UNIVERSIDAD DE GRANADA

Aprendizaje Automático

Práctica 1

Búsqueda iterativa de óptimos y regresión lineal

Autor: Francisco Javier Bolívar Expósito

Curso: 2018-2019

Índice

1. Ejercicio sobre la búsqueda iterativa de óptimos....3
   1. Implementación de Gradiente Descendente....3
   2. Minimizar E(u, v) = $(ue^{v} - 2ve^{-u})^{2}$ con GD. W_0 = [1, 1], learning rate = 0.1....4
      a) Calcular analíticamente y mostrar la expresión del gradiente de la función E(u, v)....4
      b) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de E(u, v) inferior a 10^−14?....5
      c) ¿En qué coordenadas (u, v) se alcanzó por primera vez un valor igual o menor a 10−14 en el apartado anterior?....5
   3. Considerar ahora la función f (x, y) = (x − 2)^2 + 2(y + 2)^2 + 2 sin(2πx) sin(2πy)....5
      a) Usar gradiente descendente para minimizar esta función y ver el efecto del learning rate....5
      b) Obtener el valor mínimo y los valores de las variables (x, y) en distintos puntos de inicio....6
   4. ¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?....6
2. Ejercicio sobre Regresión Lineal....7
   1. Estimar un modelo de regresión lineal a partir de los datos proporcionados....7
   2. Exploración de los errores al aumentar la complejidad....9
   Experimento usando Características Lineales....9
      a) Generar una muestra de entrenamiento de N = 1000 puntos en el cuadrado X = [−1, 1] × [−1, 1]. Pintar el mapa de puntos 2D....9
      b) Asignar etiquetas con la función f(x1, x2) y generar ruido....10
      c) Usando como vector de características (1, x1 , x2) ajustar un modelo de regresion lineal al conjunto de datos generado y estimar los pesos w....11
      d) Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes)....12
      e) Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de Ein y Eout....12
   Experimento con características no lineales....12
   Comparación de experimentos....13
2.1 Bonus....14
   1. Método de Newton....14

1. Ejercicio sobre la búsqueda iterativa de óptimos

1. Implementación de Gradiente Descendente

```
# @brief Algoritmo de Gradiente Descendiente para minimizar una función
# @param f Función (entendido desde el ámbito de la programación) que devuelve
#          la evaluación de la función matemática a minimizar
# @param deriv_f Función (entendido desde el ámbito de la programación) que devuelve
#          la evaluación de la derivada de la función matemática a minimizar
# @param w Incógnitas de la función a minimizar
# @param alpha Tasa de aprendizaje
# @param epsilon Umbral de parada del algoritmo
# @param itermax Cantidad máxima de iteraciones
# @return iterations Cantidad de iteraciones realizadas
# @return w Solución alcanzada
# @return evals Lista del valor de la función en cada iteración
def batchGradientDescent(f, deriv_f, w, alpha, epsilon=None, itermax=500):
    iterations = 0

    evals = []

    evals.append(f(*w))

    # Mientras no se llegue al umbral de parada o al limite de iteraciones
    # actualizamos los pesos
    while (epsilon is None or evals[-1] > epsilon) and iterations < itermax:
        w = w - alpha * deriv_f(*w)
        iterations += 1
        evals.append(f(*w))

    return iterations, w, evals
```

Para poder utilizar el algoritmo de Gradiente Descendente en los apartados siguientes se han introducido varios parámetros y se devuelve mucha información para que la función sea lo más flexible posible.

La función que se quiere minimizar y su derivada se pasan por parámetro para poder usarlo con las funciones que se definen más adelante ‘E(u, v)’ y ‘f(x, y)’. El punto inicial ‘w’ y el learning rate ‘alpha’ también para poder modificar estos parámetros facilmente. Finalmente tenemos dos condiciones de parada diferentes, un umbral de parada opcional útil para encontrar en que iteración se consigo un valor más pequeño al umbral y un número de iteraciones máximas.

Además de la solución obtenida devolvemos el número de iteraciones realizadas y una lista de como ha evolucionado el valor de la función en cada iteración.

2. Minimizar E(u, v) = $(ue^v - 2ve^{-u})^2$ con GD. W_0 = [1, 1], learning rate = 0.1

Simplemente usamos el algoritmo del gradiente descendente con los parámetros indicados:

_, w, evals = batchGradientDescent(function_E, deriv_E, [1, 1], 0.1)

```
En la función E(u,v)
Mínimo encontrado en [0.04473628 0.02395873] tras 19 iteraciones: 0.0
```

Resultados obtenidos

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

Procedimiento seguido para calcular el gradiente de $E(u,v)=(e^v*u-2ve^{(-u)})^2$

Calculamos las derivadas parciales:

$$\frac{\mathrm{d}}{\mathrm{d}u}\left[(\mathrm{e}^v u - 2v\mathrm{e}^{-u})^2\right]$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u}) \cdot \frac{\mathrm{d}}{\mathrm{d}u}[\mathrm{e}^v u - 2v\mathrm{e}^{-u}]$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u}) \left(\mathrm{e}^v \cdot \frac{\mathrm{d}}{\mathrm{d}u}[u] - 2v \cdot \frac{\mathrm{d}}{\mathrm{d}u}[\mathrm{e}^{-u}]\right)$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u}) \left(\mathrm{e}^v \cdot 1 - 2v\mathrm{e}^{-u} \cdot \frac{\mathrm{d}}{\mathrm{d}u}[-u]\right)$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u}) \left(\mathrm{e}^v - 2v\mathrm{e}^{-u}\left(-\frac{\mathrm{d}}{\mathrm{d}u}[u]\right)\right)$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u})\,(\mathrm{e}^v + 2v\mathrm{e}^{-u} \cdot 1)$$
$$= 2\,(\mathrm{e}^v u - 2v\mathrm{e}^{-u})\,(2v\mathrm{e}^{-u} + \mathrm{e}^v)$$

$$\frac{\mathrm{d}}{\mathrm{d}v}\left[(u\mathrm{e}^v - 2\mathrm{e}^{-u} v)^2\right]$$
$$= 2\,(u\mathrm{e}^v - 2\mathrm{e}^{-u} v) \cdot \frac{\mathrm{d}}{\mathrm{d}v}[u\mathrm{e}^v - 2\mathrm{e}^{-u} v]$$
$$= 2\,(u\mathrm{e}^v - 2\mathrm{e}^{-u} v) \left(u \cdot \frac{\mathrm{d}}{\mathrm{d}v}[\mathrm{e}^v] - 2\mathrm{e}^{-u} \cdot \frac{\mathrm{d}}{\mathrm{d}v}[v]\right)$$
$$= 2\,(u\mathrm{e}^v - 2\mathrm{e}^{-u} v)\,(u\mathrm{e}^v - 2\mathrm{e}^{-u} \cdot 1)$$
$$= 2\,(u\mathrm{e}^v - 2\mathrm{e}^{-u})\,(u\mathrm{e}^v - 2\mathrm{e}^{-u} v)$$

Obteniendo como gradiente:

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

```
# Funciones de evaluación de la función matemática E(u, v) = (ue^v − 2ve^−u )^2
def function_E(u, v):
    return (u * np.exp(v) - 2 * v * np.exp(-u))**2

def deriv_E_u(u, v):
    return 2 * (np.exp(v) * u - 2 * v * np.exp(-u)) * (2 * v * np.exp(-u) + np.exp(v))

def deriv_E_v(u, v):
    return 2 * (np.exp(v) * u - 2 * np.exp(-u)) * (u * np.exp(v) - 2 * np.exp(-u) * v)

def deriv_E(u, v):
    return np.array([deriv_E_u(u, v), deriv_E_v(u, v)])
```

Gradiente en Python

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

```
iterations, w , _ = batchGradientDescent(function_E, deriv_E, [1, 1], 0.1, 10**-14)
print(str(iterations) + ' iteraciones hasta obtener un valor inferior a 10^-14')
print('Se alcanza por 1a vez un valor inferior a 10^-14 en las cordenadas (u,v): ' + str(w))
```

```
10 iteraciones hasta obtener un valor inferior a 10^-14
Se alcanza por 1a vez un valor inferior a 10^-14 en las cordenadas (u,v): [0.04473629 0.02395871]
```

Tarda 10 iteraciones usando una tasa de aprendizaje de 0.1

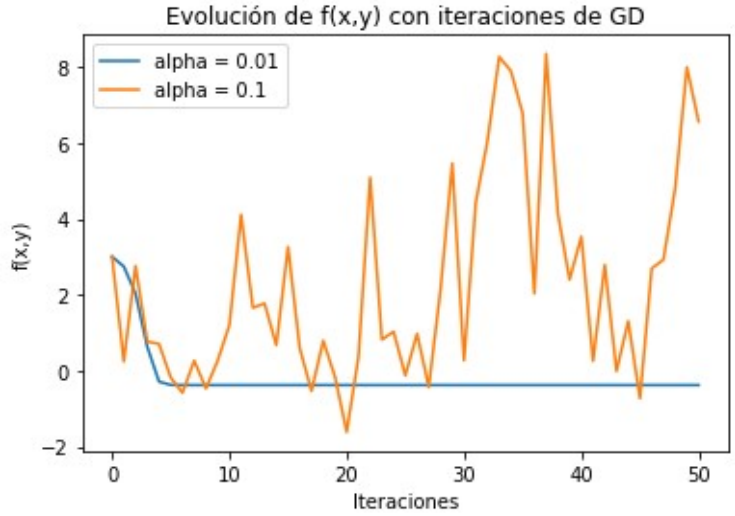
c) ¿En qué coordenadas (u, v) se alcanzó por primera vez un valor igual o menor a 10−14 en el apartado anterior?

Como podemos observar en la imagen anterior se alcanzó por primra vez un valor inferior a 10^-14 en las coordenadas (0.04473, 0.02395)

3. Considerar ahora la función f (x, y) = (x − 2)^2 + 2(y + 2)^2 + 2 sin(2πx) sin(2πy)

a) Usar gradiente descendente para minimizar esta función y ver el efecto del learning rate.

Punto inicial (x0 = 1, y0 = −1), (tasa de aprendizaje η = 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 η = 0,1, comentar las diferencias y su dependencia de η.



Con η = 0.01 se converge a un óptimo local rápidamente. Sin embargo con η = 0.1 observamos como no solo no converge tras 50 iteraciones si no que el valor no siempre desciende si no que sube y baja de forma inconsistente.

El learning rate es un parámetro crítico del gradiente descendente, gracias a la dirección de la derivada en un punto (que es tangente a la función) podemos acercarnos a un mínimo de la función, con el learning rate controlamos cuanto nos movemos en esa dirección.

Si establecemos este valor demasiado alto podemos saltarnos el mínimo de la función, sin llegar a converger, e incluso alejándonos cada vez más del mínimo. Si establecemos este valor demasiado pequeño avanzaremos una distancia mínima en cada iteración.

b) Obtener el valor mínimo y los valores de las variables (x, y) en distintos puntos de inicio.

Punto de inicio: (2,1, −2,1), (3, −3),(1,5, 1,5),(1, −1). Generar una tabla con los valores obtenidos.

Punto de Inicio	(x, y)	Valor mínimo
2.1,-2.1	2.2438049693647883,-2.2379258214861775	-1.820078541547156
3,-3	2.7309356482481055,-2.7132791261667037	-0.38124949743809955
1.5,1.5	1.779119517755436,1.0309456237906103	18.042072349312033
1,-1	1.269064351751895,-1.2867208738332965	-0.3812494974381

Valores obtenidos con learning rate = 0.01

Punto de Inicio	(x, y)	Valor mínimo
2.1,-2.1	0.9045220919767253,-0.4074647698894569	6.892535111615206
3,-3	2.953799052292122,-1.8121552134869434	0.4509479128487024
1.5,1.5	3.6922729420388123,-1.69064279308148	1.3138573544615888
1,-1	1.5211256334166054,-2.926265303006401	1.8269658383715925

Valores obtenidos con learning rate = 0.1

Observamos como varía el valor mínimo obtenido según el punto de inicio escogido y el learning rate.

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

La dificultad de encontrar el mínimo global depende mucho del **espacio de soluciones de la función,** si la función tiene muchos óptimos locales podemos caer en estos y no llegar al mínimo global. En cambio una función convexa con un solo óptimo global presentará un problema mucho más fácil.

Además el gradiente descendente está limitado por la **necesidad de que la función sea derivable.**

Después la dificultad para buscar este mínimo con el algoritmo desarrollado consistiría en eligir los parámetros adecuados:

- **Learning rate:** En el gradiente descendente no escoger un valor adecuado para el ratio de aprendizaje puede hacer que el algoritmo no converja si es demasiado alto o disminuir la velocidad del algoritmo enormemente. Tiene cierta dificultad encontrar el valor adecuado para converger (a un óptimo que puede ser solo local)
- **Punto de inicio:** Como hemos observado en los experimentos prácticas dependiendo del punto inicial podemos llegar a soluciones diferentes. Dependiendo de la cercanía a los distintos óptimos locales o el óptimo local podemos obtener una solución de mayor o menor calidad.

2. Ejercicio sobre Regresión Lineal

1. Estimar un modelo de regresión lineal a partir de los datos proporcionados

Usando tanto el algoritmo de la pseudo-inversa como Gradiente descendente estocástico (SGD). Las etiquetas serán {−1, 1}, unapara cada vector de cada uno de los números. Pintar las soluciones obtenidas junto con losdatos usados en el ajuste. Valorar la bondad del resultado usando Ein y Eout (para Eout calcular las predicciones usando los datos del fichero de test). (usar Regress_Lin(datos, label) como llamada para la función (opcional)).

```
# Funcion para calcular el error
def Err(x,y,w):
    return np.square((np.dot(x, w) - y.reshape(-1, 1))).mean()

# Derivada de la función de error
def deriv_Err(x, y, w):
    return 2 * np.dot(x.T, (np.dot(x, w) - y.reshape(-1, 1))) / len(x)

# Gradiente Descendente Estocastico
def sgd(x, y, alpha, m=64, iterations=300):
    # Inicializamos w a un vector de 0 y calculamos cuantos minibatches
    w = np.zeros((x.shape[1],1))
    n_minibatch = int(len(y)/m)

    # Barajamos las muestras
    index = np.random.permutation(len(y))
    rng_x = x[index]
    rng_y = y[index]

    # Dividimos las muestras en minibatch
    minibatch_x = np.array_split(rng_x, n_minibatch)
    minibatch_y = np.array_split(rng_y, n_minibatch)

    for i in range(iterations):
        index = np.random.permutation(len(minibatch_x))

        # Actualizamos w para cada minibatch-
        for i in index:
            w = w - alpha * deriv_Err(minibatch_x[i], minibatch_y[i], w)

    return w

# Pseudoinversa
def pseudoinverse(x, y):
    return np.dot(np.linalg.pinv(x),y.reshape(-1, 1))
```

Implementación de SGD y Pseudoinversa

Para estimar un modelo de regresión lineal con 2 carácteristicas consideramos la siguiente función.

$$y = w_0 + w_1x_1 + w_2x_2$$

Siendo X las características. Por lo tanto el problema consiste estimar las W de forma que la función se ajuste lo mejor posible a los datos. Para ello lo que queremos es minimizar la función de error, definida como la diferencia cuadrática de las predicciones y la clasificación real.

Para minimizar hemos implementado el algoritmo del Gradiente Descendente Estocástico y el de la Pseudoinversa.

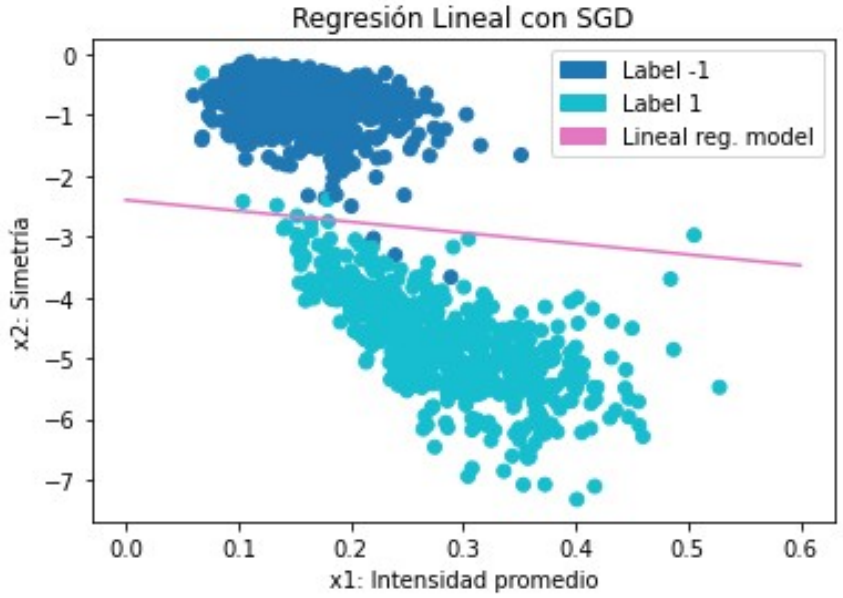
Para el Gradiente Descendente Estocástico actualizando W utilizando solo un subconjunto de las muestras cada vez, estos subconjuntos (minibatches) se han elegido dividiendo las muestras originales de forma aleatoria. Esta variación es usada con más frecuencia por su mayor velocidad de computación.

En todas las ejecuciones se ha elegido como punto inicial todas las componentes de W a 0, un learning rate de 0.035, un tamaño de minibatch de 64 y 100 iteraciones.

Para generar las gráficas de todo el ejercicio vamos a usar representaciones en 2D ya que permiten observar fácilmente como de bien se ajustan los modelos de regresión a los datos. Los datos se pintaran como puntos cambiando su color según su etiqueta y los modelos como líneas con suficientes puntos para obtener una buena representación (en este caso solo usaremos solo dos puntos para generar la línea ya que es una recta).

Al inicio del script se elige un ‘colormap’ aleatorio para decidir de que color serán los puntos según su etiqueta y de que color será el modelo de regresión lineal. La elección se hace solo de un conjunto reducido de los colormaps disponibles para asegurarse de que los colores elegidos son suficientemente diferenciables con una paleta de colores adecuada. Se ha decidido asignar los colores de esta forma simplemente para obtener gráficos estéticamente placenteros y que transmitan la información de forma efectiva.

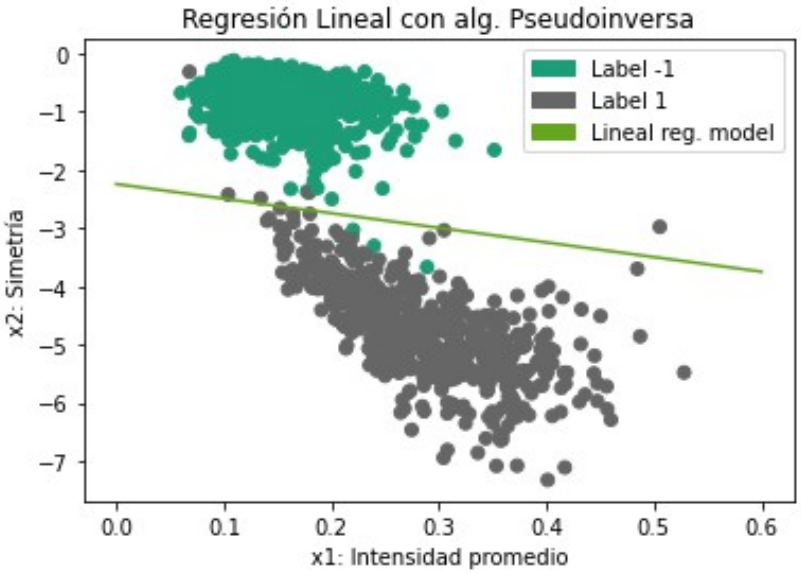
Vamos a ver los resultados obtenidos con los dos algoritmos.



Bondad del resultado para grad. descendente estocástico:

Ein: 0.07950090657635646. Eout: 0.1317841583923955

Los datos se ajustan bien a un modelo con características lineales, aunque algunas muestras quedan fuera del modelo, el modelo de regresión lineal obtenido con SGD clasifica las muestras con un error muy reducido.



Bondad del resultado para alg. de la PseudoInversa:

Ein: 0.07918658628900395. Eout: 0.13095383720052578

La Pseudoinversa consigue un modelo de regresión lineal con un error ligeramente menor.

2. Exploración de los errores al aumentar la complejidad

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] × [−size, size]

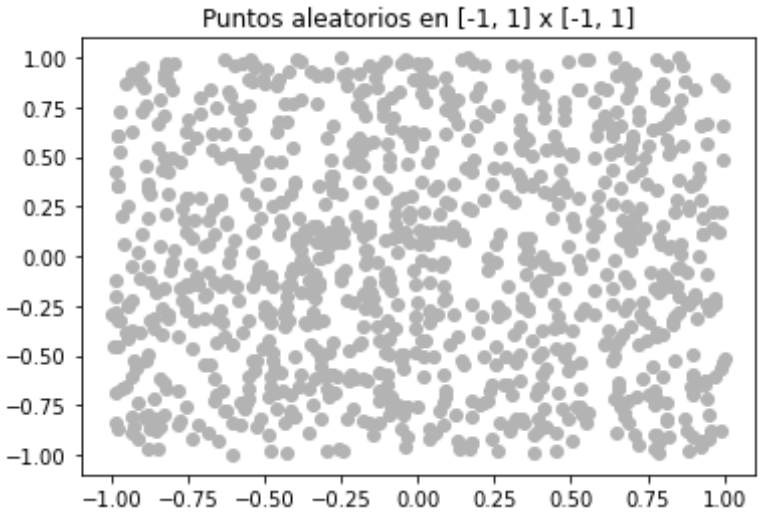
Experimento usando Características Lineales

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

Haciendo uso de la función:

```
def simula_unif(N, d, size):
    return np.random.uniform(-size,size,(N,d))
```

Obtenemos la siguiente muestra de entrenamiento en una ejecución:

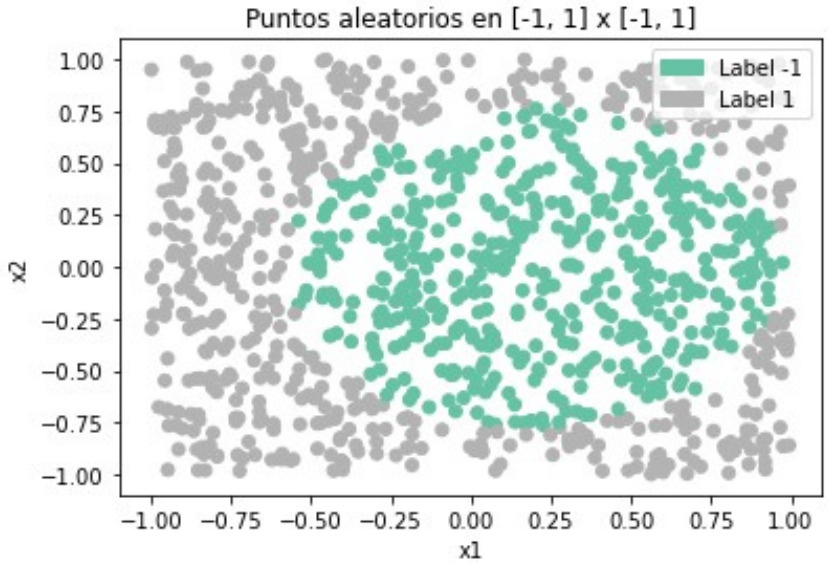


b) Asignar etiquetas con la función f(x1, x2) y generar ruido

Consideremos la función f (x1 , x2) = sign((x1 − 0,2)2 + x22 − 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.

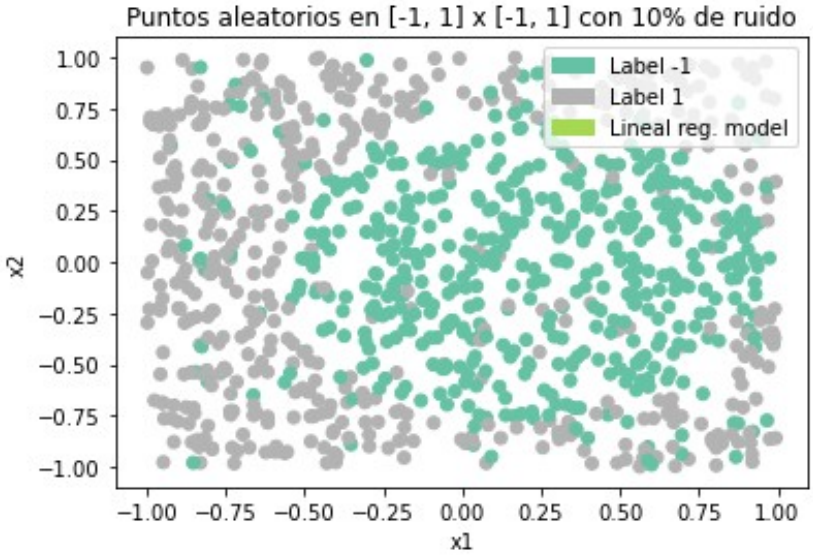
```
# Función para asignar etiquetas
def assign_labels(x1, x2):
    return np.sign((x1 - 0.2) ** 2 + x2 ** 2 - 0.6)
```

Al asignar las etiquetas según la función indicada obtenemos la siguiente clasificación:



```
# Asignamos las etiquetas con la función f y generamos un 10% de ruido
    y = assign_labels(x[:, 1], x[:, 2])
    index = np.random.choice(np.arange(len(y)), int(len(y)*0.1), replace=False)
    y[index] = -y[index]
```

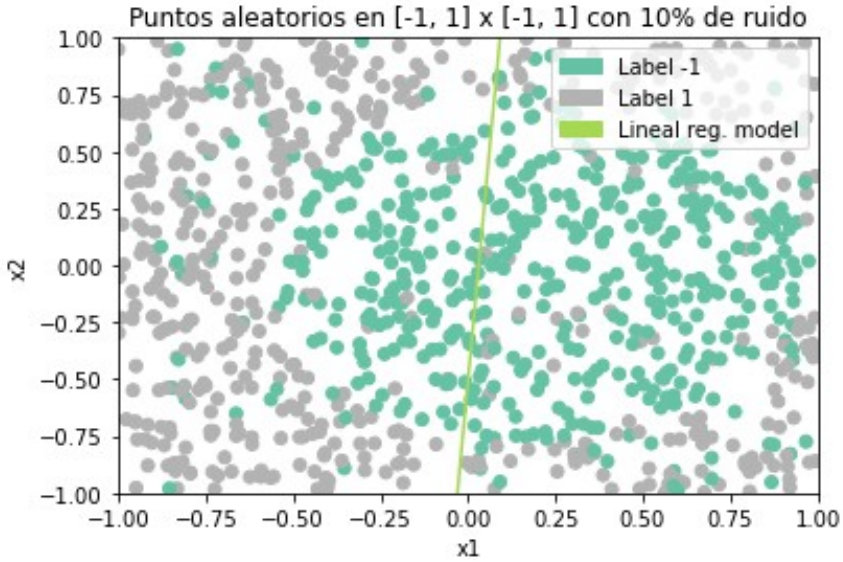
Generamos ruido cambiado el signo de un 10% de las etiquetas, seleccionándolas aleatoriamente.



c) Usando como vector de características (1, x1 , x2) ajustar un modelo de regresion lineal al conjunto de datos generado y estimar los pesos w.

Estimar el error de ajuste Ein usando Gradiente Descendente Estocástico (SGD).

Hemos obtenido el siguiente ajuste con el Gradiente Descendente Estocástico.



Bondad del resultado para SGD con un modelo de características lineales:

Ein: 0.891563172084386

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

- Calcular el valor medio de los errores Ein de las 1000 muestras.
- Generar 1000 puntos nuevos por cada iteración y calcular con ellos el valor de Eout en dicha iteración. Calcular el valor medio de Eout en todas las iteraciones-

Bondad media de los resultados tras 1000 repeticiones

Ein: 0.9272442709163895. Eout: 0.8973616679606627

e) Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de Ein y Eout

Tenemos unos errores cercanos al 90%, no es un buen ajuste lineal. Los datos claramente no se pueden ajustar correctamente a la función lineal tan simple que se está utilizando.

Además el error de salida es menor al de entrada, posiblemente por el 10% de ruido introducido en la muestra de entrenamiento que hace todavía más difícil ajustar un modelo lineal. Aun así es reseñable que el modelo generado tenga mejor ajuste a los datos reales sin ruido.

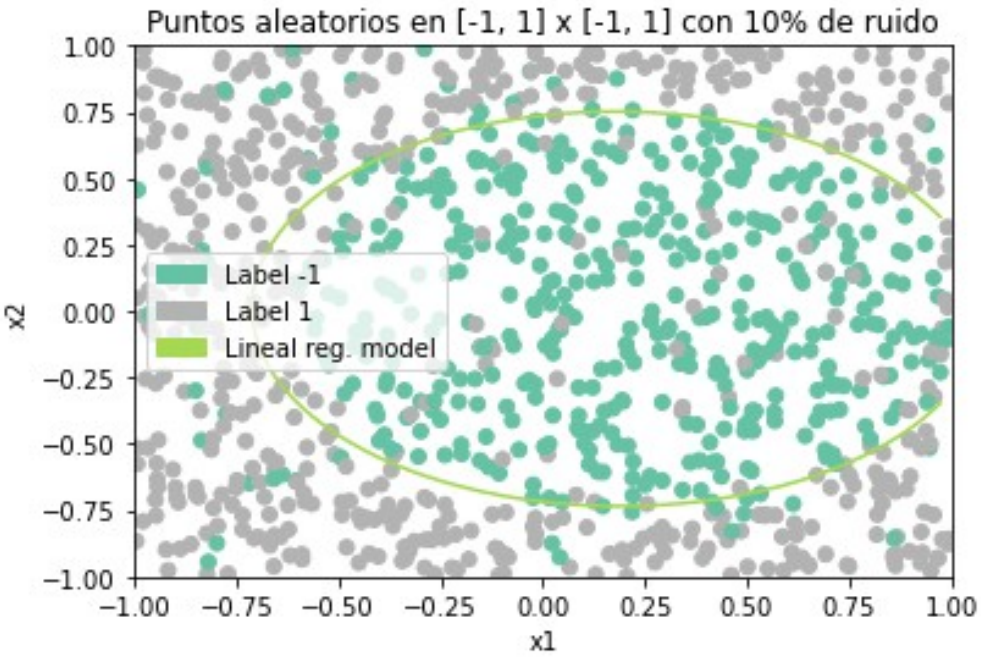
Experimento con características no lineales

Repetir el mismo experimento anterior pero usando características no lineales. Ahora usaremos el siguiente vector de características: Φ2 (x) = (1, x1 , x2, x1 x2 , x1^2 , x2^2). Ajustar el nuevo modelo de regresión lineal y calcular el nuevo vector de pesos ŵ. Calcular los errores promedio de Ein y Eout.

```
# Generación de las muestras
x = np.ones((1000, 6))
x[:,[1, 2]] = simula_unif(1000, 2, 1)
# Calculamos las características 3-5
x[:, 3] = x[:, 1] * x[:, 2]
x[:, 4] = x[:, 1] * x[:, 1]
x[:, 5] = x[:, 2] * x[:, 2]
```

Tras generar las muestras y calcular las características que dependen de x1 y x2 ya podemos estimar nuestro modelo de regresión usando SGD.

A continuación se muestra los resultados obtenidos en una ejecución concreta del experimento



Bondad media de los resultados

Ein: 0.5464087242766766

Repetimos el experimento 1000 veces generando también muestras de test para comprobar el Eout del modelo estimado.

Bondad media de los resultados usando un modelo de características no lineales tras 1000 repeticiones

Ein: 0.5803077415148908

Eout: 0.37771281939014223

Comparación de experimentos

El modelo más adecuado es el de características no lineales, donde se obtiene un Ein y un Eout mucho menor, siendo un modelo que predice mejor la clasificación de una muestra.

El modelo con características no lineales se ajusta mejor a la forma real de los datos y por lo tanto es una mejora respecto al otro modelo considerado que no se ajusta prácticamente nada.

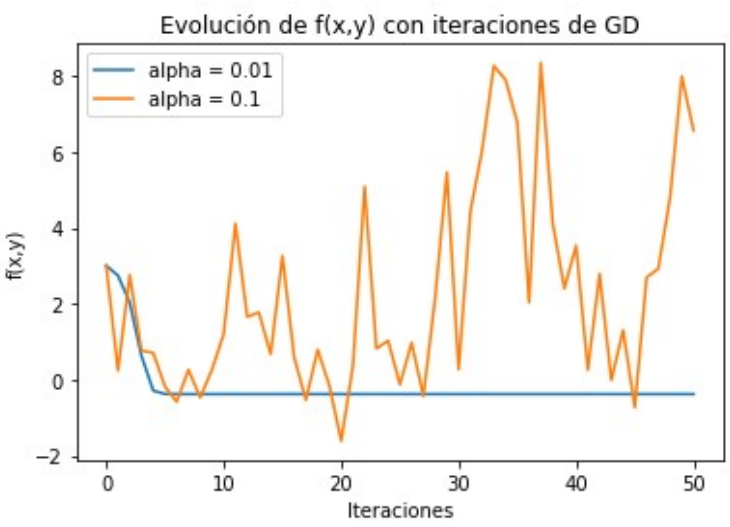
2.1 Bonus

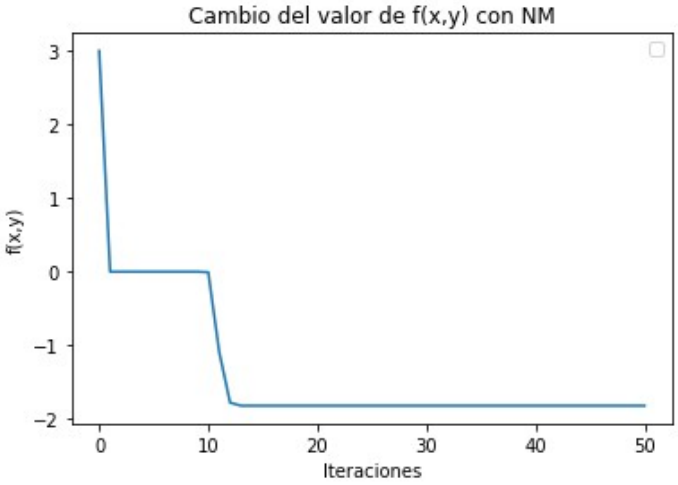
1. Método de Newton

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.

El método de Newton se utiliza para encontrar los ceros o raíces de una función teniendo su derivada. Para poder usarlo como algoritmo de minimización necesitaremos también la 2a derivada, buscando así raíces de la 1a derivada y por lo tanto obteniendo mínimos o máximos de la función.

```
# Funciones de evaluación de la función matemática f(x, y) = (x − 2)^2 + 2(y + 2)^2 + 2 sin(2πx) sin(2πy)
def function_f(x, y):
    return (x - 2)**2 + 2 * (y + 2)**2 + 2 * np.sin(2 * np.pi * x) * np.sin(2 * np.pi * y)

def deriv_f_x(x, y):
    return 4*np.pi*np.sin(2*np.pi*y)*np.cos(2*np.pi*x) + 2*(x - 2)

def second_deriv_f_x(x, y):
    return 2 - 8*np.pi**2*np.sin(2*np.pi*y)*np.sin(2*np.pi*x)

def deriv_f_y(x, y):
    return 4*(np.pi*np.sin(2*np.pi*x)*np.cos(2*np.pi*y) + y + 2)

def second_deriv_f_y(x, y):
    return 4 - 8*np.pi**2*np.sin(2*np.pi*x)*np.sin(2*np.pi*y)

def deriv_f(x, y):
    return np.array([deriv_f_x(x, y), deriv_f_y(x, y)])

def second_deriv_f(x, y):
    return np.array([second_deriv_f_x(x,y), second_deriv_f_y(x,y)])


# @brief Algoritmo del método de Newton para encontrar mínimo/máximo de una función
def newtonMethod(f, deriv_f, deriv2_f, w, iterations=100):
    evals = []

    evals.append(f(*w))

    for i in range(iterations):
        w = w - deriv_f(*w) / deriv2_f(*w)
        evals.append(f(*w))

    return w, evals
```





Empezando los dos algoritmos en el punto [1, -1] podemos observar que el algoritmo de Newton llega a un valor mejor sin tener que usar muchas más iteraciones que el Batch Gradient Descent y no se queda bloqueado en el óptimo local cercano al cero. Sin embargo este algoritmo tiene también sus desventajas como observaremos en la siguiente comparación.

Punto de Inicio	(x, y)	Valor mínimo
2.1,-2.1	2.2438049693647883,-2.2379258214861775	-1.820078541547156
3,-3	2.7309356482481055,-2.7132791261667037	-0.38124949743809955
1.5,1.5	1.779119517755436,1.0309456237906103	18.042072349312033
1,-1	1.269064351751895,-1.2867208738332965	-0.3812494974381

Valores obtenidos con Batch GD con learning rate = 0.01

Punto de Inicio	(x, y)	Valor mínimo
[2.1, -2.1]	[2.24380497 -2.23792582]	-1.8200785415471563
[3, -3]	[3.28284774 -1.73633623]	3.7350805793644866
[1.5, 1.5]	[13.89301077 -2.23522526]	142.79447395290157
[1, -1]	[1.75619503 -1.76207418]	-1.8200785415471565

Valores obtenidos con Newton's Method

Obtenemos valoremos iguales o mejores para los puntos [2.1, -2.1] y [1, -1], pero mucho peores para el resto. ¿Que está pasando? El método de Newton busca máximos o mínimos de la función, entonces según si el punto de inicio se encuentra en una parte cóncava o convexa de la función podemos estar maximizando (y por lo tanto en nuestro caso obteniendo peores valores) con cada iteración.

Una posible solución a este problema es controlar si el valor de la función está aumentando en cada iteración, y si es así utilizar el gradiente descendente que siempre intenta conseguir un mínimo o buscar otros puntos de inicio que proporcionen una solución con más calidad.