APRENDIZAJE AUTOMÁTICO

DOCUMENTACIÓN PRÁCTICA 1:

Ejercicio sobre la búsqueda iterativa de óptimos y Regresión Lineal

Universidad de Granada

GRUPO A3
(Viernes 17:30-19:30)

CURSO 2018-2019

Realizado por:
Jesús Medina Taboada
DNI:
jemeta@correo.ugr.es

ÍNDICE

Ejercicio 1: búsqueda iterativa de óptimos

1.Implementar el algoritmo de gradiente descendente

```
def gradient_descent(w, eta, grad_fun, fun, error2get, maxIter):
    iterations = 0

    while iterations <= maxIter and fun(w[0],w[1]) >= error2get:
        w = w-eta*grad_fun(w[0],w[1])
        iterations += 1
    return w, iterations
```

Función que recibe de argumento el vector w, la tasa de aprendizaje, el gradiente de una función, la función, el error máximo y el número máximo de iteraciones.
Con ello aplica la fórmula del gradiente descendente.

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

Devuelve el vector y el número de iteraciones en que se ha completado.

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 η = 0,01.

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

Partimos de la expresión dada y calculamos sus derivadas parciales para poder obtener el gradiente.

Definimos las funciones a partir de las derivadas parciales que hemos obtenido

```
#Funcion a estudiar
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)])
```

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

Partiendo de los datos siguientes llamamos a la función del gradiente descendente e imprimimos las coordenadas pertinenetes.

```
eta = 0.01
maxIter = 10000000000
error2get = 1e-14
initial_point = np.array([1.0,1.0])

w, it = gradient_descent(initial_point, eta, gradE, E, error2get, maxIter)


print ('Numero de iteraciones: ', it)
print ('Coordenadas obtenidas: (', w[0], ', ', w[1],')')
input("\n--- Pulsar tecla para continuar ---\n")
```

SOLUCIÓN: para E(u,v) obtenemos un valor por debajo de 10^{-14} después de 33 iteraciones.

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

Al final de la imagen del apartado anterior observamos que nos mostrará las coordenadas obtenidas cuando se alcance la condición de valor menos que 10^{-14}.

SOLUCIÓN: para E(u,v) las coordenadas que alcanzan por primera vez un valor inferior a 10^{-14} son (0.619207678450638 , 0.9684482690100485).

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

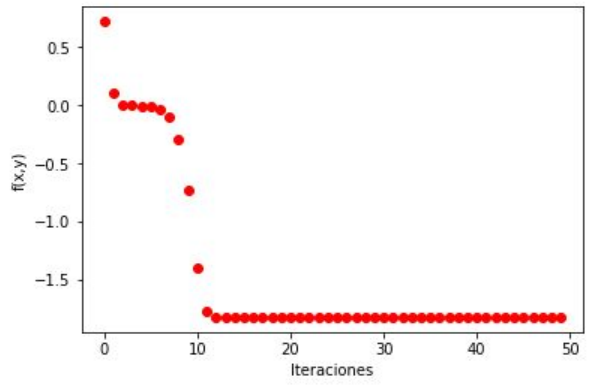
Repetimos el proceso del ejercicio anterior. A partir de la función dada, calculamos sus derivadas parciales y las definimos junto con la función principal.

```
#Segunda función a estudiar
def f(u,v):

    return u**2+2*v**2+2*math.sin(2*math.pi*u)*math.sin(2*math.pi*v)

# Derivada parcial de f respecto de u
def fu(u,v):

    return 2*u+4*math.pi*math.cos(2*math.pi*u)*math.sin(2*math.pi*v)

# Derivada parcial de f respecto de v
def fv(u,v):

    return 4*v+4*math.pi*math.sin(2*math.pi*u)*math.cos(2*math.pi*v)

# Gradiente de f
def gradf(u,v):
    return np.array([fu(u,v), fv(u,v)])
```

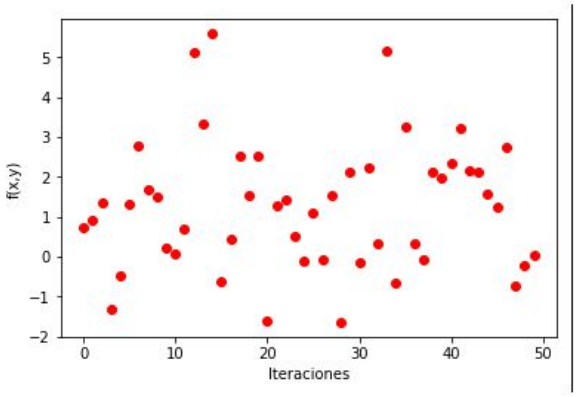
3.a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial (x_0 = 0,1, y_0 = 0,1), (tasa de aprendizaje η = 0,01 y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando η = 0,1, comentar las diferencias y su dependencia de η.

```python
# -----APARTADO A-----
# Usamos gradiente descendente con la función f para minimizarla.
# Los valores iniciales seran:
#           -Punto partida [1,1]
#           -Tasa de aprendizaje 0.01 y luego 0.1
#           -Máximo iteraciones

def gra_desc_fun(w, eta, grad_fun, fun, maxIter=50):
    graf = [fun(w[0],w[1])]
    for k in range(1,maxIter):
        w = w-eta*grad_fun(w[0],w[1])
        graf.insert(len(graf),fun(w[0],w[1]))

    plt.plot(range(0,maxIter), graf, 'bo')
    plt.xlabel('Iteraciones')
    plt.ylabel('f(x,y)')
    plt.show()

print ('Resultados ejercicio 2\n')
print ('\nGrafica con tasa de aprendizaje igual a 0.01')
gra_desc_fun(np.array([0.1,0.1]) , 0.01, gradf, f)
print ('\nGrafica con tasa de aprendizaje igual a 0.1')
gra_desc_fun(np.array([0.1,0.1]) , 0.1, gradf, f)
input("\n--- Pulsar tecla para continuar ---\n")
```

Grafica con tasa de aprendizaje igual a 0.01



Grafica con tasa de aprendizaje igual a 0.1



Las gráficas obtenidas nos muestran que el algoritmo gradiente descendente tiene una alta relación con la tasa de aprendizaje ya que en la primera gráfica encuentra un mínimo local rápidamente y en la segunda no.

En la primera con un learning rate de 0.01 converge rápidamente mínimo, pero con un learning rate de 0.1 no lo encuentra.

3.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

Para empezar definiremos otra vez el algoritmo de gradiente descendente pero esta vez con la nueva función f y le cambiaremos el máximo de iteraciones a 50.

```
# -----APARTADO B-----
# Obtener el valor minimo y los valores de las variables (x,y) se alcanzan
# cuando el punto de inicio se fija:
# (0,1, 0,1)
# (1, 1)
# (-0,5, -0,5)
# (-1, -1)

def gd(w, eta, grad_fun, fun, maxIter = 50):
    for i in range(0,maxIter):
        w = w-eta*grad_fun(w[0],w[1])

    return w
```

A continuación llamaremos a la función para distintos puntos de inicio y comprobaremos sus valores.

```
#Primer punto (0.1,0.1)

w = gd(np.array([0.1, 0.1]) , 0.01, gradf, f)
print ('Punto de inicio: (0.1, 0.1)\n')
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Minimo: ',f(w[0],w[1]))

input("\n--- Pulsar tecla para continuar ---\n")


#Segundo punto (1,1)

w = gd(np.array([1,1]) , 0.01, gradf, f)
print ('Punto de inicio: (1,1)\n')
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Minimo: ',f(w[0],w[1]))

input("\n--- Pulsar tecla para continuar ---\n")


#Tercer punto (-0.5,-0.5)

w = gd(np.array([-0.5,-0.5]) , 0.01, gradf, f)
print ('Punto de inicio: (-0.5,-0.5)\n')
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Minimo: ',f(w[0],w[1]))

input("\n--- Pulsar tecla para continuar ---\n")


#Cuarto punto (-1.0, -1.0)

w = gd(np.array([-1.0, -1.0]) , 0.01, gradf, f)
print ('Punto de inicio: (-1.0, -1.0)\n')
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Mínimo: ',f(w[0],w[1]))

input("\n--- Pulsar tecla para continuar ---\n")
```

Como conclusión obtenemos esta tabla:

Punto de inicio	(X,Y)	f(X,Y)
(0.1, 0.1)	(0.24380496936478835 , -0.23792582148617766)	-1.8200785415471563
(1,1)	(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

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

A la hora de encontrar el mínimo global de una función arbitraria tenemos que tener dos factores en cuenta, el punto de inicio y la tasa de aprendizaje.
El punto de inicio será clave ya que si hay más de un mínimo nos va a determinar a qué mínimo local encuentra ya que no podrá salir de él tras encontrarlo lo que podría conllevar no encontrar el mínimo global, que es lo que buscamos.
Por otra parte tenemos el learning rate que nos influye crucialmente ya que si le damos un valor pequeño puede tardar demasiado tiempo en converger, y sin embargo si le damos un valor muy alto podríamos no encontrar un mínimo nunca.
Por ello concluyo que ambos factores son claves aunque el segundo es más difícil de determinar ya que no podemos pasarnos ni quedarnos cortos en lo que a la tasa de aprendizaje se refiere.

Ejercicio 2: Regresión Lineal

Este ejercicio ajusta modelos de regresión a vectores de características extraídos de imágenes de dígitos manuscritos. En particular se extraen dos características concretas: el valor medio del nivel de gris y simetría del número respecto de su eje vertical. Solo se seleccionarán para este ejercicio las imágenes de los números 1 y 5.

Para empezar creamos una función para leer los archivos que usaremos de referencia y nos aseguramos de clasificar solo los números 1 y 5.

```
print('EJERCICIO SOBRE REGRESION LINEAL\n')
print('Ejercicio 1\n')

label5 = 1
label1 = -1

# Funcion para leer los datos
def readData(file_x, file_y):
    # Leemos los ficheros
    datax = np.load(file_x)
    datay = np.load(file_y)
    y = []
    x = []
    # Solo guardamos los datos cuya clase sea la 1 o la 5
    for i in range(0,datay.size):
        if datay[i] == 5 or datay[i] == 1:
            if datay[i] == 5:
                y.append(label5)
            else:
                y.append(label1)
            x.append(np.array([1, datax[i][0], datax[i][1]]))

    x = np.array(x, np.float64)
    y = np.array(y, np.float64)

    return x, y
```

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).

Implementamos las 3 funciones que necesitaremos para este ejercicio, la del Error la del Gradiente Descendente Estocástico y la del algoritmo pseudoinversa.

```
# Funcion para calcular el error
def Error(x,y,w):
    return (1/y.size)*np.linalg.norm(x.dot(w)-y)**2

# Funcion del Gradiente Descendente Estocastico
def gde(x, y, eta, maxIter, minibatch):
    w = np.zeros(len(x[0]), np.float64)
    posicion = np.array(range(0,x.shape[0]))

    for k in range(0,maxIter):
        w_2 = w
        for j in range(0,w.size):
            suma = 0
            # Desordenamos las posiciones para el minibatch
            np.random.shuffle(posicion)
            tam_minibatch = minibatch
            # Hacemos sumatoria de la función
            suma = (np.sum(x[posicion[0:tam_minibatch:1],j]*
                  (x[posicion[0:tam_minibatch:1]].dot(w_2) -
                   y[posicion[0:tam_minibatch:1]])))

            w[j] -= eta*(2.0/tam_minibatch)*suma

    return w

# Funcion de Pseudoinversa
def pseudoinversa(x, y):

    u,d,vt = np.linalg.svd(x)
    d_inversa = np.linalg.inv(np.diag(d))
    v = vt.transpose()

    # Calculo de la pseudoinversa de x
    x_inv = v.dot(d_inversa).dot(d_inversa).dot(v.transpose()).dot(x.transpose())
    w = x_inv.dot(y)
    return w
```

A continuación necesitamos hacer la lectura de los ficheros externos de entrenamiento y de test que están alojados en el directorio de trabajo/datos/(nombre de archivos).

Ejecutamos la función del Gradiente Descendente Estocástico con una tasa de aprendizaje de 0.01 como el ejercicio anterior un máximo de iteraciones de 500 y un minibatch de tamaño 64.
Tras esto mostramos la bondad de de los resultados a partir de E_{in} y E_{out}

```
# Leemos datos de para el test y lo sde entrenamiento
x_test, y_test = readData('datos/X_test.npy', 'datos/y_test.npy')
x, y = readData('datos/X_train.npy', 'datos/y_train.npy')


#Llamamos a la funcion del gradiente descendente estocastico
w = gde(x, y, 0.01, 500, 64)

print ('La bondad del resultado obtenido para el la funcion gradiente\
       descendiente estocastico a través de Ein y Eout es:\n')
print ("Ein:", Error(x,y,w))
print ("Eout:", Error(x_test, y_test, w))
```

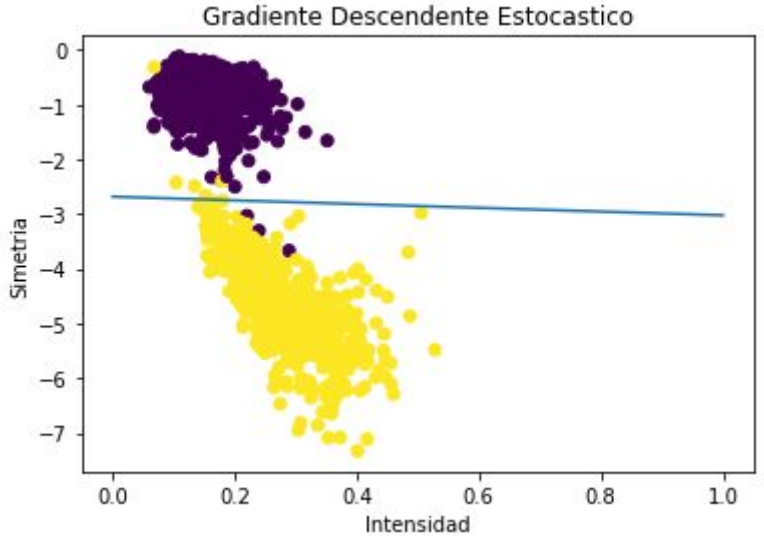
Tras la ejecución obtenemos:

E_{in} -> 0.08207838897626317

E_{out} -> 0.13696552396953135

Realizamos una gráfica representativa.

```
# Hacemos el grafico teniendo en cuenta las columnas 1 y 2 de la variable x
# que son la intensidad y la simetria. Por otra parte separamos los valores de
# y por la clase a la que pertenezcan

plt.scatter(x[:,1],x[:,2], c=y)
plt.plot([0, 1], [-w[0]/w[2], -w[0]/w[2]-w[1]/w[2]])
plt.title('Gradiente Descendente Estocastico')
plt.ylabel('Simetria')
plt.xlabel('Intensidad')
plt.show()
```



Repetimos el proceso con la función pseudoinversa.

```
#Llamamos a la funcion de la pseudoinversa
w = pseudoinversa(x, y)

print ('La bondad del resultado obtenido para el la funcion pseudoinversa\
       a través de Ein y Eout es:\n')
print ("Ein: ", Error(x,y,w))
print ("Eout: ", Error(x_test, y_test, w))
```

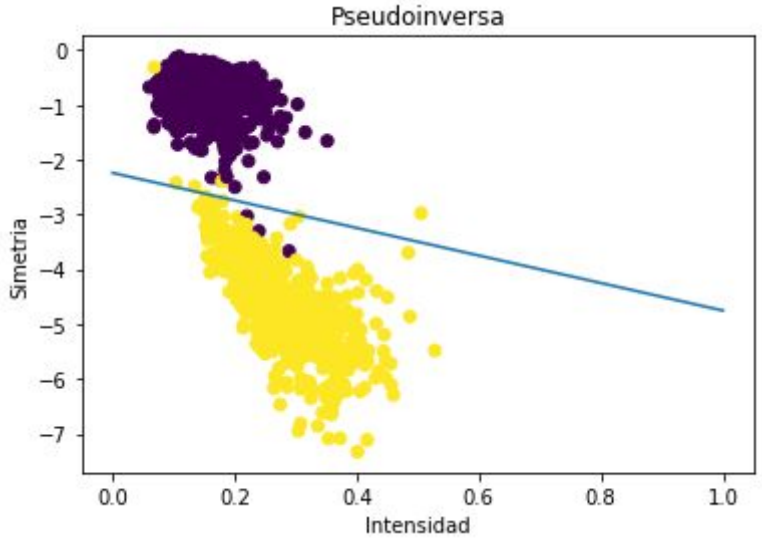
Tras la ejecución obtenemos:

E_{in} -> 0.07918658628900395

E_{out} -> 0.13095383720052586

Y realizamos la gráfica correspondiente

```
plt.scatter(x[:,1],x[:,2], c=y)
plt.plot([0, 1], [-w[0]/w[2], -w[0]/w[2]-w[1]/w[2]])
plt.title('Pseudoinversa')
plt.ylabel('Simetria')
plt.xlabel('Intensidad')
plt.show()
```



VALORACIÓN: al comparar las gráficas y ver los errores vemos que el algoritmo de la Pseudoinversa se ajusta más con una mayor separación visual y en datos ya que el E_{in} y el E_{out} son algo más bajos en comparación con el Gradiente Descendente Estocástico.

2. En este apartado exploramos cómo se transforman los errores E_{in} y E_{out} 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]

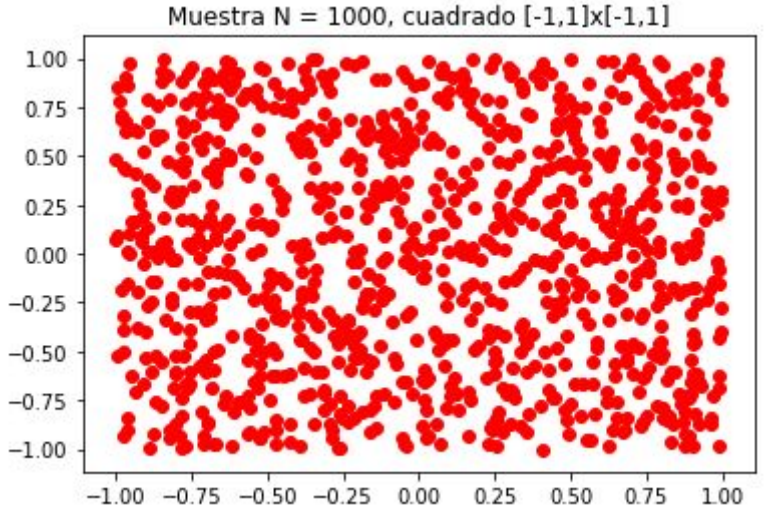
```
# Funcion simula_unif(N, 2, size) que nos devuelve N coordenadas 2D de
#puntos uniformemente muestreados dentro del cuadrado definido por
# [-size, size] × [-size, size]
def simula_unif(N, d, size):
    return np.random.uniform(-size,size,(N,d))
```

EXPERIMENTO

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

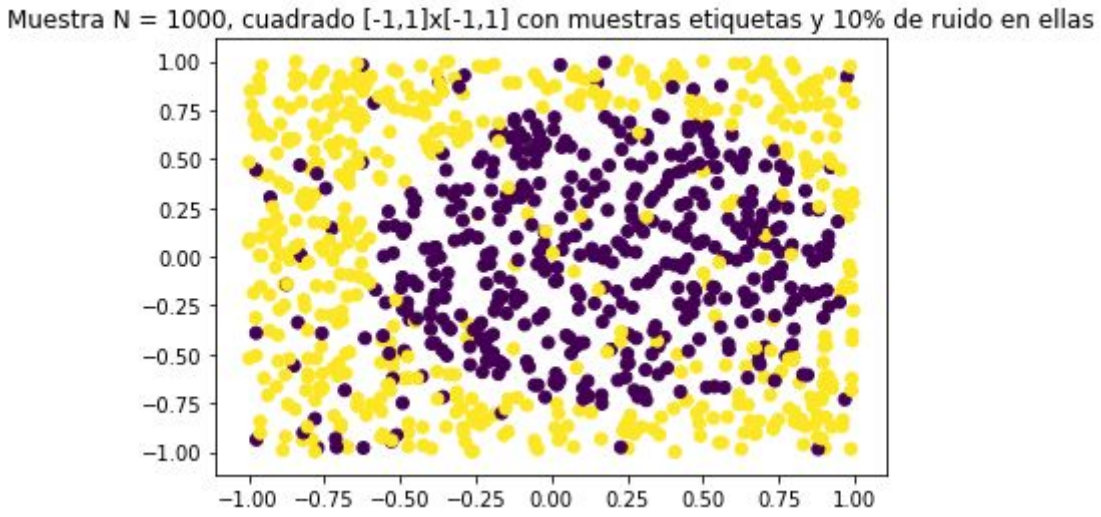
```
# -----APARTADO A-----
# Generamos una muestra de entrenamiento con N = 1000 puntos en el cuadrado
# X = [-1, 1] × [-1, 1] y lo pintamos en un mapa de puntos 2D


plt.title('Muestra N = 1000, cuadrado [-1,1]x[-1,1]')
x = simula_unif(1000, 2, 1)
plt.scatter(x[:,0],x[:,1])
plt.show()
```

Generamos la gráfica y no podemos destacar nada porque no hay etiquetas:



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.

```
# -----APARTADO B-----
# Creamos una funcion sing para devolver 1 o -1 segun el valor que se recibe
# como argumento.

# Funcion signo
def sign(x):
    if x >= 0:
        return 1
    return -1
# Creamos f2 que se encarga de asignar etiquetas a la muestra x
def f2(x1, x2):
    return sign((x1-0.2)**2+x2**2-0.6)

# Introducimos 10% de ruido a traves de un array con 10% de aletoriedad
# que simula la introducción de ruido
p = np.random.permutation(range(0,x.shape[0]))[0:x.shape[0]//10]

# Ordenamos el array obtenido
p.sort()
j = 0
y = []

for i in range(0,x.shape[0]):
    # Si i está en p cambiamos el signo si no lo mantenemos
    if i == p[j]:
        j = (j+1)%(x.shape[0]//10)
        y.append(-f2(x[i][0], x[i][1]))
    else:
        y.append(f2(x[i][0], x[i][1]))

x = np.array(x, np.float64)
y = np.array(y, np.float64)

plt.title('Muestra N = 1000, cuadrado [-1,1]x[-1,1] con muestras etiquetas\
          y 10% de ruido en ellas')
plt.scatter(x[:,0],x[:,1], c=y)
plt.show()
```



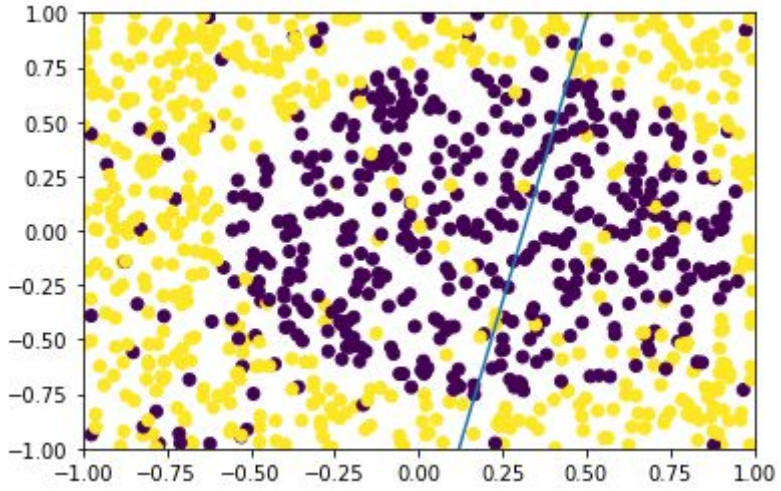
VALORACIÓN: al tener etiquetas ya podemos diferenciar dos subgrupos

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 Ein usando Gradiente Descendente Estocástico (SGD).

```
# -----APARTADO 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)

# Creamos array de unos para la regresion lineal (1, x0, x1)
q = np.array([np.ones(x.shape[0], np.float64)])
x = np.concatenate((q.T, x), axis = 1)

w = gde(x, y, 0.01, 1000, 64)

print ('Error de ajuste Ein usando Gradiente Descediente estocastico')
print ('con 1000 iteraciones')
print ("\n\nEin: ", Error(x,y,w))
plt.scatter(x[:,1],x[:,2], c=y)
plt.plot([0, 1], [-w[0]/w[2], -w[0]/w[2]-w[1]/w[2]])
plt.axis([-1,1,-1,1])
plt.show()
```

Tras la ejecución obtenemos:

E_{in} -> 0.9565534009344919



VALORACIÓN: como podemos observar la recta de regresión es muy poco representativa, y esto es debido al error tan grande que hemos obtenido(0,95~).

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.

```
# -----APARTADO D-----
# Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000
# muestras diferentes) y:
#     • Calculamos el valor medio de los errores Ein de las 1000 muestras.
#     • Generamos 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.

#Declaramos las variables de la media de Ein y Eout
Ein = 0
Eout = 0

for k in range(0,1000):
    x = simula_unif(1000, 2, 1)

    # Array de unos para la regresion lineal (1, x0, x1)
    q = np.array([np.ones(x.shape[0], np.float64)])
    x = np.concatenate((q.T, x), axis = 1)
    y = []

    # Array con 10% de indices aleatorios para introducir ruido
    b = np.random.permutation(range(0,x.shape[0]))[0:x.shape[0]//10]
    b.sort()
    j = 0

    for i in range(0,x.shape[0]):
        if i == b[j]:
            j = (j+1)%(x.shape[0]//10)
            y.append(-f2(x[i][1], x[i][2]))
        else:
            y.append(f2(x[i][1], x[i][2]))

    y = np.array(y, np.float64)

    # Solo 10 iteraciones y minibatch de 32 para que la ejecucion
    # no tarde demasiado tiempo
    w = gde(x, y, 0.01, 10, 32)
```

```
    # Simulamos los datos del test para despues calcular el Eout
    x_test = simula_unif(1000, 2, 1)
    x_test = np.concatenate((q.T, x_test), axis = 1)



    Ein += Error(x,y,w)
    Eout += Error(x_test,y,w)

Ein /= 1000
Eout /= 1000
```

CONCLUSIÓN: la media de los errores E_{in} y E_{out} tras el experimento son

-> E_{in} = 0.9901838960913539

-> E_{out} = 0.9993185733209056

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}

La valoración final de qué tan bueno es este modelo es muy impreciso y de poca utilidad de cara a realizar un ajuste eficaz.
Tras hacer el último experimento y comprobar los valores medios de los errores nos damos cuenta que son altísimos (0.99) en ambos casos y por lo tanto no se puede realizar el ajuste.
Por otra parte tenemos las gráficas que nos muestran que obviamente no funciona nuestro modelo ya que la recta de regresión que obtenemos no nos indica nada.
Un buen modelo para realizar el ajuste podría ser con circunferencias o curvas ya qu es lo que se intuye a través de las gráficas.