Trabajo 1 - Aprendizaje automático

Francisco Solano López Rodríguez

April 16, 2018

Contents

1 Ejercicio sobre la búsqueda iterativa de óptimos

Gradiente Descendente

1. Implementar el algoritmo de gradiente descendente.

```
def gd(w, lr, grad_fun, fun, epsilon, max_iters = 100000000):
    it = 0
    # Criterio de parada: numero de iteraciones y
    # valor de f inferior a epsilon
    while it <= max_iters and fun(w) >= epsilon:
        w = w-lr*grad_fun(w)
        it += 1

    return w, it
```

2. Considerar la función $E(u,v) = (u^3e^{(v-2)} - 4v^3e^{-u})^2$ Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto $(u,v) = (1,1)$ y usando una tasa de aprendizaje $\eta = 0,05$.

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

```
def E(w):
    u = w[0]
    v = w[1]
    return (u**3*np.exp(v-2)-4*v**3*np.exp(-u))**2

# Derivada parcial de E respecto de u
def Eu(w):
    u = w[0]
    v = w[1]
    return (2*(u**3*np.exp(v-2)-4*v**3*np.exp(-u))
            *(3*u**2*np.exp(v-2)+4*v**3*np.exp(-u)))

# Derivada parcial de E respecto de v
def Ev(w):
    u = w[0]
    v = w[1]
    return (2*(u**3*np.exp(v-2)-4*v**3*np.exp(-u))
           *(u**3*np.exp(v-2)-12*v**2*np.exp(-u)))

# Gradiente de E
def gradE(w):
    return np.array([Eu(w), Ev(w)])
```

Expresión de las derivadas parciales y el gradiente:

$$\frac{\partial E}{\partial u}(u,v) = 2(u^3e^{(v-2)} - 4v^3e^{-u})(3u^2e^{(v-2)} + 4v^3e^{-u})$$

$$\frac{\partial E}{\partial v}(u,v) = 2(u^3e^{(v-2)} - 4v^3e^{-u})(u^3e^{(v-2)} - 12v^2e^{-u})$$

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

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

```
# Ejecutamos el algoritmo con un learning rate de 0.05 y
# valor de E inferior a 10^-14
w, k = gd(np.array([1.0,1.0]) , 0.05, gradE, E, 10**(-14))

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

Obtuve un valor de $E(u,v)$ inferior a 10^{-14} tras 38 iteraciones.

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

Las coordenadas en las que fue alcanzado son:

$$(u,v) = (\ 1.1195438968186378,\quad 0.6539880585437983\ )$$

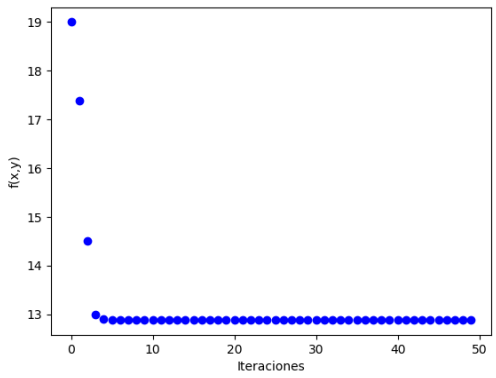
3. Considerar ahora la función $f(x,y) = (x-2)^2 + 2(y+2)^2 + 2\sin(2\pi x)\sin(2\pi y)$

```
def f(w):
    x = w[0]
    y = w[1]
    return (x-2)**2+2*(y+2)**2+2*math.sin(2*pi*x)*math.sin(2*pi*y)

# Derivada parcial de f respecto de x
def fx(w):
    x = w[0]
    y = w[1]
    return 2*(x-2)+2*math.sin(2*pi*y)*math.cos(2*pi*x)*2*pi

# Derivada parcial de f respecto de y
def fy(w):
    x = w[0]
    y = w[1]
    return 4*(y+2)+2*math.sin(2*pi*x)*math.cos(2*pi*y)*2*pi

# Gradiente de f
def gradf(w):
    return np.array([fx(w), fy(w)])
```

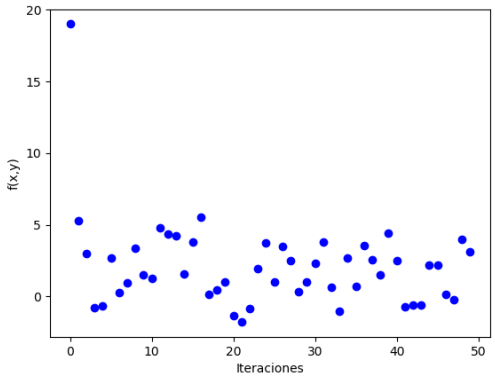
- Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = 1, y_0 = 1)$, tasa de aprendizaje $\eta = 0,01$ y un máximo de 50 iteraciones. Generar un gráfico de cómo desciende el valor de la función con las iteraciones. Repetir el experimento pero usando $\eta = 0,1$, comentar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

```
def gd_grafica(w, lr, grad_fun, fun, max_iters = 50):
    graf = [fun(w)]
    for k in range(1,max_iters):
        w = w-lr*grad_fun(w)
        graf.insert(len(graf),fun(w))
```

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

print ('Resultados ejercicio 2\n')
print ('\nGrafica con learning rate igual a 0.01')
gd_grafica(np.array([1.0,1.0]) , 0.01, gradf, f)
print ('\nGrafica con learning rate igual a 0.1')
gd_grafica(np.array([1.0,1.0]) , 0.1, gradf, f)
```



(a) $\eta = 0.01$



(b) $\eta = 0.1$

Viendo estas gráficas es mas que evidente que la convergencia del algoritmo gradiente descendente depende de la tasa de aprendizaje, en el primer caso con una tasa de aprendizaje de 0.01 vemos que se converge a un mínimo local, mientras que en el segundo caso con tasa de aprendizaje igual a 0.1 no se converge sino que va oscilando. Por lo que vemos que una elección acertada del learning rate es muy importante para asegurar la convergencia ya que si el valor es muy grande como pasa en el segundo caso puede que nunca encontremos un mínimo, por el contrario si cogemos un valor cada vez más pequeño podemos llegar a asegurar la convergencia, el problema es que si es demasiado pequeño dicha convergencia puede ser muy lenta.

- Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija: $(2, 1, -2, 1), (3, -3), (1, 5, 1, 5), (1, -1)$. Generar una tabla con los valores obtenidos.

```
def gd(w, lr, grad_fun, fun, max_iters = 50):
    for i in range(0,50):
        w = w-lr*grad_fun(w)

    return w

print ('Punto de inicio: (2.1, -2.1)\n')
w = gd(np.array([2.1, -2.1]) , 0.01, gradf, f)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

print ('Punto de inicio: (3.0, -3.0)\n')
```

```
w = gd(np.array([3.0, -3.0]) , 0.01, gradf, f)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

print ('Punto de inicio: (1.5, 1.5)\n')
w = gd(np.array([1.5, 1.5]) , 0.01, gradf, f)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

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

Punto inicio	(x,y)	f(x,y)
(2.1, -2.1)	(2.2438049693647883 , -2.237925821486178)	-1.8200785415471563
(3.0, -3.0)	(2.7309356482481055 , -2.7132791261667037)	-0.38124949743809955
(1.5, 1.5)	(1.7779244744891156 , 1.032056872669696)	18.042078009957635
(1.0, -1.0)	(1.269064351751895 , -1.2867208738332965)	-0.3812494974381

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

   La principal dificultad que nos encontramos es elegir el punto de inicio ya que este va a determinar el mínimo local al que se va a converger en caso de que haya más de un mínimo, en cuyo caso puede ser que no se converja al mínimo global. En el caso de una función convexa acotada no tenemos este problema, ya que este tipo de funciones solamente tienen un mínimo y por tanto debe ser global.

   El siguiente problema que nos encontramos es la elección del learning rate ya que con un valor alto podríamos no converger nunca. Y además en el caso de un valor demasiado pequeño podría ser una convergencia demasiado lenta.

2 Ejercicio sobre 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.

```
# 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(1)
      else:
         y.append(-1)
      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). (usar $Regress\_Lin(datos, label)$ como llamada para la función (opcional)).

   Funciones implementadas:

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

# Gradiente Descendente Estocastico
def sgd(x, y, lr, max_iters, tam_minibatch):
   w = np.zeros(len(x[0]), np.float64)
   index = np.array(range(0,x.shape[0]))

   for k in range(0,max_iters):
      w_old = w
      for j in range(0,w.size):
         suma = 0
         # Permutamos los indices para el minibatch
         np.random.shuffle(index)
         tam = tam_minibatch
         # Sumatoria en n de x_index[n]j*(w^Tx_index[n] - y_index[n])
         suma = (np.sum(x[index[0:tam:1],j]*(x[index[0:tam:1]].dot(w_old)
               - y[index[0:tam:1]])))

         w[j] -= lr*(2.0/tam)*suma

   return w

# Algoritmo pseudoinversa
def pseudoinverse(x, y):
   # Obtenemos la descomposicion en valores singulares
   u,d,vt = np.linalg.svd(x)
   d_inv = np.linalg.inv(np.diag(d))
```

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

Ejecutamos el algoritmo del Gradiente Descendente Estocástico con learning rate 0.01, número de iteraciones igual a 500 y tamaño de minibatch igual a 64. A continuación ejecutamos el algoritmo de la pseudo inversa.

```
# 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')

print ('EJERCICIO SOBRE REGRESION LINEAL\n')
print ('Ejercicio 1\n')
# Gradiente descendente estocastico

w = sgd(x, y, 0.01, 500, 64)

print ('Bondad del resultado para grad. descendente estocastico:\n')
print ("Ein: ", Err(x,y,w))
print ("Eout: ", Err(x_test, y_test, w))

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

# Para el grafico tomamos las columnas 1 y 2 de x que corresponden
# la la intensidad y simetria, y distiguimos las coordenadas por
# su valor en y, es decir por la clase a la que pertenezcan ({-1,1})
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.xlabel('Intensidad')
plt.ylabel('Simetria')
plt.title('SGD')
plt.show()

# Algoritmo Pseudoinversa

w = pseudoinverse(x, y)

print ('\nBondad del resultado para el algoritmo de la pseudoinversa:\n')
print ("Ein: ", Err(x,y,w))
print ("Eout: ", Err(x_test, y_test, w))

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

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.xlabel('Intensidad')
plt.ylabel('Simetria')
plt.title('Pseudoinversa')
plt.show()
```

Los valores obtenidos para valorar la bondad del resultado en el gradiente descendente estocástico son:

- $E_{in} = 0.08207838897626317$
- $E_{out} = 0.13696552396953135$

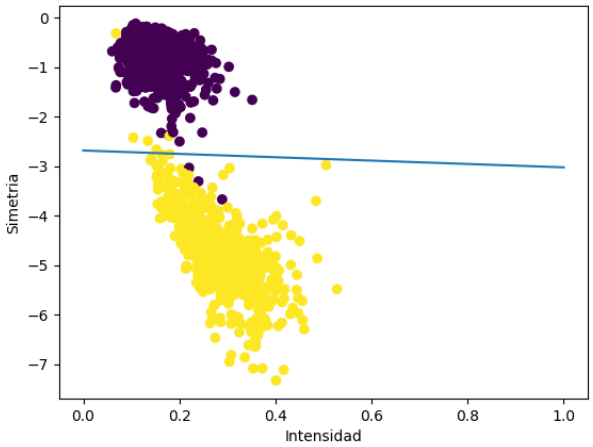


Figure 1: Gradiente Descendente Estocástico

Los valores obtenidos para valorar la bondad del resultado en el algoritmo de la pseudoinversa son:

- $E_{in} = 0.07918658628900395$
- $E_{out} = 0.13095383720052584$

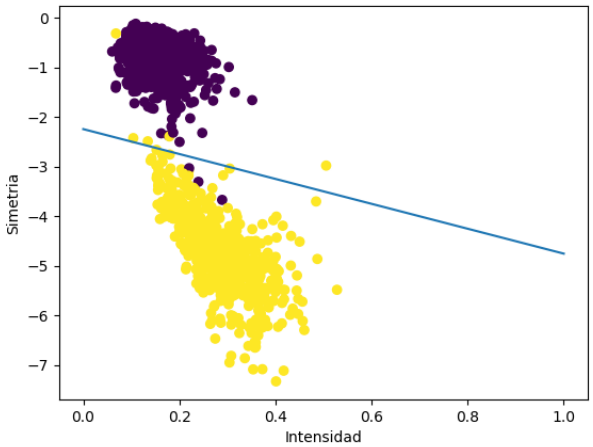


Figure 2: Pseudoinversa

Podemos ver que el mejor ajuste se consigue con el de la pseudoinversa, en donde tanto el E_{in} como el E_{out} son algo más bajos que en el SGD.

2. En este apartado exploramos como 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 $[-size, size] \times [-size, size]$

```
# Simula datos en un cuadrado [-size,size]x[-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 $= [-1, 1] \times [-1, 1]$. Pintar el mapa de puntos 2D. (ver función de ayuda)

```
x = simula_unif(1000, 2, 1)
plt.scatter(x[:,0],x[:,1])
plt.show()
```

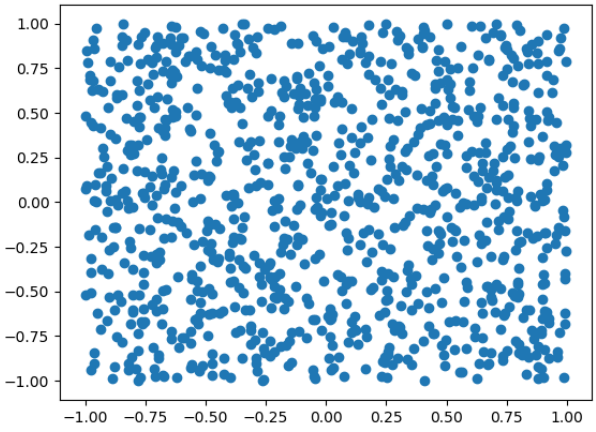


Figure 3: Muestra de entrenamiento N = 1000, [-1,1]x[-1,1]

(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.

```
# ruido al 10% de las mismas

# Funcion signo
def sign(x):
    if x >= 0:
        return 1
    return -1

def f2(x1, x2):
    return sign((x1-0.2)**2+x2**2-0.6)

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

```
j = 0
y = []

for i in range(0,x.shape[0]):
   # Si i estÃ¡ en p cambiamos el signo
   if i == p[j]:
      j = (j+1)%(x.shape[0]//10)
      y.append(-f2(x[i][0], x[i][1]))
   # En otro caso mantenemos el signo
   else:
      y.append(f2(x[i][0], x[i][1]))

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

print ('Muestra N = 1000, cuadrado [-1,1]x[-1,1]')
print ('con etiquetas y ruido en el 10% de las etiquetas')

plt.scatter(x[:,0],x[:,1], c=y)
plt.show()
```

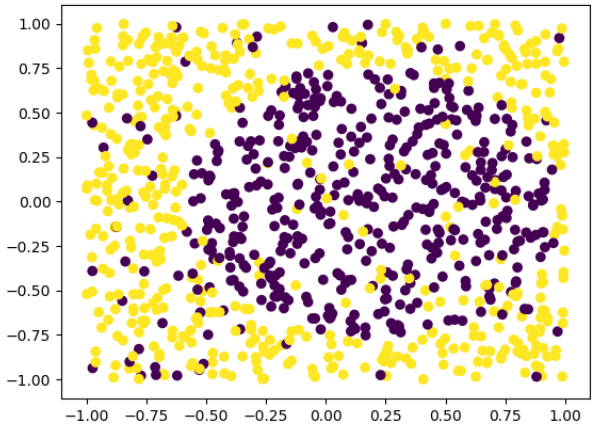


Figure 4: Muestra de entrenamiento N = 1000, [-1,1]x[-1,1] con etiquetas y ruido introducido

(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).

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

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

print ('Estimacion del error de ajuste Ein usando SGD')
print ('con 1000 iteraciones y un de minibatch de 64')
print ("\n\nEin: ", Err(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()
```

El error de ajuste E_{in} obtenido es:

$$E_{in} = 0.9565534009344919$$

El cual es un error bastante grande. Como podemos apreciar en la siguiente figura el ajuste es bastante malo.

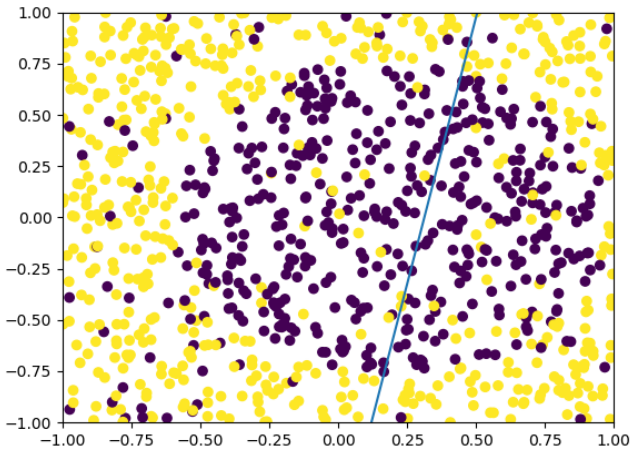


Figure 5: Recta de regresión

(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.

```
Ein_media = 0
Eout_media = 0

for k in range(0,1000):
    x = simula_unif(1000, 2, 1)
    # vector de unos para la regresion lineal (1, x0, x1)
    a = np.array([np.ones(x.shape[0], np.float64)])
    x = np.concatenate((a.T, x), axis = 1)
    y = []

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

    for i in range(0,x.shape[0]):
        if i == p[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 = sgd(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((a.T, x_test), axis = 1)

    Ein_media += Err(x,y,w)
    Eout_media += Err(x_test,y,w)

Ein_media /= 1000
Eout_media /= 1000

print ('Errores Ein y Eout medios tras 1000reps del experimento:\n')
print ("Ein media: ", Ein_media)
print ("Eout media: ", Eout_media)
```

Los errores medios obtenidos han sido los siguientes:

- E_{in} medio: 0.9901838960913539
- E_{out} medio: 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}

Evidentemente el ajuste con este modelo lineal es nefasto, solo con ver los datos representados ya intuimos que con una recta no vamos a poder hacer un buen ajuste, cosa que aseguramos tras calcular los valores medios de los errores E_{in} y E_{out}.
Sin tener que hacer cálculos solo con la vista se puede apreciar que una buena forma de realizar el ajuste seria con una circunferencia o tal vez mejor una elipse es decir una curva del tipo $h_1(x_1 - p_1)^2 + h_2(x_2 - p_2)^2 = R^2$, que precisamente es del mismo tipo con el que se realizó la asignación de etiquetas.

3 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.

```
# Segundas derivadas de f para el calculo de la hessiana

def fxx(w):
    x = w[0]
    y = w[1]
    return 2-2*math.sin(2*pi*y)*2*pi*2*pi*math.sin(2*pi*x)

def fxy(w):
    x = w[0]
    y = w[1]
    return 2*math.cos(2*pi*y)*math.cos(2*pi*x)*2*pi*2*pi

def fyy(w):
    x = w[0]
    y = w[1]
    return 4-2*math.sin(2*pi*y)*2*pi*2*pi*math.sin(2*pi*x)
```

```
# Hessiana de f
def Hf(w):
   return np.array([np.array([fxx(w), fxy(w)]), np.array([fxy(w), fyy(w)])])


# Metodo de Newton
def newtonMethods(w, lr, grad_fun, fun, H, max_iters = 50):
   graf = [fun(w)]
   for k in range(1,max_iters):
      # w_{k+1} = w_k - lr*H^-1*gradiente
      w = w - lr*np.linalg.inv(H(w)).dot(grad_fun(w))
      graf.insert(len(graf),fun(w))

   return w, graf
```

- Generar un gráfico de como desciende el valor de la función con las iteraciones.
- Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con gradiente descendente.

Realización de los experimentos que se realizaron en el ejercicio 3.

```
# Ejecutamos el metodo de Newton con punto de inicio (1,1) y lr 0.01
print ('BONUS: metodo de Newton\n')
print ('Punto de inicio: (1.0, 1.0), Learning rate: 0.01\n')
w, g = newtonMethods(np.array([1.0,1.0]), 0.01, gradf, f, Hf, 50)

# Grafica iteraciones con valores de f obtenidos en el met, Newton
plt.plot(range(0,len(g)), g, 'bo')
plt.xlabel('Iteraciones')
plt.ylabel('f(x,y)')
plt.show()

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


# Ejecutamos el metodo de Newton con punto de inicio (1,1)
# y learning rate 0.1
print ('Punto de inicio: (1.0, 1.0), Learning rate: 0.1\n')
w, g = newtonMethods(np.array([1.0,1.0]), 0.1, gradf, f, Hf)

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

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

print ('Punto de inicio: (2.1, -2.1)\n')
w, g = newtonMethods(np.array([2.1, -2.1]) , 0.01, gradf, f, Hf)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

print ('Punto de inicio: (3.0, -3.0)\n')
w, g = newtonMethods(np.array([3.0, -3.0]) , 0.01, gradf, f, Hf)
```

```
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

print ('Punto de inicio: (1.5, 1.5)\n')
w, g = newtonMethods(np.array([1.5, 1.5]) , 0.01, gradf, f, Hf)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor minimo: ',f(w))

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

print ('Punto de inicio: (1.0, -1.0)\n')
w, g = newtonMethods(np.array([1.0, -1.0]) , 0.01, gradf, f, Hf)
print ('(x,y) = (', w[0], ', ', w[1],')\n')
print ('Valor mÃnimo: ',f(w))
```

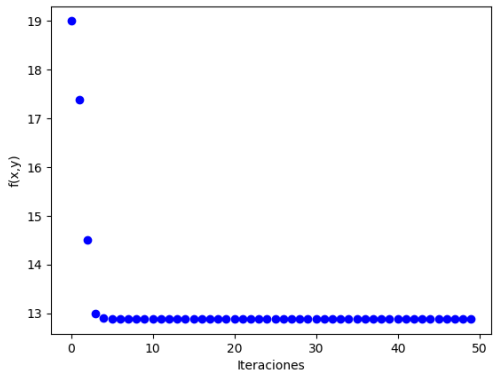
Tabla que obtuvimos con el gradiente descendente:

Punto inicio	(x,y)	f(x,y)
(2.1, -2.1)	(2.2438049693647883 , -2.237925821486178)	-1.8200785415471563
(3.0, -3.0)	(2.7309356482481055 , -2.7132791261667037)	-0.38124949743809955
(1.5, 1.5)	(1.7779244744891156 , 1.032056872669696)	18.042078009957635
(1.0, -1.0)	(1.269064351751895 , -1.2867208738332965)	-0.3812494974381

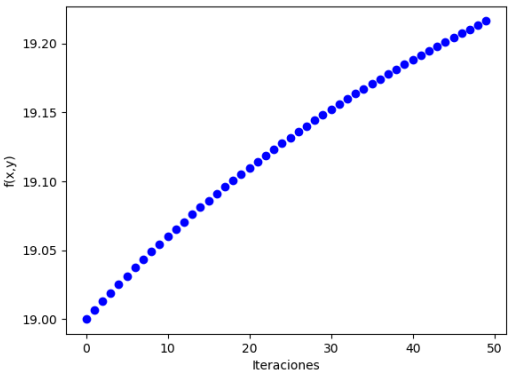
Tabla obtenida con el método de Newton:

Punto inicio	(x,y)	f(x,y)
(2.1, -2.1)	(2.0483213104654476 , -2.048629673742527)	-0.17280511826376682
(3.0, -3.0)	(3.0203273365663774 , -3.010455841088583)	3.0663860721823983
(1.5, 1.5)	(1.4282443947688035 , 1.5074740500301447)	24.890743051687227
(1.0, -1.0)	(0.9796726634336232 , -0.9895441589114156)	3.0663860721824006

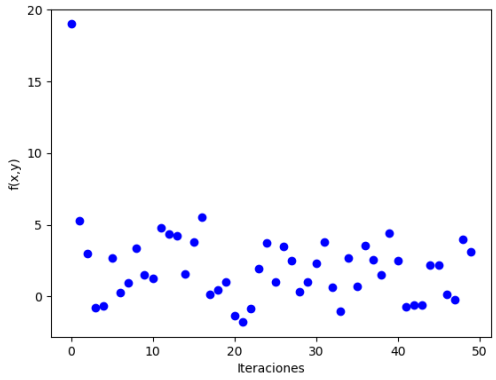
Vemos que los resultados obtenidos con el método de Newton son mucho peores a los obtenidos con gradiente descendente.Ahora pasamos a ver las gráficas comparativas de ambos algoritmos con learning rate igual a 0.01 y 0.1.



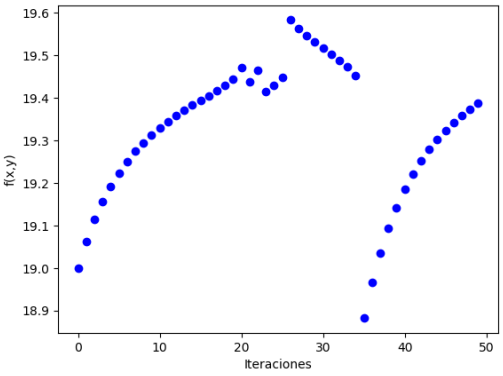
(a) Gradiente descendente $\eta = 0.01$



(b) Método de Newton $\eta = 0.01$



(c) Gradiente descendente $\eta = 0.1$



(d) Método de Newton $\eta = 0.1$

Podemos ver que ni siquiera converge en el método de Newton.

El método de Newton bajo ciertas condiciones converge más rápido que el gradiente descendente el problema es que se necesitan cumplir las condiciones de convergencia ya que el método ni siquiera garantiza la convergencia. Para asegurar la convergencia debemos de elegir un punto inicial próximo al mínimo buscado.

He realizado el experimento con el método de Newton con punto de inicio $(1.0, 1.0)$, learning rate de 0.01 y 2000 iteraciones.

```
print ('Punto de inicio: (1.0, 1.0), Learning rate: 0.01')
print ('2000 iteraciones')
w, g = newtonMethods(np.array([1.0,1.0]), 0.01, gradf, f, Hf, 2000)

# Grafica iteraciones con valores de f obtenidos en el met, Newton
plt.plot(range(0,len(g)), g, 'bo')
plt.xlabel('Iteraciones')
plt.ylabel('f(x,y)')
plt.show()
```

La gráfica obtenida es la siguiente:

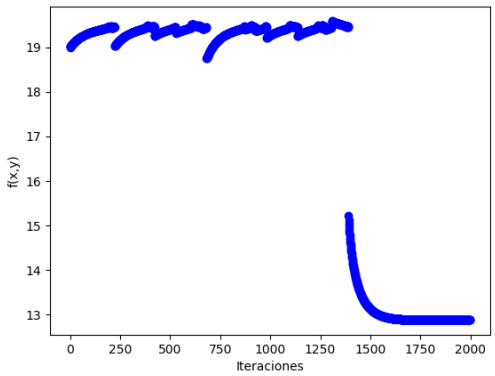


Figure 6: Método de Newton 2000 iteraciones

Vemos que al principio no converge y oscila mucho, pero al final acaba convergiendo. Esto puede ser debido a que no se encontraba cerca de ningún mínimo, pero después de muchas iteraciones ha llegado a un sitio con un mínimo cercano y a partir de ese momento a convergido rapidamente hacia él.