Trabajo 1: Programación

Ignacio Vellido Expósito

1. Ejercicio sobre la búsqueda iterativa de óptimos

Apartado 1

```
"""
Gradiente descendente
Argumentos:
  Tasa de aprendizaje: eta
  Nº máximo de iteraciones: maxIter
  Tasa de error: error2get - En este caso valor a alcanzar
  Punto inicial: initial_point
"""
def gradient_descent(eta, maxIter, error2get, initial_point):
    # Variables auxiliares para el algoritmo
    iterations = 0
    surpassedMinError = False

    # Inicializamos w
    w = np.copy(initial_point)

    while iterations < maxIter and not surpassedMinError:
      gradient = eta * gradE(w[0], w[1])
      w = w - gradient
      error = E(w[0], w[1])

      surpassedMinError = True if error < error2get else False
      iterations += 1

    return w, iterations
```

Se sigue la implementación del pseudocódigo visto en clase, pero orientándolo al apartado 2. Por ello **error** equivaldrá al valor de la función y **error2get** el valor a alcanzar.

Apartado 2

Siendo: $E(u,v) = (u^2e^v - 2v^2e^{-u})^2$

Tenemos:

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

$$\frac{\partial E(u,v)}{\partial v} = 2(u^2e^v - 2v^2e^{-u}) * (u^ve^v - 4ve^{-u})$$

$$\nabla E(u,v) = \left[\frac{\partial E(u,v)}{\partial u}, \frac{\partial E(u,v)}{\partial v}\right]$$

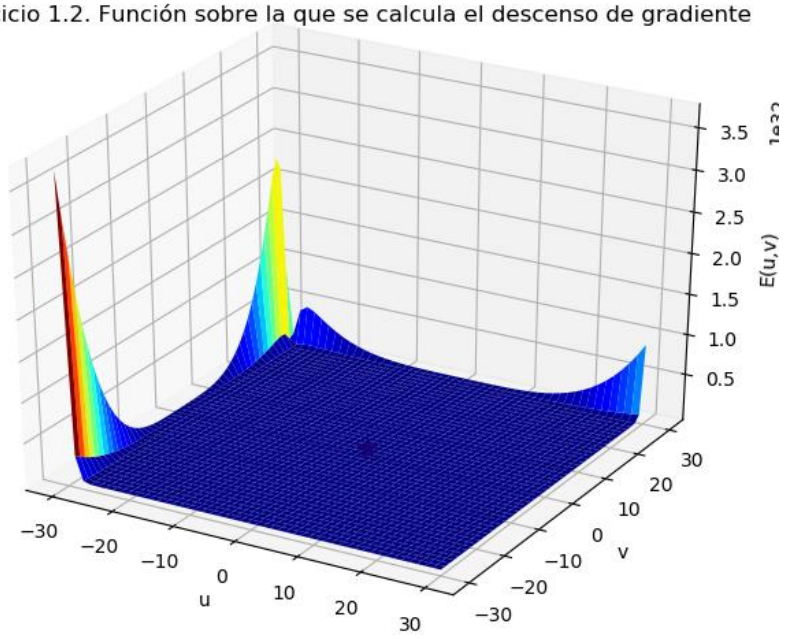
```
# Función
def E(u,v):
    return ((u**2)*(np.exp(v)) - 2*(v**2)*(np.exp(-u)))**2

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

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

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

Al aplicar el algoritmo de gradiente descendente, tras 33 iteraciones obtenemos un valor de $E(u,v)$ inferior a e^{-14} en las coordenadas *(0.6192076784506378 , 0.968448269010048)*

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

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

print ('Numero de iteraciones: ', it)
print ('Coordenadas obtenidas: (', w[0], ', ', w[1], ')')
```

```
Ejercicio 1 y 2

Numero de iteraciones:  33
Coordenadas obtenidas: ( 0.6192076784506378 ,  0.9684482690100485 )
```

Ejercicio 1.2. Función sobre la que se calcula el descenso de gradiente



Apartado 3

Siendo: $f(x,y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$

Tenemos:

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

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

$$\nabla f(x,y) = \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right]$$

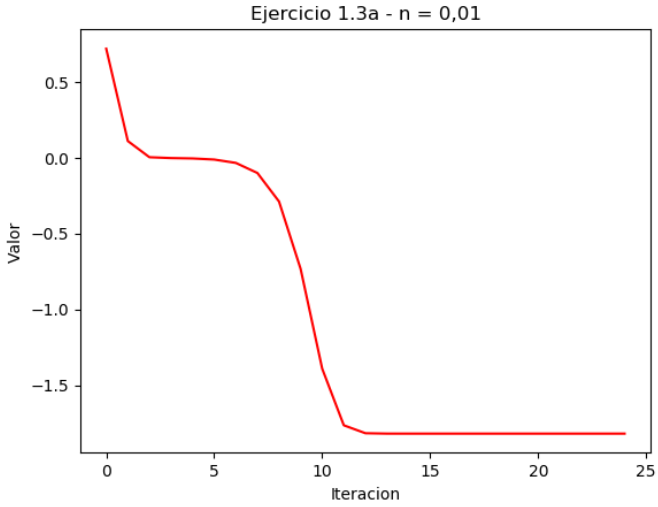
```
# Función
def F(x,y):
  return (x**2 + 2*(y**2) + 2*np.sin(2*np.pi*x)*np.sin(2*np.pi*y))

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

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

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

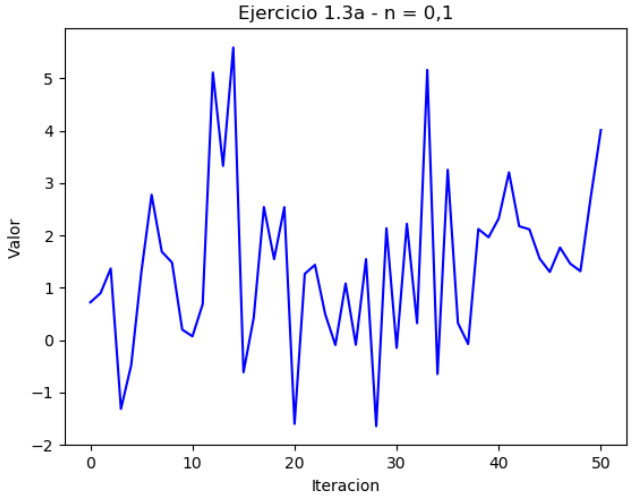
Para este apartado se modifica el código del gradiente descendente, pues estamos interesados en los valores que alcanza la función en el proceso. Al no tener un valor que alcanzar, como motivo de parada (en adición al número máximo de iteraciones) se comprueba si ha habido descenso en el valor de la función.

```
"""
Gradiente descendente que además devuelve los valores de la función durante
el proceso. Para cuando se llega al máximo de iteraciones o se alcanza un mínimo
Argumentos:
  Tasa de aprendizaje: eta
  Nº máximo de iteraciones: maxIter
  Tasa de error: error2get - No se utiliza
  Punto inicial: initial_point
"""
def printing_gradient_descent(eta, maxIter, error2get, initial_point):
    # Variables auxiliares para el algoritmo
    iterations = 0
    isMin = False

    # Inicializamos w
    w = np.copy(initial_point)

    # Inicializamos F(x,y) en el punto inicial
    values = F(w[0], w[1])

    while iterations < maxIter and not isMin:
      gradient = eta * gradF(w[0], w[1])
      old_w = np.copy(w)
      w = w - gradient
      descent = abs(F(w[0], w[1]) - F(old_w[0], old_w[1]))

      # Guardamos w
      values = np.vstack((values, F(w[0], w[1])))

      isMin = True if descent == 0 else False
      iterations += 1

    return w, iterations, values
```

La minimizamos con gradiente descendente y realizamos dos experimentos:

Primeramente **(apartado a)**, partiendo del punto *(0.1, 0.1)* y un máximo de 50 iteraciones, variamos la tasa de aprendizaje con η = 0.01 y η = 0.1. Teniendo el descenso del valor de la función:






```
eta = 0.01
maxIter = 50
error2get = 0
initial_point = np.array([0.1,0.1])

w, it, v = printing_gradient_descent(eta, maxIter, error2get, initial_point)
```

```
eta = 0.1
maxIter = 50
error2get = 0
initial_point = np.array([0.1,0.1])

w, it, v = printing_gradient_descent(eta, maxIter, error2get, initial_point)
```

Y como resultados:

```
Ejercicio 3

Apartado a)

Numero de iteraciones:  24
Coordenadas obtenidas: ( 0.24380496934646034 ,  -0.23792582148074198 )


--- Pulsar tecla para continuar ---

Numero de iteraciones:  50
Coordenadas obtenidas: ( -0.9142649340131985 ,  -1.3870133188529152 )
```

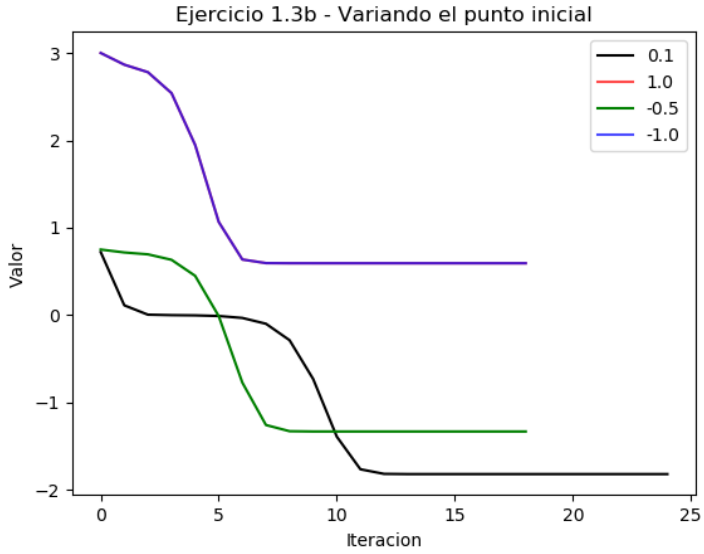
De esta manera se aprecia la importancia de elegir una buena tasa de aprendizaje para nuestro gradiente, ya que es la que ayuda a marcar la distancia del salto en la función y mientras que valores pequeños nos pueden aumentar el número de iteraciones necesarias, los valores grandes pueden hacer que no lleguemos a encontrar el mínimo.

En el segundo experimento **(apartado b)** mantenemos η = 0,1 y el máximo de iteraciones en 50, y en este caso variamos el punto inicial en (0.1, 0.1), (1, 1), (-0.5, -0.5) y (-1, -1). Resultando:

Punto inicial	Valor mínimo	(x, y) final
(0.1, 0.1)	[-1.82007854]	(0.24380496934646034 , - 0.23792582148074198)
(1, 1)	[0.59326937]	(1.2180703009052047 , 0.7128119503387537)
(-0.5, -0.5)	[-1.33248106]	(- 0.7313774598701067 , - 0.2378553629555273)
(-1, -1)	[0.59326937]	(- 1.2180703009052047 , - 0.7128119503387537)

```
eta = 0.01
maxIter = 50
error2get = 0

initial_point = np.array([0.1,0.1])
w1, it1, v1 = printing_gradient_descent(eta, maxIter, error2get, initial_point)

#-------

initial_point = np.array([1.0,1.0])
w2, it2, v2 = printing_gradient_descent(eta, maxIter, error2get, initial_point)

#-------

initial_point = np.array([-0.5,-0.5])
w3, it3, v3 = printing_gradient_descent(eta, maxIter, error2get, initial_point)

#-------

initial_point = np.array([-1.0,-1.0])
w4, it4, v4 = printing_gradient_descent(eta, maxIter, error2get, initial_point)
```

```
Apartado b)

Para [0.1,0.1]
Numero de iteraciones:  24
Coordenadas obtenidas: ( 0.24380496934646034 ,  -0.23792582148074198 )
Valor obtenido:  [-1.82007854]

Para [1.,1.]
Numero de iteraciones:  18
Coordenadas obtenidas: ( 1.2180703009052047 ,  0.7128119503387537 )
Valor obtenido:  [0.59326937]

Para [-0.5,-0.5]
Numero de iteraciones:  18
Coordenadas obtenidas: ( -0.7313774598701067 ,  -0.2378553629555273 )
Valor obtenido:  [-1.33248106]

Para [-1.0,-1.0]
Numero de iteraciones:  18
Coordenadas obtenidas: ( -1.2180703009052047 ,  -0.7128119503387537 )
Valor obtenido:  [0.59326937]
```



Desde los puntos *(1, 1)* y *(-1, -1)* se obtiene la misma gráfica, por lo que las líneas se superponen (se ha añadido transparencia para mostrarlo)

A través de las gráficas se aprecia que la función alcanza un mínimo local para el primer y tercer punto inicial, y otro mínimo de mayor profundidad para el segundo y cuarto punto.

De esto podemos deducir que, al igual que con la tasa de aprendizaje, la elección de un punto inicial influye de gran manera en la búsqueda del mínimo, ya que podemos converger en mínimos no globales.

Apartado 4

Hemos visto que gran parte del problema consiste en elegir una tasa de aprendizaje y un punto inicial apropiados, pero para elegirlos correctamente se debe conocer el comportamiento de la función, lo que no será habitual con funciones reales.

En caso de tener poca información sobre la función, una buena opción puede ser lanzar el algoritmo de gradiente descendente con parámetros arbitrarios, o al algunos que nos ayuden a descubrir el comportamiento de la función en alguna zona (como una tasa de aprendizaje baja con un número alto de iteraciones). De esa forma, obtenemos información de la función y nos permite elegir los parámetros finales con más detenimiento.

También tenemos que tener en cuenta que una tasa de aprendizaje medianamente grande puede evitarnos de caer en mínimos no globales, pero sigue siendo un arma de doble filo. Por ello, una tasa cuyo valor descienda con las iteraciones puede ser una buena aproximación al problema.

2. Ejercicio sobre Regresión Lineal

Apartado 1

```python
# Cálculo del gradiente, derivada de Ein
def dEin(x,y,w):
    M = len(x)
    xt = x.transpose()           # Transpuesta de x, para multiplicar
    h = np.dot(w, xt)            # h(x)
    diff = h - y                 # Diferencia
    sumatoria = np.dot(xt, diff)

    return np.dot((2/M), sumatoria)

# Funcion para calcular el error
def Err(x,y,w):
    M = len(x)
    xt = x.transpose()           # Transpuesta de x, para multiplicar
    h = np.dot(w, xt)            # h(x)
    diff = h - y                 # Diferencia
    sqr = np.dot(diff, diff.transpose())
    sumatoria = np.sum(sqr)

    return np.dot((1/M), sumatoria)
```

```python
"""
Gradiente Descendente Estocastico
Argumentos:
  Datos de aprendizaje: X
  Vector de salidas: Y
  Tasa de aprendizaje: eta
  Nº máximo de iteraciones: maxIter
  Tasa de error: error2get
  Punto inicial: initial_point
"""
def sgd(X, Y, eta, maxIter, error2get, initial_point):
    # Variables auxiliares
    surpassedMinError = False
    minibatch_size = 32

    # Iniciamos w
    w = np.copy(initial_point)

    # Ordenamos aleatoriamente las muestras
    fullbatch_x, fullbatch_y = shuffle(X, Y, random_state=0)

    # Separamos en minibatches
    batches_x = np.array_split(fullbatch_x, np.ceil(len(X) / minibatch_size))
    batches_y = np.array_split(fullbatch_y, np.ceil(len(X) / minibatch_size))

    for (minibatch_x, minibatch_y) in zip(batches_x, batches_y):
      iterations = 0

      # Aplicamos el algoritmo a cada minibatch
      while iterations < maxIter and not surpassedMinError:
        error = dEin(minibatch_x, minibatch_y, w)
        gradient = eta * error
        w = w - gradient

        error = Err(minibatch_x, minibatch_y, w)

        # Comprobando salida del bucle
        iterations += 1
        surpassedMinError = True if error < error2get else False

      # Devolvemos si hemos alcanzado el error
      if surpassedMinError:
        return w

    return w
```

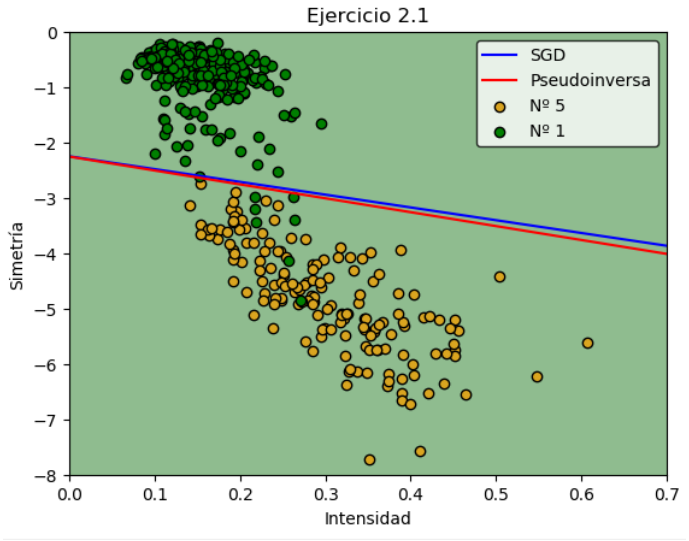
```python
# Pseudoinversa
def pseudoinverse(x,y):
    p_inverse = np.linalg.pinv(x)
    return p_inverse.dot(y)
```

Ejecutamos:

```python
# 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')

eta = 0.01
maxIter = 10000
error2get = 1e-14
initial_point = np.array([1.0,1.0,1.0])

w = sgd(x, y, eta, maxIter, error2get, initial_point)

w_p = pseudoinverse(x, y)
```

Y obtenemos:

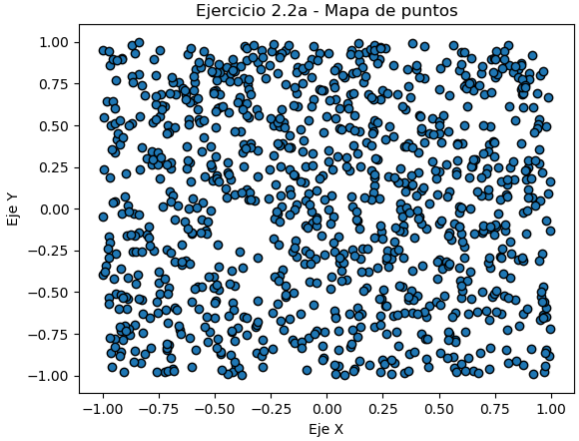
```
Ejercicio 1

Bondad del resultado para grad. descendente estocastico:

Learning rate:  0.01
Nº máximo de iteraciones:  10000
Error mínimo a alcanzar:  1e-14
Punto inicial:  [1. 1. 1.]
------------------------------
w:  [-1.14299735 -1.17841298 -0.51076567]
Ein:  0.08044961466236134
Eout:  0.13564242814111482


Bondad del resultado para la pseudoinversa:

w:  [-1.11588016 -1.24859546 -0.49753165]
Ein:  0.07918658628900396
Eout:  0.13095383720052578
```

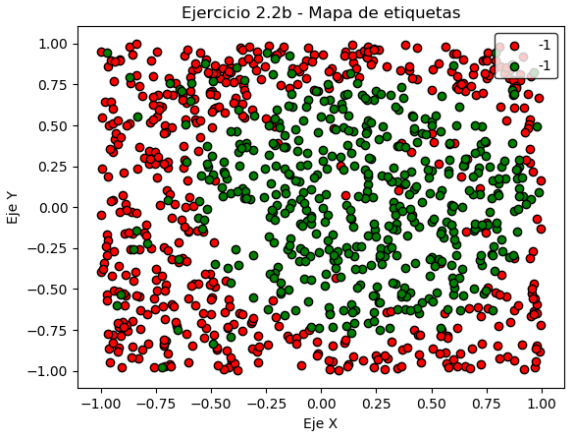


Vemos que con ambos métodos obtenemos separadores bastante similares, y con errores tanto fuera como dentro de la muestra bajos.

Apartado 2

Tenemos la muestra:





Donde a la derecha se muestran los valores por etiquetas habiendo introducido cambios con un 10% de probabilidad. En código:

```
# Función que asigna las etiquetas
# sign(x) = | -1 si x<0
#           |  1 si x>0
def setLabel(x1, x2):
  v = ((x1 - 0.2)**2) + x2**2 - 0.6

  return -1 if v < 0 else 1
```

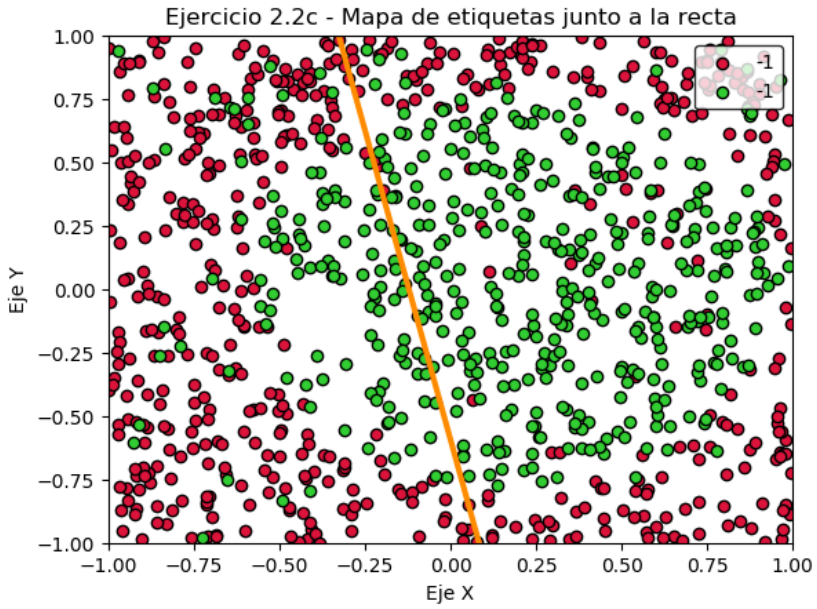
```
# Calculamos las etiquetas
labels = np.array([[0,0]])
labels = np.delete(labels, (0), axis=0)

for (a,b) in zip(train[:,0], train[:,1]):
  labels = np.append(labels, setLabel(a,b))

# Cambiamos el singo con un 10% de probabilidad
for i, l in enumerate(labels):
  n = np.random.choice(2, 1, p=[0.9, 0.1])
  if n == 1:
    labels[i] = -l
```

Lanzamos el algoritmo:

```
# Añadimos a train la característica con valor 1
c0 = np.full(len(train),1)
train = np.column_stack((c0, train))

eta = 0.01
maxIter = 10000
error2get = 1e-14
initial_point = np.array([1.0,1.0,1.0])

w = sgd(train, labels, eta, maxIter, error2get, initial_point)
```

Y mostrando el error dentro de la muestra tenemos:

```
Apartado c)

Bondad del resultado para grad. descendente estocastico:

w:  [-0.1183842  -0.97860893 -0.20084296]
Ein:  1.037876632511826
```



Fácilmente se aprecia que la aproximación es bastante mala, pues el conjunto de datos no es separable linealmente.

A continuación, repetimos los pasos anteriores 1000 veces, calculando el valor medio $\mathbf{E_{in}}$ y $\mathbf{E_{out}}$:

```
# Inicializamos los valores
eout = ein = 0

eta = 0.01
maxIter = 50
error2get = 1e-14
initial_point = np.array([1.0,1.0,1.0])

for n in range(1000):
  train = simula_unif(1000, 2, 1)
  test = simula_unif(1000, 2, 1)

  # Calculamos las etiquetas
  labels = np.array([[0,0]])
  labels = np.delete(labels, (0), axis=0)

  for (a,b) in zip(train[:,0], train [:,1]):
    labels = np.append(labels, setLabel(a,b))

  # Cambiamos el singo con un 10% de probabilidad
  for i, l in enumerate(labels):
    n = np.random.choice(2, 1, p=[0.9, 0.1])
    if n == 1:
      labels[i] = -l

  # Añadimos a train y a test la característica con valor 1
  c0 = np.full(len(train),1)
  train = np.column_stack((c0, train))
  test = np.column_stack((c0, test))

  w = sgd(train, labels, eta, maxIter, error2get, initial_point)

  ein += (Err(train, labels, w) / 1000)

  # Calculamos el error fuera de la muestra
  eout += (Err(test, labels, w) / 1000)
```

Obteniendo:

```
Apartado d)

Ein medio obtenido:  0.9482500910782989
Eout medio obtenido:  0.9482500910782989
```

Estos resultados vuelven a ser pésimos, pues como se ha dicho anteriormente estamos aplicando regresión lineal a un conjunto de datos no lineales. Se debería aplicar alguna trasformación a la muestra para poder conseguir un ajuste correcto.

3. Bonus

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

def dSecondFxy(x,y):
  return 8 * (np.pi**2) * np.cos(2*np.pi*x) * np.cos(2*np.pi*y)

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

def dSecondFyx(x,y):
  return 8 * (np.pi**2) * np.cos(2*np.pi*y) * np.cos(2*np.pi*x)
```

```python
def hessian_newton(eta, maxIter, initial_guess):
  # Variables auxiliares
  iterations = 0
  isMin = False

  # Inicializamos w
  w = initial_guess.copy()
  values = F(w[0], w[1]) # Donde guardamos los valores

  while iterations < maxIter and not isMin:
    # Construimos la matriz Hessiana
    d2fxx = dSecondFxx(w[0], w[1])
    d2fxy = dSecondFxy(w[0], w[1])
    d2fyy = dSecondFyy(w[0], w[1])
    d2fyx = dSecondFyx(w[0], w[1])


    hessian = np.empty((2,2))
    hessian[0][0] = d2fxx
    hessian[0][1] = d2fxy
    hessian[1][0] = d2fyx
    hessian[1][1] = d2fyy

    # La invertimos
    hessian = np.linalg.inv(hessian)

    # Calculamos el gradiente
    grad = gradF(w[0], w[1])

    # Actualizamos w
    old_w = w
    w = w - eta * np.dot(hessian, grad)

    # Comprobamos si hemos descendido en el valor de la función
    descent = abs(F(w[0], w[1]) - F(old_w[0], old_w[1]))
    isMin = True if descent == 0 else False

    # Guardamos el valor de la función
    values = np.vstack((values, F(w[0], w[1])))

    iterations += 1

  return w, values
```

Partiendo de la función: $f(x,y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$

Y lo calculado en el ejercicio 1:

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

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

Ahora tenemos:

$$\frac{\partial^2 f(x,y)}{\partial x^2} = 2 - \sin(2\pi y) * 8\pi^2 * \sin(2\pi x)$$

$$\frac{\partial^2 f(x,y)}{\partial y^2} = 4 - \sin(2\pi x) * 8\pi^2 * \sin(2\pi y)$$

$$\frac{\partial^2 f(x,y)}{\partial x \partial y} = 8\pi^2 * \cos(2\pi x) * \cos(2\pi y)$$

$$\frac{\partial^2 f(x,y)}{\partial y \partial x} = 8\pi^2 * \cos(2\pi y) * \cos(2\pi x)$$

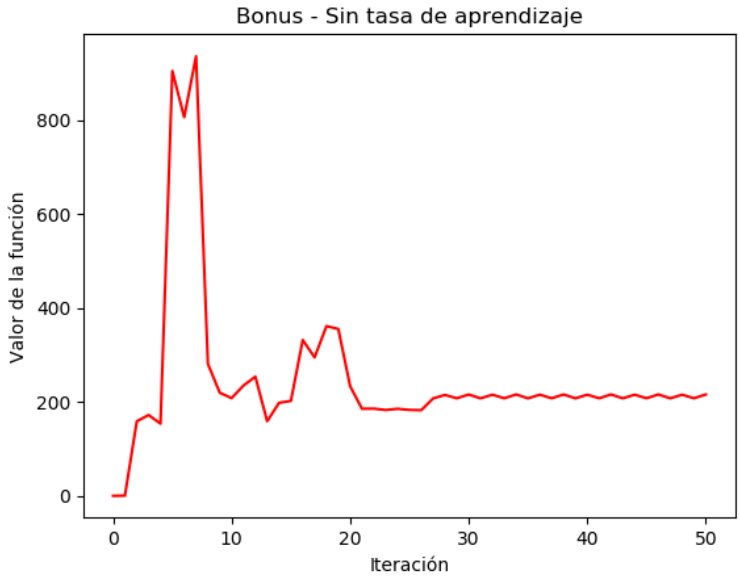
Y la matriz Hessiana:

$$\mathrm{H} = \begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial x^2} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial y \partial x} & \frac{\partial^2 f(x,y)}{\partial y^2} \end{bmatrix}$$

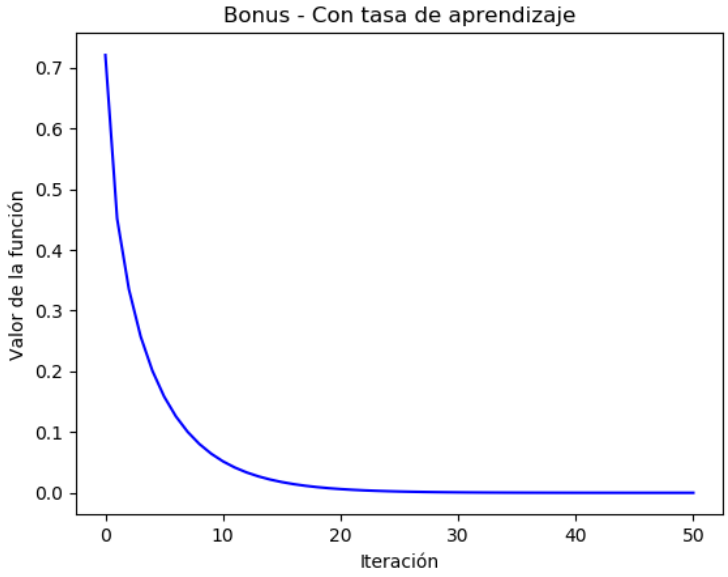
Ejecutamos:

```
maxIter = 50
initial_guess = np.array([0.1,0.1])
eta = 1

w, values = hessian_newton(eta, maxIter, initial_guess)
```

Partiendo del mismo punto inicial que el del algoritmo de gradiente descendente, obtenemos:



Vemos que el método de Newton realiza varios saltos hasta un mínimo con valor mucho mayor al encontrado con gradiente descendente. Para intentar mejorar el resultado, le añadimos una tasa de aprendizaje de 0,01:




```
Sin tasa de aprendizaje:
Coordenadas obtenidas: ( -14.773268957164783 ,  -0.23404785190162958 )
Valor final:  216.39030041026652

Con tasa de aprendizaje:
Coordenadas obtenidas: ( 0.00036805000701643054 ,  0.00036411901322880687 )
Valor final:  1.098192919541259e-05
```

Lo que mejora enormemente el funcionamiento del algoritmo.

El comportamiento del método de Newton está bastante ligado al comportamiento de la función, y es por falta de generalización la que no lo convierte en un algoritmo muy conveniente en aprendizaje automático. La problemática que genera las funciones que no son globalmente convexas y la facilidad que tiene de realizar grandes saltos lo hacen poco atractivo en nuestro campo.

Ya de por sí está claro que el Newton es computacionalmente más exigente que el gradiente descendiente, puesto que tenemos que calcular segundas derivadas de la función. Esto puede ser suficiente para no decantarnos por este método para funciones complejas. Y para problemas suficientemente simples, el cálculo de la pseudoinversa puede interesarnos aún más que el método de Newton al no ser iterativo.