UNIVERSIDAD DE GRANADA

PRÁCTICA 1: Búsqueda Iterativa de Óptimos y Regresión Lineal

Aprendizaje Automático

Alumno: José María Sánchez Guerrero **Grupo:** A3

CONTENIDO

EJERCICIO SOBRE LA BÚSQUEDA ITERATIVA DE ÓPTIMOS

Gradiente Descendente

Ejercicio 1

Este algoritmo sirve para minimizar funciones el cual empieza en un punto y va descendiendo por la pendiente más pronunciada. Está determinado por la siguiente fórmula:

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

En nuestra implementación vamos a pasarle los siguientes parámetros correspondientes a la fórmula anterior:

- **initial_point** (w_j). Punto inicial y salida de la función (mínimo local).
- **eta** (η). Tasa de aprendizaje.
- **func**. Será la función derivable a minimizar.

La implementación es la siguiente:

```python
####### EJERCICIO 1 #######

# Implementación del gradiente descendente
def gradient_descent(initial_point, eta, maxIter, error2get, func):
    w = np.copy(initial_point)  # Copiamos en w el punto inicial
    iterations = 0              # Creamos una variable para las iteraciones

    # Comienzo del bucle
    while True:
        # Dependiendo del parámetro hace el gradiente de:
        #   E(u,v) -> Ejercicio 2
        #   f(x,y) -> Ejercicio 3
        if func=='E':
            gradiente = gradE(*w)
        else:
            gradiente = gradF(*w)

        w = w - eta * gradiente     # Cuerpo de la función del gradiente descendente
        iterations += 1             # Aumentamos en 1 las iteraciones

        # Dependiendo del parámetro termina el while de la siguiente forma:
        #   E -> Ejercicio 2, apartado b) -> cuando el error llegue a 10^-14
        #   f -> Ejercicio 3 -> solo paran cuando lleguen al numero de iteraciones máximo
        if func == 'E':
            if E(*w) < error2get or iterations >= maxIter:
                break
        else:
            if iterations >= maxIter:
                break
    return w, iterations
```

Ilustración 1. Implementación del algoritmo gradiente descendente.

Ejercicio 2

Apartado a). En la implementación mostrada anteriormente, tanto el gradiente de la función o **gradE(*w)** como la propia función E y sus derivadas, se calculan de la siguiente forma (**gradF(*w)** será para los ejercicios que vienen a continuación):

```
####### EJERCICIO 2 #######

### Apartado a) ###
def E(u, v):
    return ((u ** 2) * (np.exp(v)) - 2 * (v ** 2) * (np.exp(-u))) ** 2

# Derivada parcial de E con respecto a u
def dEu(u, v):
    return 4 * np.exp(-2 * u) * ((u ** 2) * (np.exp(u + v)) - 2 * (v ** 2)) * (u * np.exp(u + v) + v ** 2)

# Derivada parcial de E con respecto a v
def dEv(u, v):
    return 2 * np.exp(-2 * u) * ((u ** 2) * (np.exp(u + v)) - 4 * v) * ((u ** 2) * np.exp(u + v) - 2 * (v ** 2))

# Gradiente de E
def gradE(u, v):
    return np.array([dEu(u, v), dEv(u, v)])
```

Ilustración 2. Función E, sus derivadas y su gradiente.

Apartados b) y c). Para estos apartados he inicializado las variables a los valores correspondientes, posteriormente realizo el gradiente descendente y muestro los resultados por pantalla:

```
# Inicializamos las variables a los valores que nos piden
eta = 0.01          # Tasa de aprendizaje
maxIter = 10000     # Número máximo de iteraciones
error2get = 1e-14   # Valor del error
initial_point = np.array([1.0, 1.0])    # Punto donde comienza la función

# Llamada a la función con los parámetros anteriores y el argumento E
w, it = gradient_descent(initial_point, eta, maxIter, error2get, 'E')

print('\nEjercicio 2')
### Apartado b) ###
print('Apartado b) Numero de iteraciones: ', it)
### Apartado c) ###
print('Apartado c) Coordenadas obtenidas: (', w[0], ', ', w[1], ')')
```

Ilustración 3. Inicialización de variables y ejecución de la función

Los resultados obtenidos son:

- Número de iteraciones: **33**
- Coordenadas obtenidas: **(0.619207678450638 , 0.9684482690100485)**

Si probamos a cambiar el valor de 'eta' podemos ver cómo tanto el número de iteraciones como las coordenadas obtenidas pueden cambiar.

Por ejemplo, si probamos con **0.001 (menor tasa de aprendizaje)** tarda más iteraciones en llegar, y el resultado es muy similar.

- Número de iteraciones: **379**
- Coordenadas obtenidas: **(0.626692227091052 , 0.9989466146573935)**

Por otra parte, si probamos con **0.1 (mayor tasa de aprendizaje)**, se supone que tardará menos iteraciones, pero si ejecutamos vemos que nunca llega a converger:

- Numero de iteraciones: **10000**
- Coordenadas obtenidas: **(11.25754682177815 , -24.324009350654297)**

Tenemos que tener mucho cuidado y elegir bien nuestra tasa de aprendizaje, pues como acabamos de ver, podemos llegar a la solución más rápido con una tasa de aprendizaje mayor, pero también corremos el riesgo de que nuestra función nunca converja.

Adicionalmente, el profesor nos proporcionó un trozo de código para mostrar el gráfico resultante de calcular el gradiente descendente, y este es el resultado:

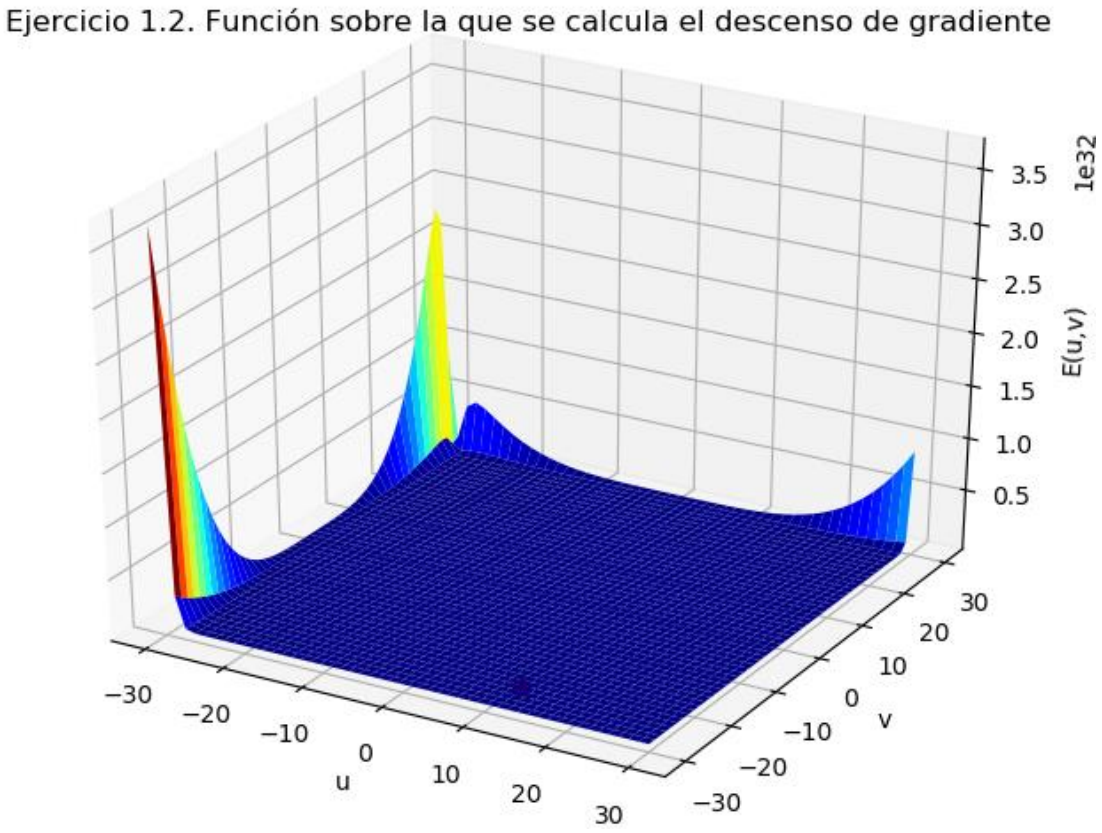

Ilustración 4. Función sobre la que se calcula el descenso de gradiente.

Ejercicio 3

Como hemos comentado antes en el ejercicio 2, vamos a utilizar ahora **gradF(*w)**. Ésta es su implementación y la de sus derivadas:

```
####### EJERCICIO 3 #######
def F(x, y):
    return (x**2) + 2*(y**2) + 2*np.sin(2*np.pi*x)*np.sin(2*np.pi*y)

# Derivada parcial de F con respecto a x
def dFu(x, y):
    return 2 * (2*np.pi * np.cos(2*np.pi*x) * np.sin(2*np.pi*y) + x)

# Derivada parcial de F con respecto a y
def dFv(x, y):
    return 4 * (np.pi * np.sin(2*np.pi*x) * np.cos(2*np.pi*y) + y)

# Gradiente de F
def gradF(x, y):
    return np.array([dFu(x, y), dFv(x, y)])
```

Ilustración 5. Función F, sus derivadas y su gradiente

Apartado a). En este apartado tenemos que mostrar los cambios que se producen al hacer el descenso en F cuando variamos el valor de la tasa de aprendizaje. Utilizaremos un máximo de 50 iteraciones, un punto inicial de (x = 0,1; y = 0,1) y los valores del 'eta' serán 0.1 y 0.01.

```
### Apartado a) ###
# Asignamos los nuevos valores que nos piden
eta = 0.01          # Tasa de aprendizaje
maxIter = 50        # Número máximo de iteraciones
initial_point = np.array([0.1, 0.1])    # Punto donde comienza la función

# Llamada a la función con los parámetros anteriores y con el argumento F
w, it = gradient_descent(initial_point, eta, maxIter, 0, 'F')

print('Ejercicio 3')
print('Numero de iteraciones con eta = 0.01: ', it)
print('Coordenadas obtenidas con eta = 0.01: (', w[0], ', ', w[1], ')')

# Ahora cambiamos el eta y repetimos el experimento
eta = 0.1       # Tasa de aprendizaje
w, it = gradient_descent(initial_point, eta, maxIter, 0, 'F')
print('Numero de iteraciones con eta = 0.1: ', it)
print('Coordenadas obtenidas con eta = 0.1: (', w[0], ', ', w[1], ')')
```

Ilustración 6. Inicialización de variables y ejecución de la función

Los resultados obtenidos son:

- Numero de iteraciones con eta = 0.01: **50**
- Coordenadas con eta = 0.01: **(0.24380496936478, -0.23792582148617)**
- Numero de iteraciones con eta = 0.1: **50**
- Coordenadas con eta = 0.1: **(0.10039167365942, -1.0157510051441)**

Podemos ver cómo con la tasa de aprendizaje más alta llegamos a un mejor mínimo, pero como habíamos dicho anteriormente, con un mayor valor corremos el riesgo de que la función no converja.

A continuación, un gráfico del mínimo en la función:

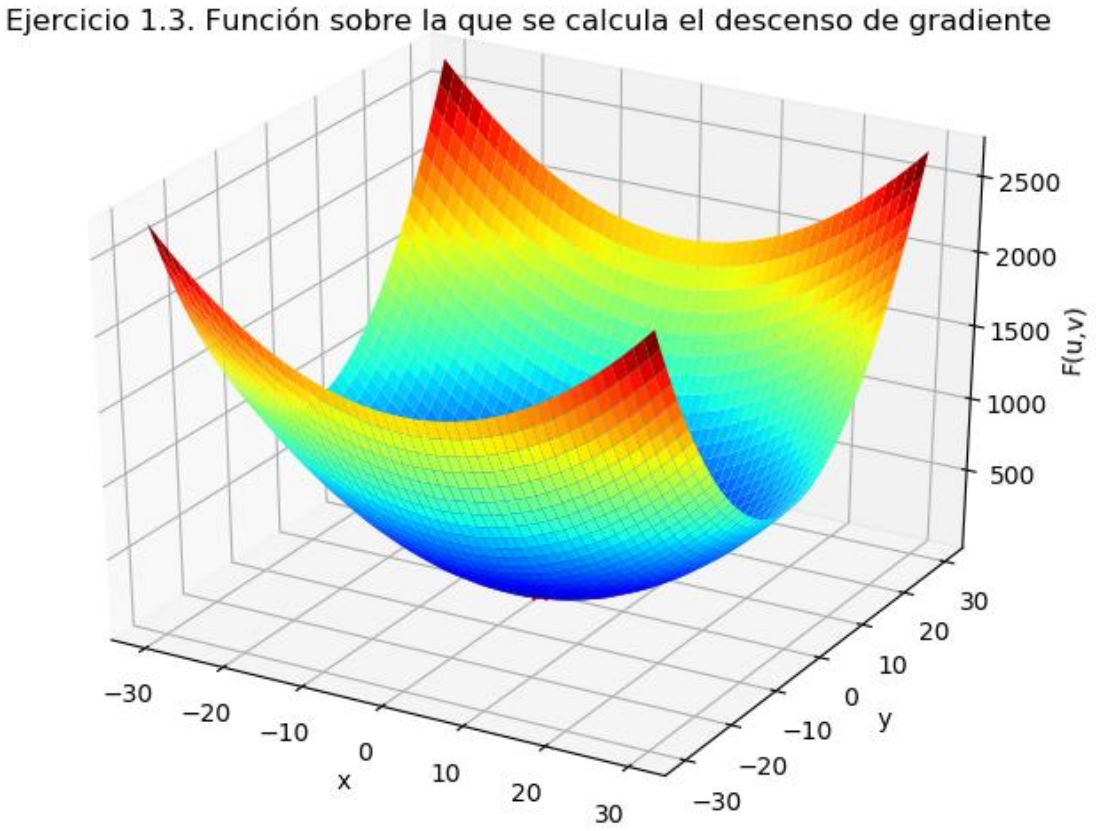

Ilustración 7. Mínimo de la función F

Apartado b). Ahora tenemos que comprobar los distintos valores (x,y) en donde se alcanza el mínimo con distintos puntos iniciales. Vamos a mostrarlo en la siguiente tabla:

P. inicial	**Valor de x**	**Valor de y**	**Valor mínimo**
(0.1, 0.1)	0.24380496936478835	-0.23792582148617766	-1.8200785415471563
(1.0, 1.0)	1.2180703013110816	0.7128119506017776	0.5932693743258357
(-0.5, -0.5)	-0.7313774604138037	-0.23785536290157222	-1.3324810623309777
(-1.0, -1.0)	-1.2180703013110816	-0.7128119506017776	0.5932693743258357

Ejercicio 4

Mi conclusión es que lo más importante para encontrar el mínimo global en una función es el **punto inicial**, ya que los mínimos locales pueden ser muy diferentes dependiendo del punto inicial.

Por ejemplo, eligiendo un punto inicial de (1, 1) en una función X convergemos al mínimo global y con un punto inicial de (0.9, 0.9) podemos converger a un mínimo local con un valor mucho mayor que el global.

Aun así, no podemos descuidar la **tasa de aprendizaje**, ya que una muy grande puede hacer que no converjamos y una muy pequeña que tardemos muchas iteraciones.

EJERCICIO SOBRE REGRESIÓN LINEAL

Gradiente Descendente Estocástico (SGD)

Ejercicio 1

Para estimar un modelo de regresión lineal vamos a utilizar dos algoritmos: el de la **pseudoinversa** y el **gradiente descendente estocástico**. Estas son sus implementaciones:

```python
# Gradiente Descendente Estocastico
def sgd(x, y, initial_point, eta, maxIter):
    w = np.copy(initial_point)  # Copiamos en w el punto inicial
    iterations = 1

    # Obtenemos un vector aleatorio de indices de tamaño 64
    index = np.random.choice(y.size, size=64, replace=False)
    # Asignamos los valores de los indices de 'x' e 'y' en los minibatches
    minibatch_x = x[index,:]
    minibatch_y = y[index]

    # Calculamos el gradiente descendente
    while iterations < maxIter:
        w = w - eta * dErr(minibatch_x, minibatch_y, w)
        iterations += 1

    return w


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

Ilustración 8. Implementación del SGD y de la pseudoinversa.

La fórmula del gradiente descendente estocástico (SGD) es como la del gradiente descendente, pero en este caso no vamos a coger toda la muestra, sino una parte

de ella la cual llamaremos **minibatch** (y que en nuestro caso será de **64** de tamaño).
La función **dErr(minibatch_x, minibatch_y, w)** es para calcular la derivada del error. Corresponde a la siguiente fórmula vista en teoría:

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

En la implementación tenemos tanto ésta fórmula como la del error (sin derivar), que también utilizaremos en los siguientes ejercicios:

```python
# Funcion para calcular el error
def Err(x, y, w):
    y = y.reshape(-1,1)
    # Realizamos el producto de x*w, le restamos 'y' y lo elevamos al cuadrado
    error = (x.dot(w) - y)**2
    # Hacemos la media
    error = np.mean(error, axis=0)
    # Asignamos una forma determinada al array
    error = error.reshape(-1, 1)

    return error


# Funcion para calcular la derivada del error
def dErr(x, y, w):
    # Asignamos un shape a la 'y' para que no de error
    y = y.reshape(-1,1)
    # Realizamos el producto de x*w y le restamos 'y'
    dError = x.dot(w) - y
    # Hacemos la media
    dError = 2*np.mean(x*dError, axis=0)
    # Asignamos una forma determinada al array
    dError = dError.reshape(-1,1)

    return dError
```

Ilustración 9. Implementación de las funciones error y derivada del error

Para leer los datos el profesor nos proporcionó una función **readData()** ya implementada. Una vez extraídos los datos, utilizamos las funciones implementadas para sacar un modelo de regresión lineal.

En mi caso voy a pintar en la misma gráfica tanto el modelo calculado por la pseudoinversa como el modelo calculado por el gradiente descendente estocástico.

Después de esto, también vamos a valorar la bondad del resultado usando Ein y Eout. Para el caso de Eout vamos a calcular las predicciones usando los datos del fichero de test.

Lectura de los datos y cálculo de los modelos de regresión de la pseudoinversa y del gradiente descendente estocástico:

```
######## EJERCICIO 1 ########

# Lectura de los datos de entrenamiento
x, y = readData('datos/X_train.npy', 'datos/y_train.npy')
# Lectura de los datos para el test
x_test, y_test = readData('datos/X_test.npy', 'datos/y_test.npy')

# Cálculo de la w del modelo de regresión utilizando gradiente descendente estocástico
sgdW = sgd(x, y, np.zeros((3,1)), 0.01, 2000)
# Cálculo de la w del modelo de regresión utilizando la pseudoinversa
pinvW = pseudoinverse(x,y).transpose()
pinvW = pinvW.reshape((3,1))

# Genero puntos del tamaño de y entre 0 y 1. Luego calculo su valor en 'y' para el sgd
sgdX = np.linspace(0, 1, y.size)
sgdY = (-sgdW[0] - sgdW[1]*sgdX) / sgdW[2]

# Genero puntos del tamaño de y entre 0 y 1. Luego calculo su valor en 'y' para la pseudoinversa
pinvX = np.linspace(0, 1, y.size)
pinvY = (-pinvW[0] - pinvW[1]*pinvX) / pinvW[2]
```

Ilustración 10. Cálculo de los modelos de regresión.

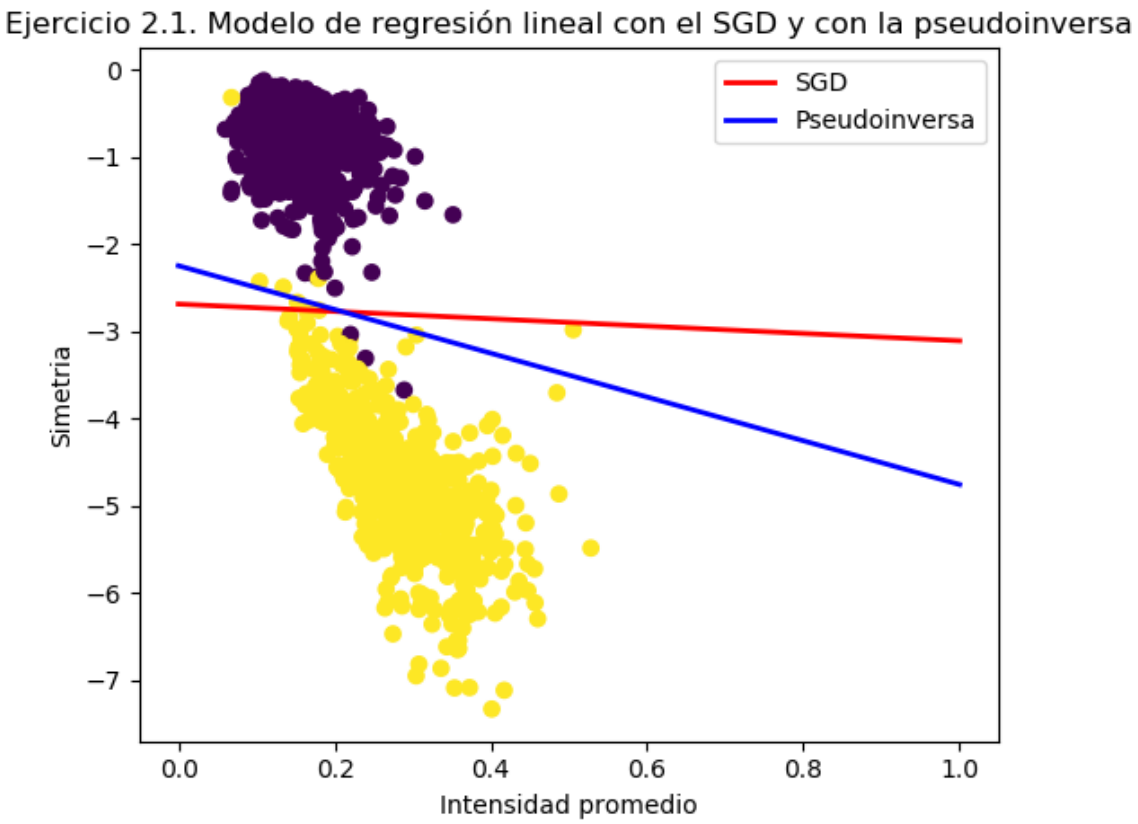

Ilustración 11. Gráfica de los modelos de regresión

Bondad del resultado para el gradiente descendente estocástico (**SGD**):

- Ein: **0.08176076** - Eout: **0.13543899**

Bondad del resultado para la **pseudoinversa**:

- Ein: **0.07918659** - Eout: **0.13095384**

Como podemos observar, los errores de los dos modelos de regresión son muy bajos. Esto es porque los datos son linealmente separables, es decir, con una línea los podemos separar en dos "categorías".

Los errores que salen con los ficheros del test también son bajos. Esto es porque la línea está relativamente bien trazada y para la mayoría de puntos nuevos que llegan, los clasifica correctamente.

Ejercicio 2

Aparatado a). Tenemos que crear la función **simula_unif(N, d, size)** para crear una muestra de 1000 puntos en el cuadrado X = [-1,1] x [-1,1] y sacarlo por pantalla. A continuación, la implementación de la función y el gráfico resultante:

```
# Apartado a)
# Simula datos en un cuadrado [-size,size]x[-size,size]
def simula_unif(N, d, size):
    return np.random.uniform(-size, size, (N, d))

# Creamos una muestra de 1000 puntos en el cuadrado X = [-1,1] x [-1,1]
x = simula_unif(1000, 2, 1);
plt.scatter(x[:,0], x[:,1])
plt.show();
```

Ilustración 12. Implementación del simula_unif()

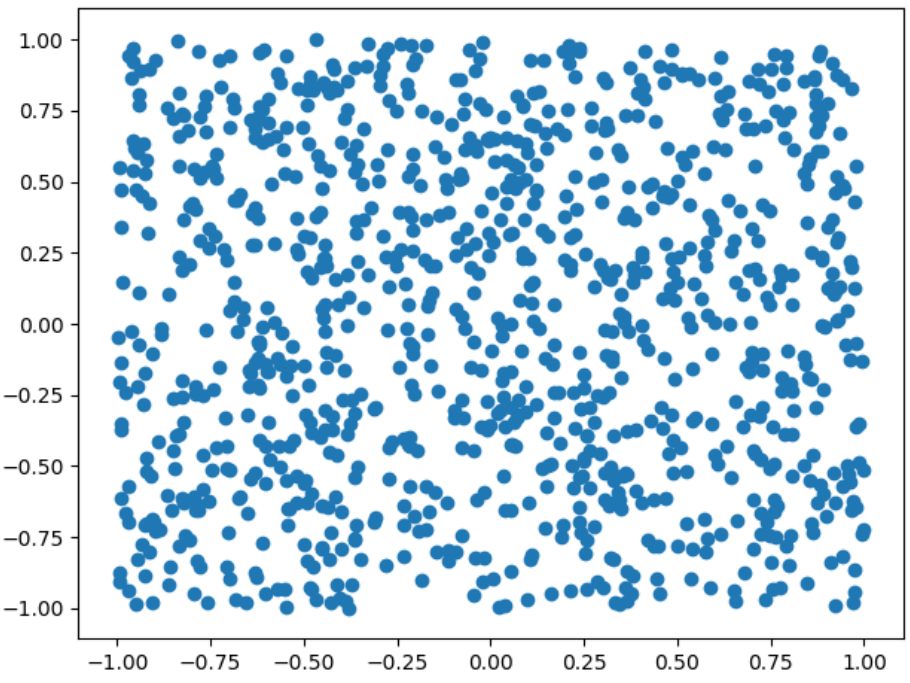

Ilustración 13. Mapa de puntos generado

Apartado b). Utilizar la función $\mathbf{f(x_1, x_2) = sign\big((x_1 - 0,2)^2 + x_2^2 - 0,6\big)}$ para asignar una etiqueta a cada punto de la muestra. Además, añadimos un 10% de **ruido** aleatoriamente a las etiquetas.

```
# Apartado b)
# Función para asignar una etiqueta a cada punto
def f(x1, x2, ruido):
    f = []  # Creamos un array vacío

    # Recorremos todos los valores de x1 y x2 (que tienen el mismo tamaño)
    for i in range(x1.size):
        # Si el resultado de la función es mayor que 0, le asignamos un 1
        if ((x1[i]-0.2)**2 + x2[i]**2 - 0.6) >= 0:
            f.append(1)
        # Se es menor, un -1
        else:
            f.append(-1)

    f = np.array(f)   # Lo convertimos a un np.array

    # Si queremos añadirle ruido, dependerá del parámetro pasado como argumento
    if ruido:
        # Sacamos indices aleatorios con el 10% del tamaño
        index = np.random.choice(int(f.size), size=int((f.size)/10), replace=False)
        # Cambiamos los 1 por -1 y viceversa
        for i in range(f.size):
            if np.isin(i,index):
                f[i] = -f[i]

    return f


#Calculamos un nuevo 'y' con ruido y lo mostramos por pantalla
y = f(x[:,0],x[:,1],True)
plt.scatter(x[:,0], x[:,1], c=y)
plt.show()
```

Ilustración 14. Asignación de etiquetas y de ruido

El resultado de este nuevo mapa de etiquetas con ruido es el siguiente:

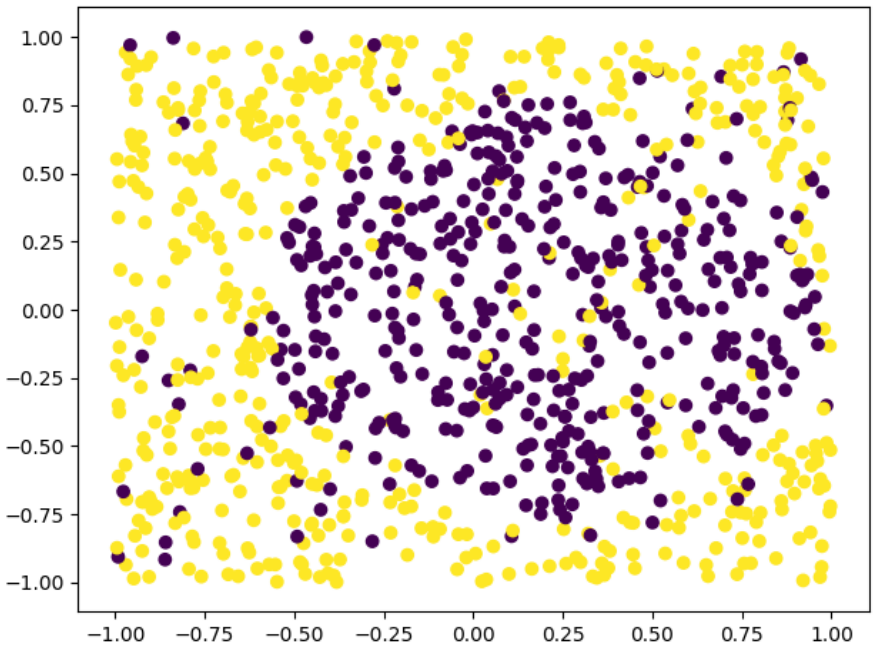

Ilustración 15. Mapa de etiquetas con ruido.

Apartado c). Ahora tendremos que ajustar un modelo de regresión al mapa de puntos generado anteriormente. Para ello, tendremos que añadir primero una columna de unos al principio del array para convertirlo en la forma **(1, x_1, x_2).**

Después estimaremos el error de ajuste Ein utilizando el SGD. La implementación del código es la siguiente:

```
# Apartado c)
# Añadimos una columna de 1 al principio de la matriz
x = np.c_[np.ones((1000, 1), np.float64), x]
# Calculamos el SGD y éste los valores para 'x' e 'y' para la línea de regresión
w = sgd(x, y, np.zeros((3,1)), 0.01, 200)
lineX = np.linspace(-1, 1, y.size)
lineY = (-w[0] - w[1]*lineX) / w[2]

# Mostramos el gráfico por pantalla
plt.scatter(x[:,1], x[:,2], c=y)
plt.plot(lineX, lineY, 'r-', linewidth=2)

plt.title('Ejercicio 2.2. Modelo de regresión lineal con ruido')
plt.ylim(-1.0,1.0)
plt.show()
```

Ilustración 16. Asignación del modelo de regresión.

El error Ein que sale es de: **0.94686308.**

Adicionalmente, voy a mostrar la gráfica de cómo se vería el modelo de regresión calculado en el mapa. Hay que tener en cuenta que se generan aleatoriamente, por lo que el modelo podría cambiar:

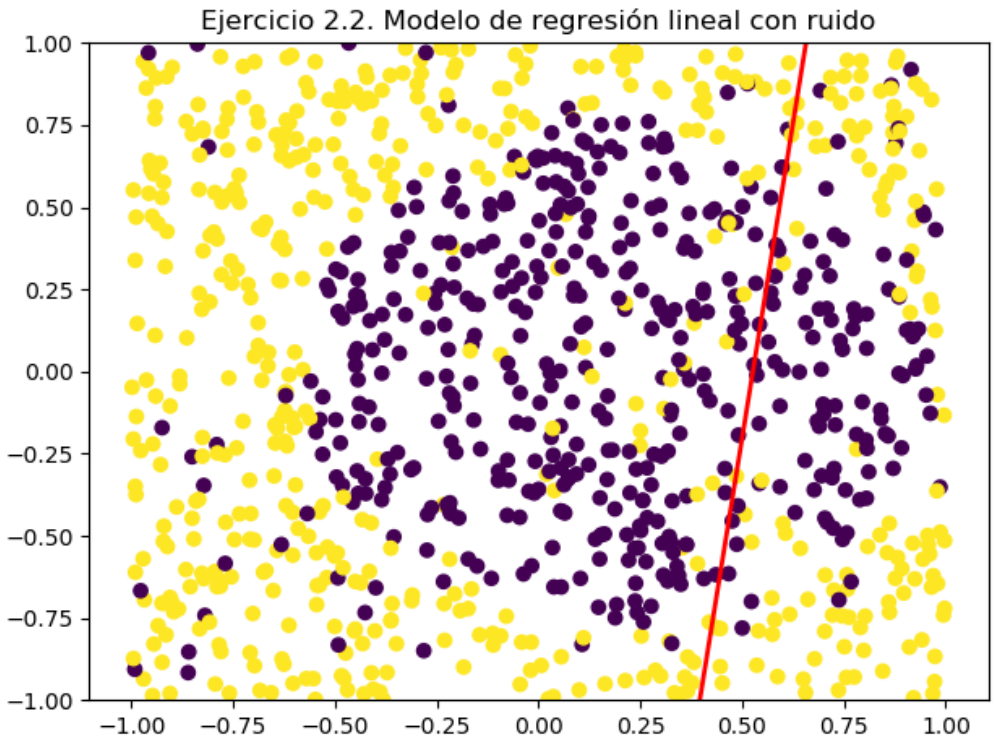

Ilustración 17. Línea de regresión en el mapa.

Apartado d). En este apartado tenemos que realizar los apartados a) y c) 1000 veces, es decir, coger 1000 muestras como hemos hecho en el apartado a) y luego **realizar 1000 veces el experimento**.

A su vez, iremos guardando los valores tanto de Ein como de Eout en cada repetición para luego realizar la media y poder valorar los resultados en el apartado e).

A continuación, la implementación del código del apartado. También hay que decir que generamos muchas muestras aleatorias y realizamos el bucle muchas veces, asi que el código tarda bastante en ejecutarse:

```
# Apartado d)
M = 1000    # Número de experimentos
N = 1000    # Número de muestras

# Inicializamos los siguientes valores a 0
iterations = 0
EinM = 0    # Media del Ein
EoutM = 0   # Media del Eout

while iterations < M:
    # Generamos una muestra aleatoria como hemos hecho anteriormente sin ruido
    x = simula_unif(N, 2, 1)
    y = f(x[:,0], x[:,1], False)
    x = np.c_[np.ones((N, 1), np.float64), x]

    # Generamos una muestra aleatoria para el test, esta vez con ruido
    x_test = simula_unif(N, 2, 1)
    y_test = f(x_test[:, 0], x_test[:, 1], True)
    x_test = np.c_[np.ones((N, 1), np.float64), x_test]

    # Obtenemos el SGD para calcular los errores
    w = sgd(x, y, np.zeros((3,1)), 0.01, 200)
    EinM = EinM + Err(x, y, w)                  # Vamos acumulando los valores
    EoutM = EoutM + Err(x_test, y_test, w)      # en estas variables
    iterations += 1

# Una vez terminado, dividimos por el M para obtener la media
EinM = EinM/M
EoutM = EoutM/M

# Imprimimos los errores
print('Valores medios de los errores:')
print("Ein: ", EinM)
print("Eout: ", EoutM)
```

Valores medios de los errores:

- Ein: **0.92275841** - Eout: **0.95829303**

Apartado e). Podemos ver claramente cómo los valores de los errores son muy altos en este caso, y no es como en el ejercicio anterior, que apenas nos salían errores de 0.1 aproximadamente. Esto se debe a que el modelo generado no se puede dividir linealmente, es decir, con una línea recta no puedes dividir el conjunto en dos, y si además le añadimos ruido, lo complicamos todavía más.

Obviamente el resultado de Eout no va a ser mucho mejor, ya que pese a ser muestras aleatorias, la línea de regresión apenas divide los datos correctamente (como hemos visto en la Ilustración 17).