *mini-batch*, conjunto de datos con los que se va a trabajar. De resto es similar al Gradiente Descendente. Es más, en vez de crear una nueva función para este algoritmo, podría haberse utilizado la función del Gradiente Descendente, utilizando una cota de 0, un máximo de iteraciones de 1 y un *mini-batch* como datos de entrada, utilizando un bucle que controlara las condiciones de parada. Pero, para mejor comprensión, se optó por definir una nueva función.

Como se ha especificado, el algoritmo es similar al Gradiente Descendente, a diferencia de que en cada nueva iteración, mezcla los datos de entrada y elige los primeros x datos, siendo x un valor acorde al número de datos de la muestra. En este ejercicio se utiliza un tamaño de *batch* de 64. El resto del algoritmo funciona de la misma manera.

Los valores de los pesos conseguidos tras un número máximo de 50 iteraciones y un *learning rate* de 0.01 son los de la Ilustración 5. Como puede leerse, no se consigue un error de 0, lo cual sería lo idóneo, si un error próximo a 0, en el caso del error en los datos de entrenamiento, y bastante inferior a 1, en el caso del error con los datos del test.

- Caso 1:

```
-> Learning Rate:  0.01
-> Pesos finales (x1, x2, 1): (-0.000941, -0.002253, -0.003701)
-> Error dentro de la muestra:  0.058936579115951314
-> Error fuera de la muestra:   0.2358490566037736
```

Ilustración 5 - Resultados de SGD (Caso1)

- Caso 2:

```
-> Learning Rate:  0.1
-> Pesos finales (x1, x2, 1): (-0.232943, -0.252194, -0.287884)
-> Error dentro de la muestra:  0.5076707786360438
-> Error fuera de la muestra:   0.5308782949627328
```

Ilustración 6 - Resultados de SGD (Caso 2)

Como puede notarse, el valor de la constante de aprendizaje ha cambiado, pasando de 0.01 a 0.1. Esto es debido a que, al realizar la ejecución con la implementación correcta, el error conseguido era pero que con un *Learning Rate* de 0.1, como puede apreciarse en la Ilustración 7.

```
-> Learning Rate:  0.01
-> Pesos finales (x1, x2, 1): (-0.045582, -0.046337, -0.047882)
-> Error dentro de la muestra:  0.8697307045771477
-> Error fuera de la muestra:   0.8690667801910719
```

Ilustración 7 - Resultados de SGD con 0.01 (Caso 2)

En el primer caso, Ilustración 5, el error en la muestra (con los datos de entrenamiento) es bastante menor al error con los datos del test. En cambio, en el segundo caso, los errores son más similares.

El resultado gráfico al predecir las clases de los datos de entrenamiento es el que se ve en la Ilustración 8, caso 1, y en la ilustración 9, caso 2. No todos los datos están etiquetados de forma correcta, como dice el error, pero la mayoría cuenta con la etiqueta correcta.

- Caso 1:

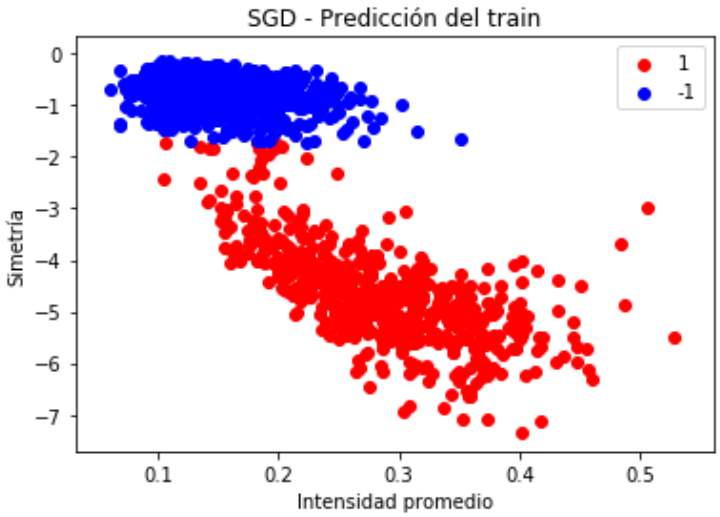

Ilustración 8 – SGD (Caso 1)

- Caso 2:

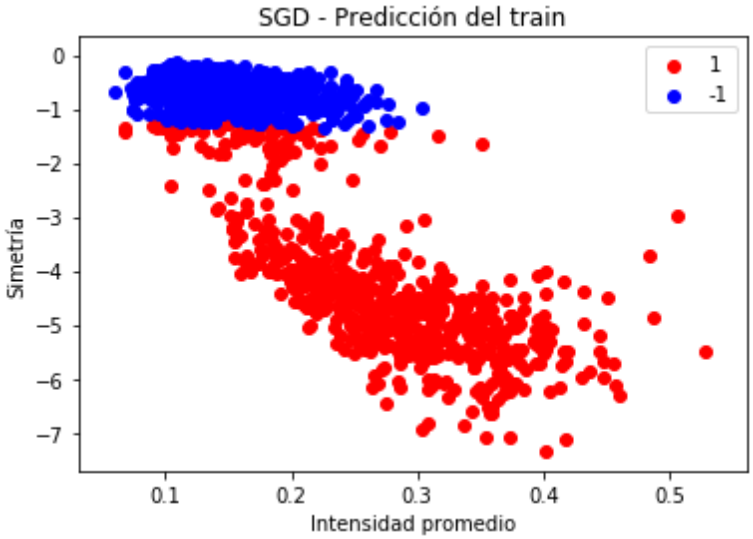

Ilustración 9 - SGD (Caso 2)

Como puede verse en las gráficas, error de asignación de etiquetas es mucho menor en el primer caso, en el segundo es menos preciso.

Para aplicar el algoritmo de la pseudoinversa se llevó a cabo la descomposición en valores singulares de la matriz de datos de entrada. Con este algoritmo se consigue mucho menos error, es más preciso a la hora de estimar los pesos. En ambos casos consigue errores muy buenos.

- Caso 1:

```
-> Pesos finales (x1, x2, 1): (-1.248595, -0.497532, -1.115880)
-> Error dentro de la muestra:  0.020499679692504803
-> Error fuera de la muestra:   0.0660377358490566
```

Ilustración 10 - Resultados de la Pseudoinversa (Caso 1)

- Caso 2:

```
-> Pesos finales (x1, x2): (-1.248595, -0.497532, -1.115880)
-> Error dentro de la muestra:  0.07918658628900375
-> Error fuera de la muestra:   0.13095383720052553
```

Ilustración 11 - Resultados Pseudoinversa (Caso 2)

En la Ilustración 12 y en la Ilustración 13, puede verse dos agrupaciones claras que podía intuirse en el gráfico anterior.

- Caso 1:

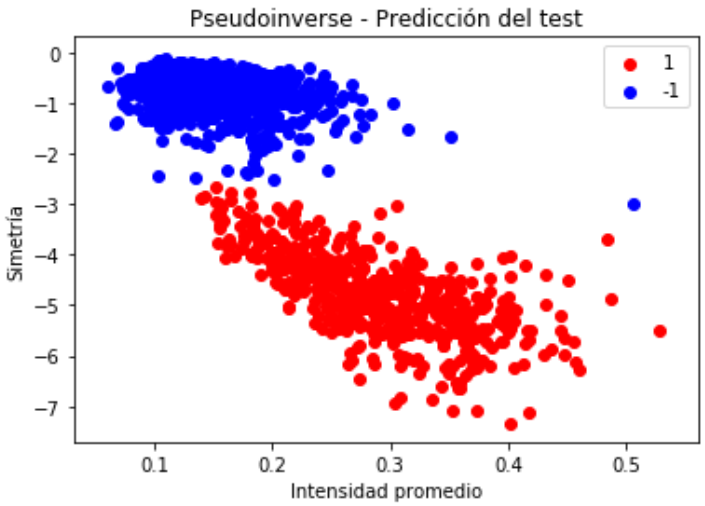

Ilustración 12 – Pseudoinversa (Caso 1)

- Caso 2:

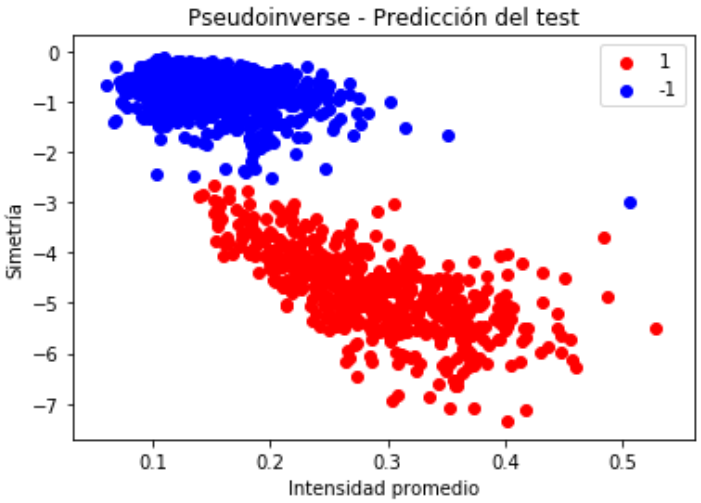

Ilustración 13 - Pseudoinversa (Caso 2)

Si se comparan ambos métodos, Gradiente Descendente Estocástico frente al método de la Pseudoinversa, puede afirmarse que es este segundo el que proporciona mejor resultados. Además, no es necesario establecer una constante de aprendizaje ni un punto de inicio, parámetros que interfieren a la hora de encontrar el óptimo global.

El problema de este método es el coste que supone. Es decir, se trabaja directamente con matrices, descomponiéndolas, transportándolas, etc. Si se tienen pocos datos, como en este caso, estas operaciones apenas suponen un coste grande. En cambio, si aumentan las dimensiones de la matriz, el coste computacional de este método aumentará.

Por tanto, si se tienen pocos datos puede trabajarse con el método de la inversa, pero si hay mayor cantidad, es preferible trabajar con el Gradiente Descendente Estocástico, en este ejemplo. No se está diciendo que este segundo método sea malo, sino que requiere un mayor estudio para unos mejores resultados.

Este ejercicio se agrupa en una única función, *RegressLin*, que recibe como parámetros de entrada los datos de entrenamiento y los datos del test.

2. En este apartado exploramos como se transforman los errores Ein y Eout cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función $simula\_unif(N, 2, size)$ que nos devuelve N coordenadas $2D$ de puntos uniformemente muestreados dentro del cuadrado definido por $[-size, size] \times [-size, size]$.

La función *simula_unif*, aparte de crear *N* coordenadas *2D* de puntos uniformemente muestreados entre *[-size, size] x [-size, size]*, acondiciona los datos para poder utilizar el SGD. Es

decir, le asigna las etiquetas según lo especificado en el siguiente apartado b y añade el valor para calcular el valor constante.

- EXPERIMENTO:

  a. Generar una muestra de entrenamiento de $N = 1000$ puntos en el cuadrado $X = [-1, 1] \times [-1, 1]$. Pintar el mapa de puntos $2D$. (ver función de ayuda)

El mapa de puntos generados es el que se muestra en la Ilustración 14.

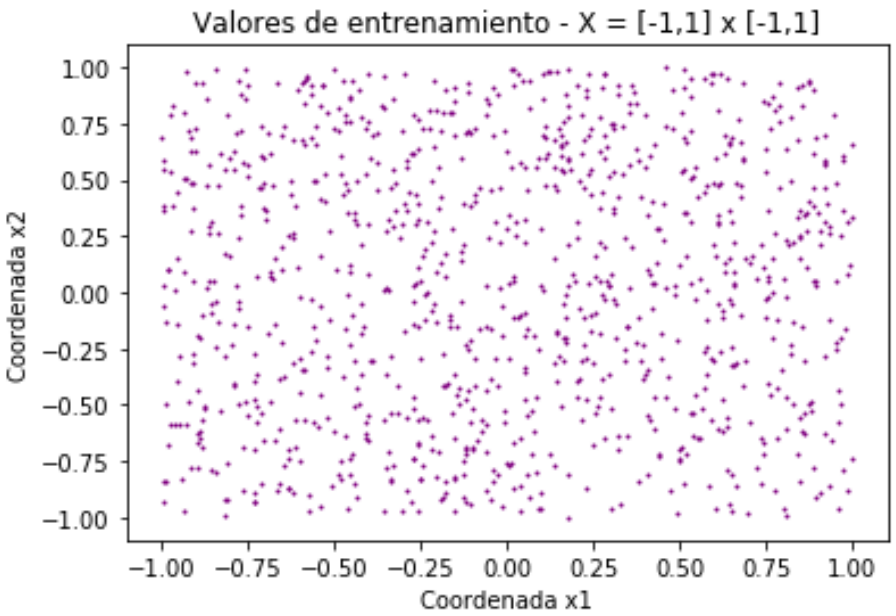

Ilustración 14 - Mapa de puntos 2D

  b. Consideremos la función $f(x_1, x_2) = sign((x_1 - 0.2)^2 + x_2^2 - 0.6)$ que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10% de las mismas. Pintar el mapa de etiquetas obtenido.

Esta función asigna una clase, *{-1, 1}*, a cada conjunto de datos. Al 10% de los datos, se le cambia la etiqueta para generar ruido. El mapa de etiquetas obtenido es el que se muestra en la Ilustración 15.

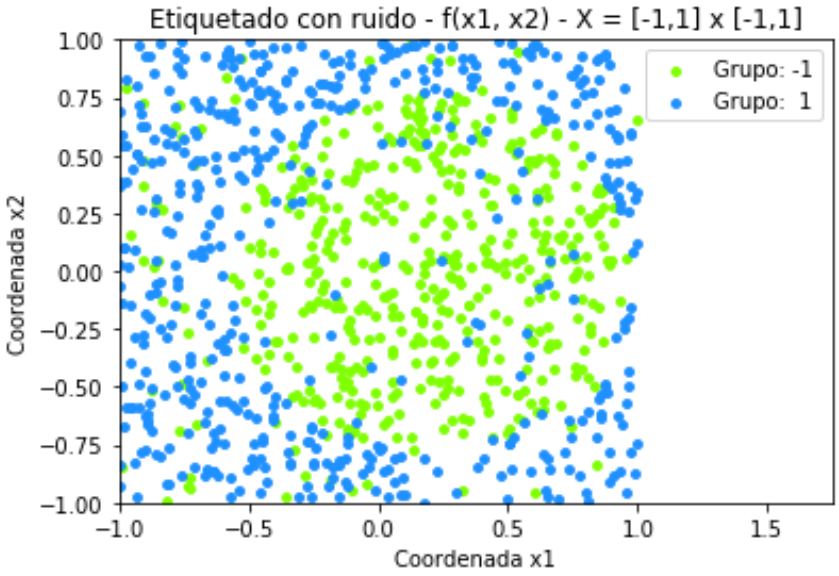

Ilustración 15 - Mapa de etiquetas

c. Usando como vector de características $(1, x_1, x_2)$ ajustar un modelo de regresión lineal al conjunto de datos generado y estimar los pesos w. Estimar el error de ajuste E_{in} usando Gradiente Descendente Estocástico (SGD).

Se llama a la función del Gradiente Descendente con una constante de aprendizaje de 0.01, una cota de 1 y un máximo de 100 iteraciones. Como se ve en la Ilustración 15, es difícil conseguir una clara diferenciación de clases de forma lineal, por ello, el error que se va a conseguir es alto, debido a esa dificultad. El error conseguido dentro de la muestra es de 1.744, en el caso 1, y de 0.999 en el caso 2. A la dificultad de encontrar los pesos para la clasificación debe añadírsele el ruido introducido en los datos. Por ello, el encontrar unos valores que minimicen el error resulta más complicado.

- Caso 1:

```
Pesos:  [-0.00603015 -0.00464215  0.00066277]
Error de ajuste:  1.744
```

Ilustración 16 - Resultado de ejecución (Caso 1)

- Caso 2:

```
Pesos:  [-0.00603015 -0.00464215  0.00066277]
Error de ajuste:  0.9985652055557047
```

Ilustración 17 - Resultado de ejecución (Caso 2)

d. Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes) y

   - Calcular el valor medio de los errores E_{in} de las 1000 muestras.
   - Generar 1000 puntos nuevos por cada iteración y calcular con ellos el valor de E_{out} en dicha iteración. Calcular el valor medio de E_{out} en todas las iteraciones.

Los apartados de a, b y c se recogen dentro de una única función, *Experimento*, por tanto, para este apartado se llama 1000 veces a esta función, pero con 10 iteraciones en el SGD. En cada iteración, se generan datos para el entrenamiento y datos para el test, y se van acumulando los errores de los dos conjuntos. Una vez se acaba el total de iteraciones, se calcula el error medio y se muestra por pantalla. El resultado de dicha ejecución sería el mostrado en la Ilustración 18, para el caso 1, y en la Ilustración 19, para el caso 2.

- Caso 1:

```
Error medio en la muestra:       1.5485320000000018
Error medio fuera de la muestra:         1.564516000000018
```

Ilustración 18 - Resultado de ejecución (Caso 1)

- Caso 2:

```
Error medio en la muestra:       0.9993005796522009
Error medio fuera de la muestra:         0.9995044596324849
```

Ilustración 19 - Resultado de ejecución (Caso 2)

En este caso si se consigue un error menor en la muestra que fuera de ella. Es importante destacar que la ejecución de este apartado, según se cronometró, tarda alrededor de 7 minutos.

e. Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de E_{in} y E_{out}.

Después de los experimentos realizados se observa que, si se tiene una muestra de datos "perfecta", es decir, sin fallos de etiquetado, y tiene una clara diferenciación lineal, como es en el caso del apartado 1 de este ejercicio, este modelo resulta bastante bueno, con un error, en general, próximo a 0.

En cambio, de cara a los resultados de este último experimento, donde los errores medios son bastante altos, como puede observarse en las ilustraciones anteriores, no es el mejor modelo aplicable, ya que no es capaz de determinar unos buenos valores para los pesos.

2.1. EJERCICIO SOBRE REGRESIÓN LINEAL

1. **Método de Newton** Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio.3. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

   - Generar un gráfico de como desciende el valor de la función con las iteraciones.
   - Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.

Para poder implementar el Método de Newton, debe definirse las segundas derivadas de la función $f(x, y)$, las cuales son:

$$\frac{df(x,y)}{dx^2} = 2 - 8\pi^2 \sin(2\pi y)\sin(2\pi x) \qquad \frac{df(x,y)}{dxdy} = 8\pi^2 \cos(2\pi x)\cos(2\pi y)$$

$$\frac{df(x,y)}{dy^2} = 4 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) \qquad \frac{df(x,y)}{dydx} = 8\pi^2 \cos(2\pi y)\cos(2\pi x)$$

Según muestra el material de teoría, la ecuación para encontrar los pesos es la mostrada en la Ilustración 13. Este algoritmo es similar al Gradiente Descendente, con la diferencia de que en vez de multiplicar el gradiente del error por una constante de aprendizaje, se multiplica por la matriz Hessiana del error, es decir, la matriz que recoge las segundas derivadas de la función de error.

$$\hat{\boldsymbol{\beta}}_{k+1} = \hat{\boldsymbol{\beta}}_{k} - \boldsymbol{H}^{-1} \frac{\partial E(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\Big|_{\widehat{\boldsymbol{\beta}_i}}$$

Esta es la idea principal. Las gráficas generadas en este apartado son las presentadas en la Ilustración 20. Las gráficas presentadas en la Ilustración 21 son las conseguidas en la ejecución del Gradiente Descendente.

```
        -> Gráfica 1 - pesos: (7326064188726601737830 4.000000,
4800424982431148697190 4.000000)
        -> Gráfica 2 - pesos: (5233237984038391210381724876 80.000000,
-2989976822474814356502484090 88.000000)
        -> Gráfica 3 - pesos: (-1463770905904040976911958016.000000,
-8649431639120537376509657088.000000)
        -> Gráfica 4 - pesos: (-5233237984038391210381724876 80.000000,
2989976822474814356502484090 88.000000)
```

Ilustración 20 - Pesos conseguidos

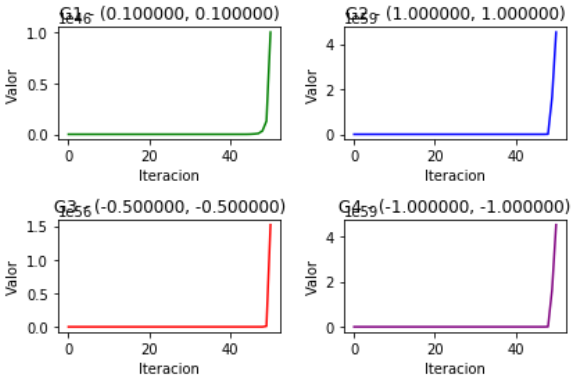

Ilustración 21 - Método de Newton

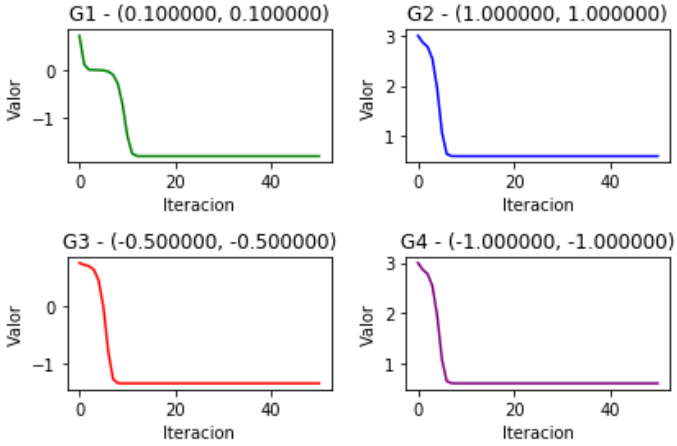

Ilustración 22 - Gradiente Descendente

Como puede observarse, los resultados del Método de Newton son bastante ilógicos. Se muestra un resultado inverso al que se quiere conseguir, el valor de la función aumenta respecto al número de iteraciones, en vez de disminuir, como pasa al ejecutarlo con el Gradiente Descendente.

Por este motivo, se probó una versión de este método, añadiendo un *Learning Rate*, es decir, multiplicando el Gradiente del Error por la matriz Hessiana y por la constante de aprendizaje. En este caso, se consiguieron los pesos de la Ilustración 22, algo que de primeras ya parece más creíble, y las gráficas de la Ilustración 23, con tendencias similares a las obtenidas por el Gradiente Descendente.

```
-> Gráfica 1 - pesos: (0.049107, 0.048901)
-> Gráfica 2 - pesos: (0.965198, 0.990651)
-> Gráfica 3 - pesos: (-0.482715, -0.495408)
-> Gráfica 4 - pesos: (-0.965198, -0.990651)
```

Ilustración 23 - Pesos resultantes

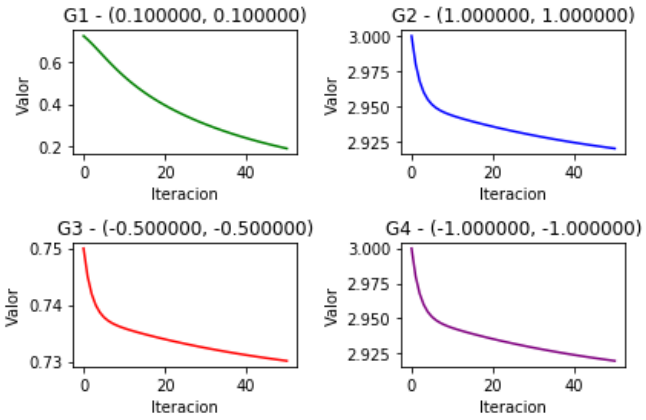

Ilustración 24 - Método de Newton con Learning Rate

Si se estudian las gráficas del método de Newton en comparación a las del Gradiente descendente, puede comprobarse que el método de Newton recórrela función de manera más lenta, por lo que 50 iteraciones no les son suficientes para alcanzar el mínimo valor, como si pasa en el Gradiente Descendente. Es por este que el método de Newton suele utilizarse cuando ya se está cerca del mínimo, para alcanzarlo con más precisión.

3. BIBLIOGRAFÍA

The Scipy community (2017). Numpy.matrix.transpose. Recuperado el 8 de marzo de 2019, de https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.matrix.transpose.html

The Scipy community (2017). Numpy.linalg.qr. Recuperado el 16 de marzo de 2019, de https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.qr.html

The Scipy community (2019). Numpy.linalg.inv. Recuperado el 16 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html

The Scipy community (2019). Numpy.fill_diagonal. Recuperado el 16 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.fill_diagonal.html

The Scipy community (2019). Numpy.diag. Recuperado el 16 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.diag.html

The Scipy community (2019). Numpy.linalg.pinv. Recuperado el 18 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.pinv.html

Unicoos (2012). Vector Gradiente UNIVERSIDAD unicoos matemáticas derivadas parciales. Recuperado el 18 de marzo de 2019, de https://youtu.be/hQjPW5hsU2o

The Scipy community (2019). Numpy.dot. Recuperado el 18 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html?highlight=dot#numpy.dot

Tutorialspoint. Python – Functions. Recuperado el 18 de marzo de 2019, de https://www.tutorialspoint.com/python/python_functions.htm

The SciPy comuunity (2019). Numpy.transpose. Recuperado el 18 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.transpose.html

Python Software Foundation (2019). 9.2. match – Mathematical functions. Recuperado el 18 de marzo de 2019, de https://docs.python.org/2/library/math.html

Scherfgen, D. (2019). Calculadora de Derivadas. Recuperado el 18 de marzo de 2019, de https://www.calculadora-de-derivadas.com

The SciPy comuunity (2018). Numpy.sign. Recuperado el 22 de marzo de 2019, de https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.sign.html

Python Software Foundation (2019). Random – Generate pseudo-random numbers. Recuperado el 22 de marzo de 2019, de https://docs.python.org/3/library/random.html#random.randint

Uned. Normas (APA) para las referencias bibliográficas. Recuperado el 24 de marzo de 2019, de https://www2.uned.es/simposioconstruirfuturos/documentos/Normas%20APA.pdf

DataCamp. Python For Data Science Cheat Sheet NumPy Basic. Recuperado el 24 de marzo de 2019, de https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf

DataCamp. Python For Data Science Cheat Sheet Matplotlib. Recuperado el 24 de marzo de 2019, de https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Matplotlib_Cheat_Sheet.pdf

The SciPy community (2019). Numpy.linspace. Recuperado el 24 de marzo de 2019, de https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html

Wikipedia (2019). Sign function. Recuperado el 24 de marzo de 2019, de https://en.wikipedia.org/wiki/Sign_function

The SciPy community (2018). Numpy.sign. Recuperado el 24 de marzo de 2019, de https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.sign.html

Matplotlib development team (2017). Color example code: named_colors.py. Recuperado el 24 de marzo de 2019, de https://matplotlib.org/examples/color/named_colors.html

Wikipedia (2019). Matriz hessiana. Recuperado el 24 de marzo de 2019, de https://es.wikipedia.org/wiki/Matriz_hessiana

Scipython. Vstack and hstack. Recuperado el 24 de marzo de 2019, de https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/