Aprendizaje automático

Trabajo 1
Programación

Johanna Capote Robayna
5 del Doble Grado en Informática y Matemáticas
Grupo A

UNIVERSIDAD
DE GRANADA

Índice

1. Gradiente descendente

Este ejercicio se desarrolla en el archivo `p1_ej1.py`.

1. Implementar el algoritmo de gradiente descendente.

   Para implementar el algoritmo definimos la función `gd` a la cuyos argumentos son:

   - **w**: punto inicial del algoritmo
   - **lr**: tasa de aprendizaje.
   - **grad_fun**: gradiente de la función.
   - **fun**: función sobre la que itera el algoritmo.
   - **epsilon**: umbral de parada del algoritmo.
   - **max_iters**: número de máximo de iteraciones que realiza el algoritmo en caso de que no se satisfaga los criterios de parada.

```
def gd(w, lr, grad_fun, fun, epsilon, max_iters = MAX_ITER):
  for it in range(max_iters):
    if(fun(w) < epsilon):
      break
    w = w - lr * grad_fun(w)

  return w, it
```

2. Considerar la función $E(u,v) = (ue^v - 2ve^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0{,}1$.

   a) Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u,v)$.

   Calculamos las derivadas parciales de la función $E(u,v) = (ue^v - 2ve^{-u})^2$:

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

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

b) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de E(u, v) inferior a 10^{14}?

Ejecutamos el algoritmo de gradiente descendente con punto inicial $(1, 1)$, tasa de aprendizaje $\eta = 0{,}1$ y $\epsilon = 10^{-14}$. Tras la ejecución vemos que tarda **10 iteraciones** en obtener un valor inferior a 10^{-14}.

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

En las coordenadas $(0.0447, 0.0239)$ se alcanzó por primera vez un valor menor o igual a 10^{-14}.

3. Considerar ahora la función $f(x, y) = (x - 2)^2 + 2(y + 2)^2 + 2\sin(2\pi x)\sin(2\pi y)$

En primer lugar calculamos las derivadas parciales de la funcion $f(x, y)$

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

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

a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = 1, y_0 = 1)$, (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.

Ejecutamos en primer lugar el algoritmode gradiente descendente con punto inicial $(1, -1)$, tasa de aprendizaje $\eta = 0.01$ y un máximo de 50 iteraciones, obteniendo la siguiente gráfica:

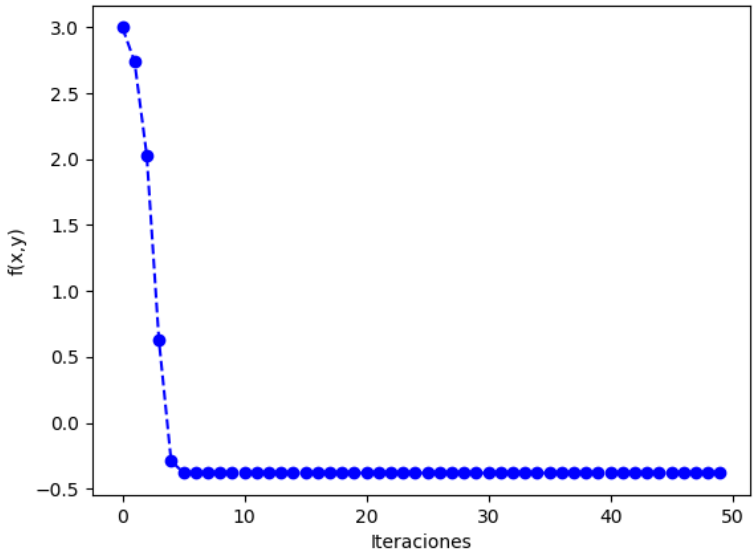

FIGURA 1: Tasa de aprendizaje $\eta = 0.01$.

A continuación repetimos el experimento cambiando el parámetro de tasa de aprendizaje por $\eta = 0.1$, obteniendo:

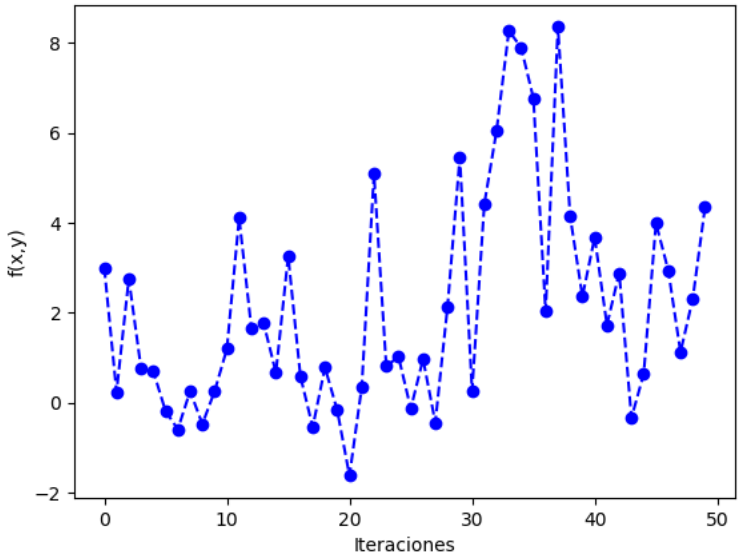

FIGURA 2: Tasa de aprendizaje $\eta = 0.1$.

Si comparamos las dos gráficas vemos que con una tasa de aprendizaje $\eta = 0.01$ la función converge rápidamente en menos de 10 iteraciones, mientras que una tasa de aprendizaje $\eta = 0.1$ ya es un valor muy grande y hace que la función oscile sin llegar a converger. Observamos también que con una tasa de aprendizaje mayor consigue valores más pequeños que el mínimo alcanzado con $\eta = 0.01$, pero sin embargo no se estabiliza.

Es por esto que en este caso una tasa de aprendizaje de $\eta = 0.1$ es demasiado elevada y depende del número de iteraciones que acabe en un punto mayor o menor, mientras que con una tasa de $\eta = 0.01$ el algoritmo converge rápido y alcanza un mínimo, aunque este dependa del punto inicial para que sea el mínimo absoluto.

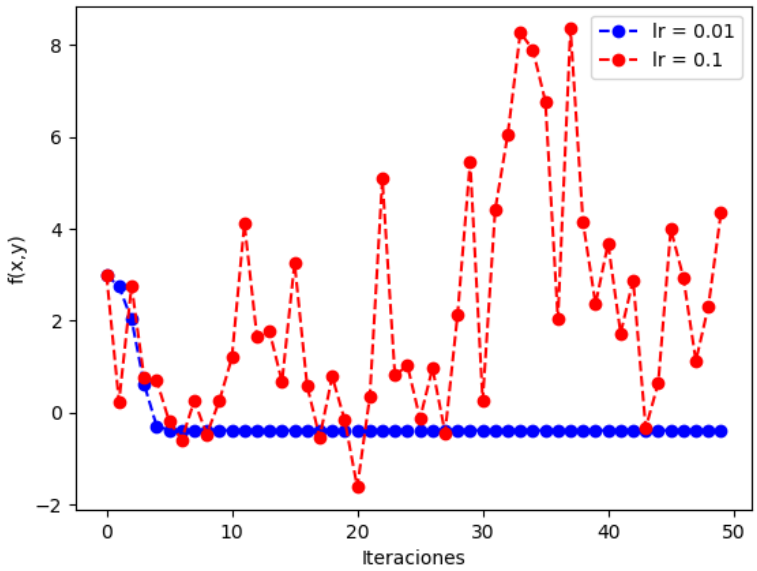

Figura 3: Comparación de las dos gráficas $\eta = 0.01$ y $\eta = 0.1$.

b) Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija en: $(2.1, 2.1)$, $(3, 3)$, $(1.5, 1.5)$, $(1, 1)$. Generar una tabla con los valores obtenidos.

Se ha ejecutado el algoritmo con una tasa de aprendizaje $\eta = 0.01$.

Punto inicio	valor mínimo	(x, y)
$(2.1, -2.1)$	-1.820	$(2.244, -2.238)$
$(3, -3)$	-0.381	$(2.731, -2.713)$
$(1.5, 1.5)$	18.042	$(1.779, 1.031)$
$(1, -1)$	-0.381	$(1.269, -1.287)$

Podemos observar que el valor mínimo que alcanza el algoritmo depende fuertemente del punto inicial, esto es debido a la naturaleza de la función que, como veremos en la siguiente imagen, tiene muchos mínimos locales y como nuestra tasa de aprendizaje es pequeña converge rápidamente al mínimo local más cercano.

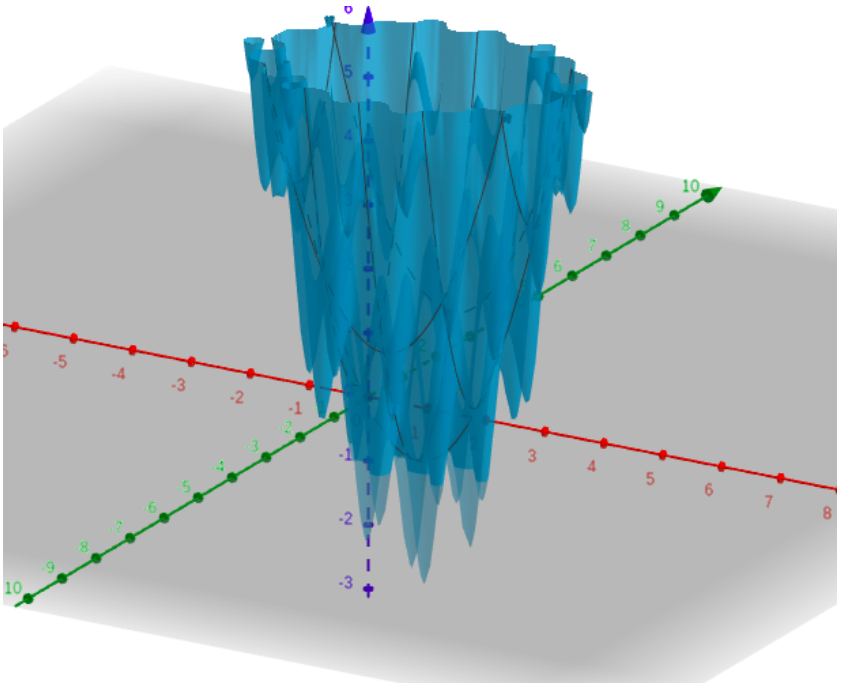

FIGURA 4: Función $f(x, y)$ dibujada con Geogebra.

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

   La verdadera dificultad a la hora de encontrar el mínimo global en una función arbitraria es ajustar los parámetros del algoritmo: la tasa de aprendizaje y el

punto inicial.

Por una parte si escogemos una **tasa de aprendizaje** muy elevada puede ser que el algoritmo converja muy despacio o que oscile y nunca alcance un mínimo, mientras que una tasa demasiado pequeña puede hacer que el algoritmo converja demasiado rápido cayendo en mínimos locales. Esta claro que es difícil dar con el valor exacto de la tasa de aprendizaje para que el algoritmo converja correctamente al mínimo absoluto, una solución a este problema podría ser un valor iterativo de tasa de aprendizaje, comenzando con un valor muy elevado para poder "explorar" al principio la función y disminuyendo el valor de la tasa de aprendizaje iterativamente para ayudar al algoritmo a converger hacía un mínimo.

Por otra parte elegir un **punto inicial** adecuado puede ser clave, ya que como vimos en nuestro experimento si elegiamos un punto muy alejado del mínimo absoluto, al estar este rodeado de mínimos locales el algoritmo converge a uno de estos.

Estas dos tareas se complican cuando no podemos dibujar la función, dejando el ajuste de estos parámetros a la experimentación.

2. Regresión lineal

Este ejercicio se ha desarrollado en el archivo `p1_ej2.py`.

1. Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números usando tanto el algoritmo de la pseudo-inversa 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}.

   Ejecutamos el algoritmo de gradiente descendente estocástico y pintamos los resultados. En primer lugar ejecutamos el algoritmo de gradiente descendente estocástico con parámetros:

   - **x**: vector de características que nos proporciona el ejercicio.

- **y**: vector de etiquetas que nos proporciona el ejercicio.
- **lr**: tasa de aprendizaje $\eta = 0.01$
- **max_iters**: número máximo de iteraciones 20000.
- **tam_minibatch**: tamaño 32.
- [1]**tam**: por defecto esta a 3, cambiará su valor en el último experimento.

Para estudiar la bondad de los modelos utilizamos la función de error cuadrático medio:

$$E_{in}(w) = \frac{1}{N} = \sum_{n=1}^{N}(w^T x_n - y_n)^2$$

La función de error E_{out} se calcula aplicando la misma fórmula pero a los datos de test.

También mostramos la recta $w^T x = 0$ obtenida en el método.

Obtenemos como resultados:

```
Vector de pesos:
[-1.17209772 -0.77060781 -0.48046423]
Bondad del resultado para grad. descendente estocastico:

Ein: 0.07968129651434186
Eout: 0.13221558149479878
```

[1]Parámetro añadido. Este parámetro indica el tamaño del vector w, el cual cambia su tamaño en el último experimento.

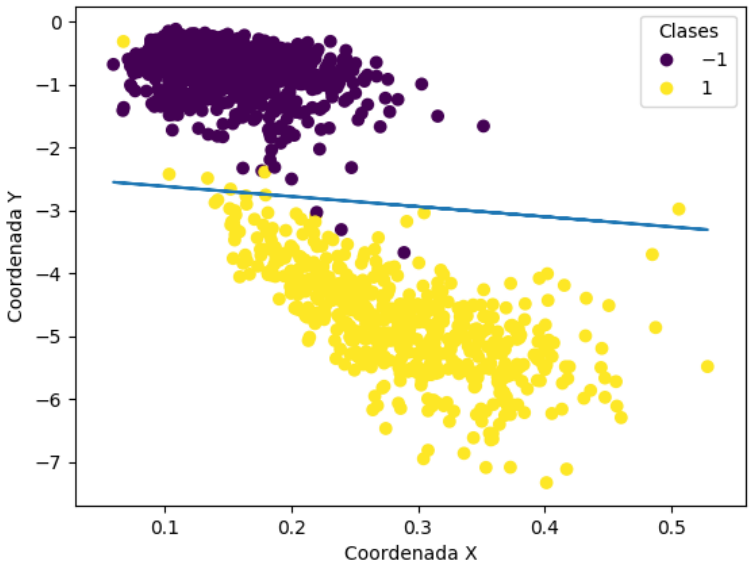

FIGURA 5: Algoritmo de gradiente descendente estocástico.

Para programar el algoritmo de las pseudo-inversa a la hora de calcular el vector w en vez de utilizar su versión analítica $w = X^+y$, donde X es la matrix $X = (x_n^T)_{n=1}^N$ con $x_i = (1, x_{n1}, x_{n2})^T$ vector de características y $X^+ = (X^TX)^{-1}X^T$ llamada *pseudoinversa* de la matriz, se utiliza la descomposición SVD de la matriz X.

Hemos visto en teoría que si X tiene dimensiones $Nx(d+1)$ podemos escribir $X = U\Sigma V^T$, donde U es una matriz ortogonal NxN, Σ una matriz rectangular diagonal $Nx(d+1)$ y V una matriz ortogonal $(d+1)x(d+1)$. Siguiendo un razonamiento teórico llegamos a que $X^+ = V\Sigma U^T$, fórmula que utilizaremos para calcular la pseudo-inversa.

Ejecutamos a continuación el algoritmo de la pseudo-inversa con los datos proporcionado para el ejercicio y obtenemos:

```
Vector de pesos:
[-1.11588016 -1.24859546 -0.49753165]

Bondad del resultado para el algoritmo de la pseudoinversa:

Ein: 0.07918658628900395
```

```
Eout: 0.13095383720052586
```

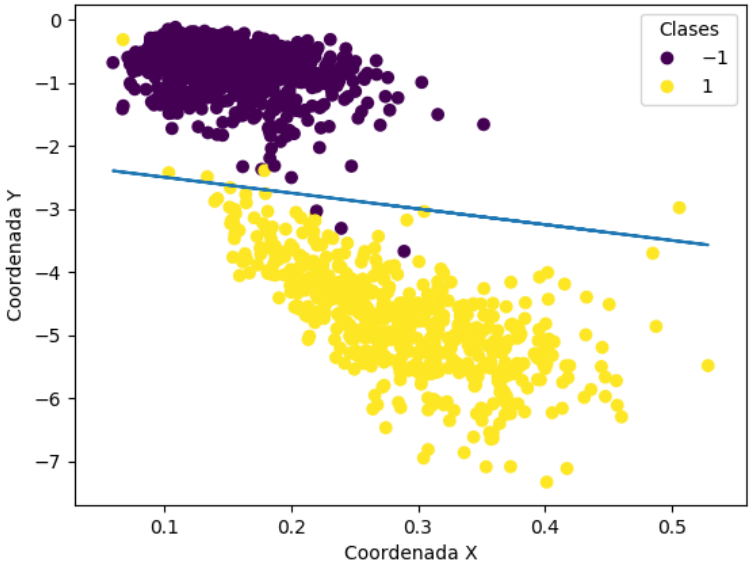

FIGURA 6: Algoritmo de la pseudo-inversa.

Si comparamos los dos métodos vemos que en general los dos obtienen muy buenos resultados ya que el error obtenido es bastante bajo. Consiguiendo el mejor resultado por muy poco el algoritmo de la pseudo-inversa.

Método	E_{in}	E_{out}
SGD	0.0797	0.1322
Pseudo-inversa	0.0792	0.1310

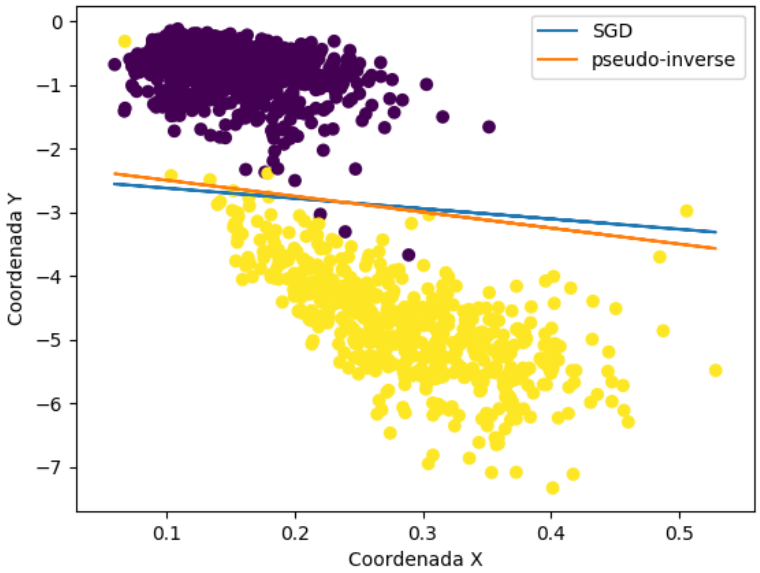

FIGURA 7: Comparación entre gradiente descendente estocastico y pseudo-inversa.

2. En este apartado exploramos como se transforman los errores E_{in} y E_{out} cuando aumentamos la complejidad del modelo lineal usado.

   a) Generar una muestra de entrenamiento de $N = 1000$ puntos en el cuadrado $X = [1, 1][1, 1]$. Pintar el mapa de puntos 2D.

   Para ello se ha definido la función `simula_unif` que es llamada dentro de la función `genera_conjunto`, donde si se le pasa como parámetro el valor *True* se dibuja el mapa de puntos 2D.

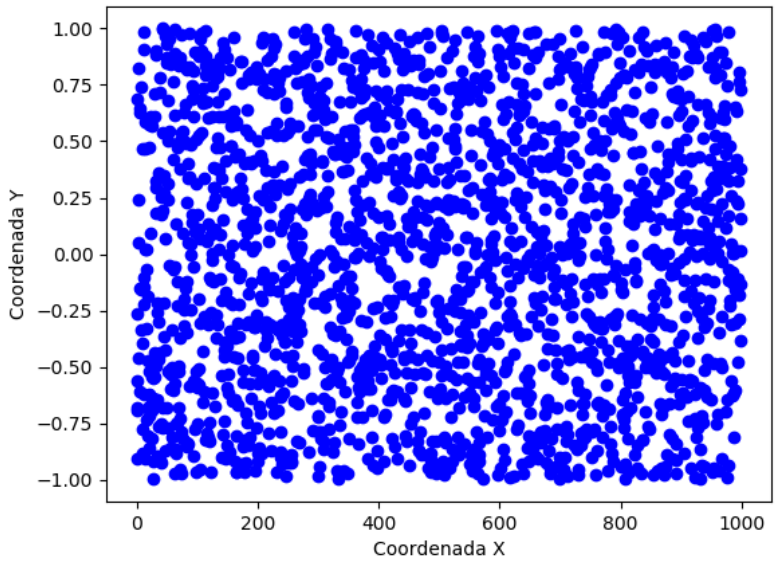

FIGURA 8: Muestra de 1000 puntos distribuidos en el cuadrado $X = [-1, 1]x[-1, 1]$.

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.

Para este apartado se ha definido la función `genera_conjunto`, que además de fabricar las etiquetas con la función que nos han proporcionado y añadiendo un ruido del 10 %, también aprovechamos para añadir un 1 al principio del vector de características x. Lo ejecutamos y los resultados obtenidos son:

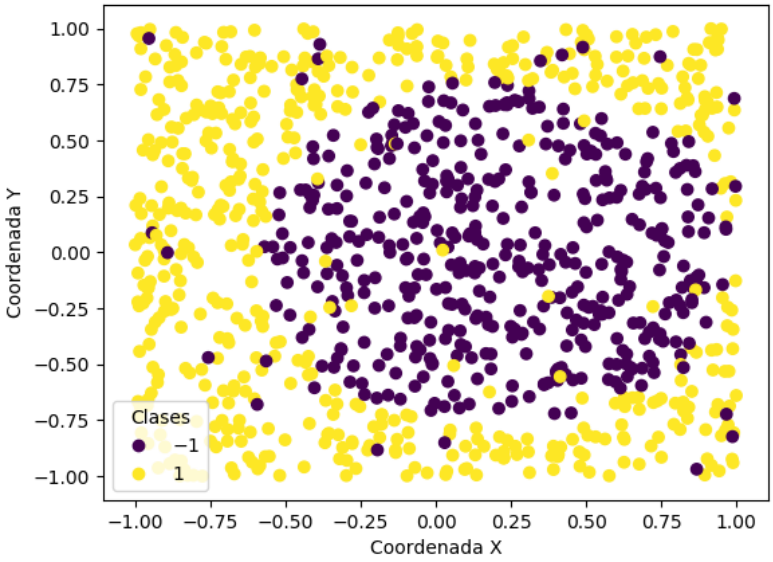

FIGURA 9: Muestra etiquetada.

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).

Como en el apartado anterior ya le añadimos el 1 al vector de características ejecutamos el algoritmo de gradiente descendente estocástico con estos datos y con: $\eta = 0.01$, iteraciones máximas 20000 y tamaño del *minibatch* 32. Obtenemos los siguientes resultados:

```
Muestra N = 1000, cuadrado [-1,1]x[-1,1]
Vector de pesos:
[ 0.03971169 -0.49208204 -0.02785509]
Bondad del resultado para grad. descendente estocastico:

Ein: 0.9156762682380968
```

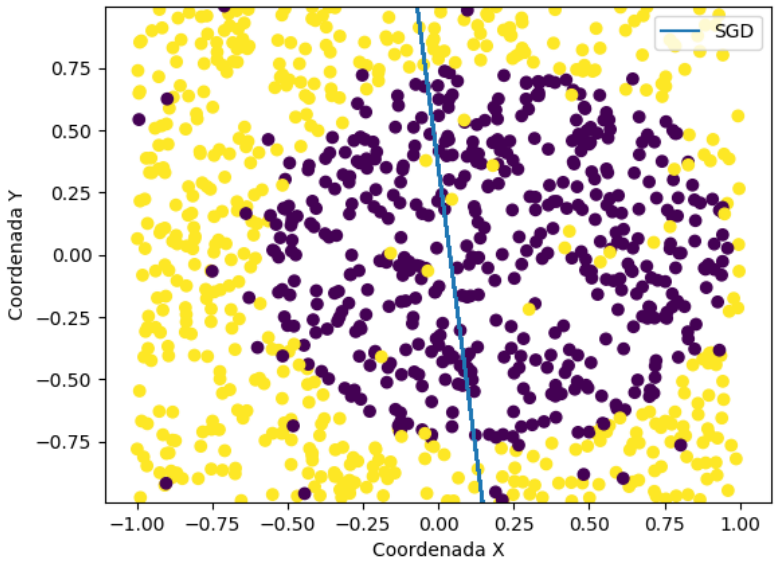

FIGURA 10: Algoritmo de gradiente descendente estocástico.

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.

Repetimos el experimento anterior en un bucle de 1000 iteraciones fabricando de nuevo en cada iteración los conjuntos de datos y al finalizar calculamos los errores medios, obteniendo:

```
Errores Ein y Eout medios tras 1000reps del experimento:

Ein media: 0.909014264842863
Eout media: 0.9149857744864048
```

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}.

Tras ejecutar el experimiento 1000 veces observamos que el error es muy alto, esto no nos sorprende puesto que viendo la distribución del mapa de etiquetas es prácticamente imposible que una recta separe las dos clases.

- Repetir el mismo experimento anterior pero usando características no lineales. Ahora usaremos el siguiente vector de características: $\Phi_2(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$. Ajustar el nuevo modelo de regresión lineal y calcular el nuevo vector de pesos . Calcular los errores promedio de E_{in} y E_{out}.

  Para este apartado se ha definido una función `aniade_caract` cuya función es añadir las características al vector x. Ejecutando de nuevo el experimento con este cambio obtenemos los siguientes resultados:

```
Errores Ein y Eout medios tras 1000reps del experimento con más
   características:

Ein media: 0.5704825441885348
Eout media: 0.5759666841725378
```

- A la vista de los resultados de los errores promedios E_{in} y E_{out} obtenidos en los dos experimentos ¿Que modelo considera que es el más adecuado? Justifique la decisión.

  Los datos hablan por si solos, si añadimos características no lineales conseguimos una regresión que se ajusta mejor consiguiendo disminuir los errores considerablemente. Al añadir términos cuadráticos conseguimos que se asemeje a la función que utilizamos para etiquetar y logramos que los errores disminuyan a costa de aumentar el vector de pesos, aunque el error sigue siendo considerable al haber bastantes puntos de ruido que elevan este valor. En la siguiente gráfica podemos observar como la regresión se ajusta bien al modelo.

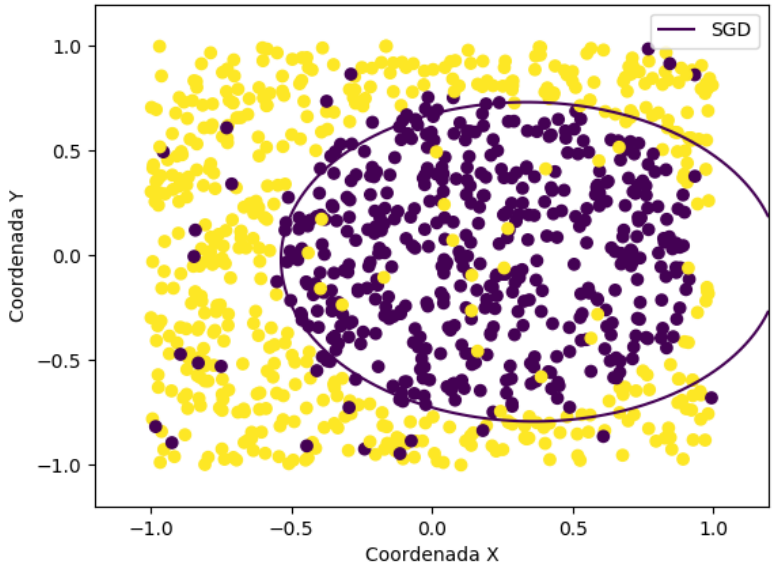

FIGURA 11: Regresión utilizando el método SGD.

3. BONUS

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.

Este ejercicio se ha desarrollado en el archivo `bonus.py`. Para este ejercicio se han implementado dos funciones `newton_grafica` que va guardando los resultados del algoritmo en una gráfica y `newton` que ejecuta el método de Newton. Este algoritmo busca ceros y como lo aplicamos sobre el gradiente podemos encontrar puntos mínimos, máximos y puntos de silla. Esto se aleja de nuestro objetivo que es buscar el mínimo global.

En primer lugar ejecutamos el algoritmo con punto inicial $(1, -1)$, tasa de aprendizaje $\eta = 0.01$ y un máximo de iteraciones de 50. Obteniendo los siguientes resultados.

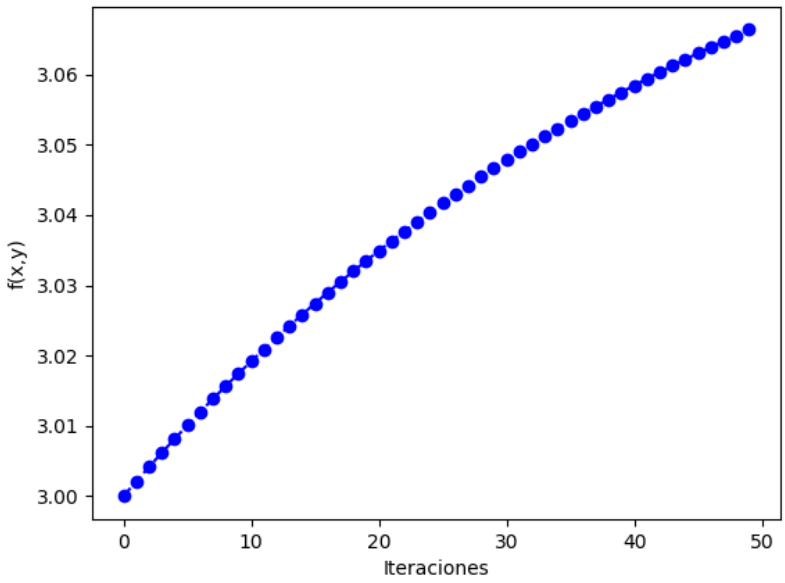

FIGURA 12: Tasa de aprendizaje $\eta = 0.01$

Ejecutamos otra vez con los mismos parámetros cambiando la tasa de aprendizaje a $\eta = 0.1$. Obtenemos los siguientes resultados.

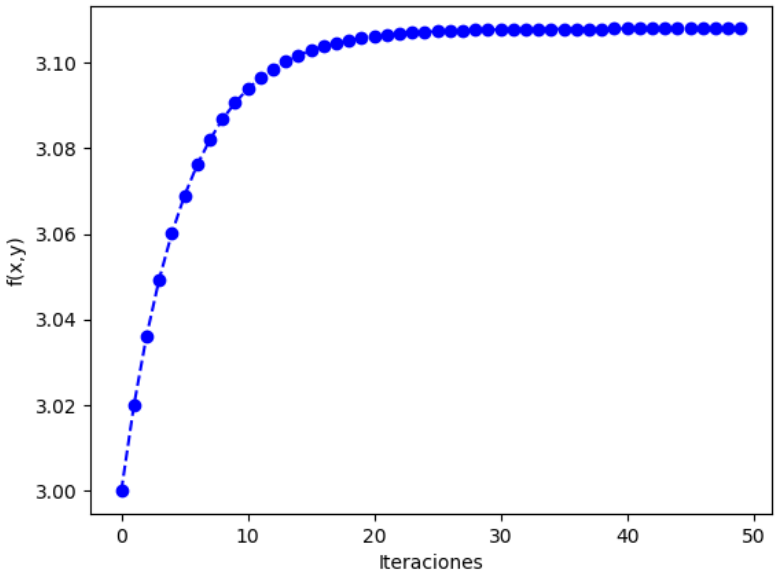

FIGURA 13: Tasa de aprendizaje $\eta = 0.1$

Si comparamos las dos tasas podemos ver que en este caso una tasa $\eta = 0.01$ converge demasiado lento mientras que con $\eta = 0.01$ converge rápidamente al óptimo (máximo o punto de silla).

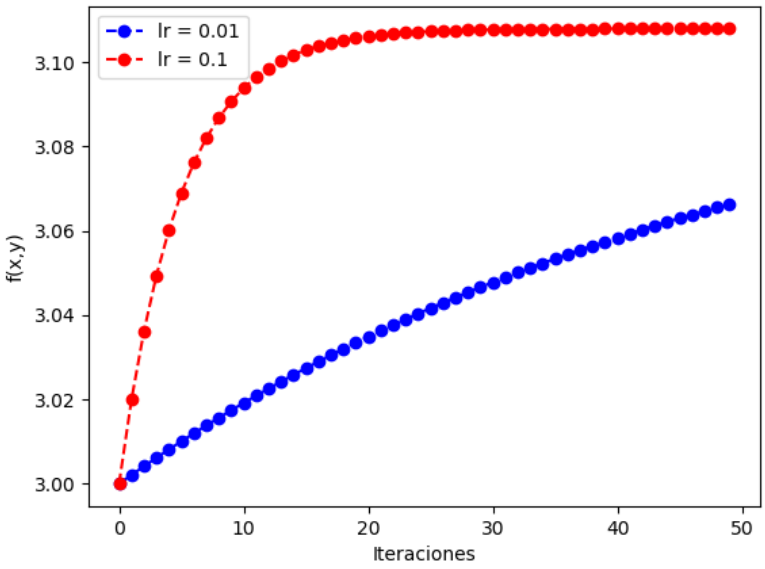

FIGURA 14: Comparación entre las dos tasas de aprendizaje.

Ejecutamos el método con una tasa de aprendizaje $\eta = 0.1$, un máximo de iteraciones de 50 y cambiando el punto inicial. Obtenemos el siguiente resultado:

Punto inicio	valor de f	(x, y)
$(2.1, -2.1)$	-0.000001	$(2.0, -2.0)$
$(3, -3)$	3.108	$(3.054, -3.028)$
$(1.5, 1.5)$	23.690	$(1.425, 1.369)$
$(1, -1)$	3.108	$(0.946, -0.972)$

TABLA 1: Tasa de aprendizaje $\eta = 0.1$.

Ejecutamos el método con una tasa de aprendizaje $\eta = 0.01$, un máximo de iteraciones de 50 y cambiando el punto inicial. Obtenemos el siguiente resultado:

Punto inicio	valor de f	(x, y)
$(2.1, -2.1)$	-0.169	$(2.048, -2.048)$
$(3, -3)$	3.067	$(3.021, -3.011)$
$(1.5, 1.5)$	24.893	$(1.427, 1.508)$
$(1, -1)$	3.067	$(0.979, -0.989)$

Tabla 2: Tasa de aprendizaje $\eta = 0.01$.

Podemos comprobar que cambiando la tasa de aprendizaje cambia un poco los valores obtenidos, esto es debido a que dependiendo de los parámetros el algoritmo alcanzará un mínimo, un máximo o un punto de silla. De hecho para estos puntos el algoritmo no busca el mínimo puesto que no decrece.

Para observar como decrece el algoritmo ejecutamos de nuevo `newton_grafica` con una tasa de aprendizaje $\eta = 1$ y como punto inicial $(2.1, -2.1)$ obteniendo la siguiente gráfica:

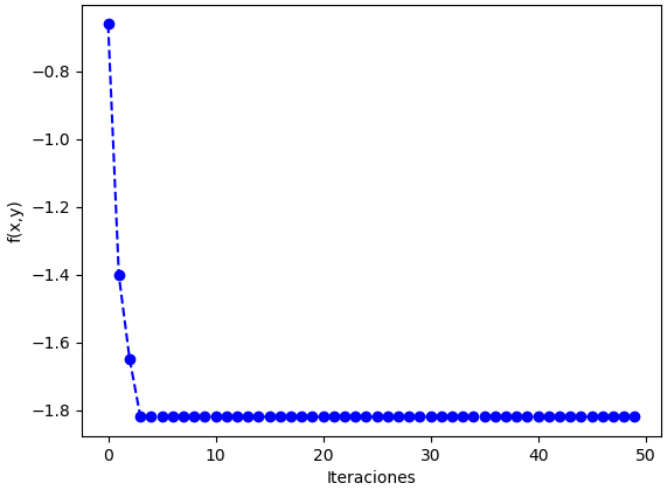

Figura 15: Tasa de aprendizaje $\eta = 1$

Como conclusión, si comparamos los resultados con el algoritmo de gradiente descendente el método de Newton nos da peores resultados, a cambio este algoritmo converge más rápido aunque tiene un mayor coste computacional ya que tiene que calcular una inversa.