

# Doble Grado en Ingeniería Informática y Matemáticas

APRENDIZAJE AUTOMÁTICO  
(E. Computación y Sistemas Inteligentes)

## **TRABAJO-1: Programación**



**UNIVERSIDAD  
DE GRANADA**

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 3 - Lunes

*Email:* carlossamu7@correo.ugr.es

*7 de marzo de 2020*

# Índice

<b>1. Ejercicio sobre la búsqueda iterativa de óptimos</b>	<b>2</b>
Apartado 1 . . . . .	2
Apartado 2 . . . . .	3
Apartado 3 . . . . .	4
Apartado 4 . . . . .	7
<b>2. Ejercicio sobre Regresión Lineal</b>	<b>8</b>
Apartado 1 . . . . .	8
Apartado 2 . . . . .	10
<b>3. Bonus</b>	<b>13</b>

# 1. Ejercicio sobre la búsqueda iterativa de óptimos

**Gradiente Descendente.**

## Apartado 1

**Implementar el algoritmo de gradiente descendente.**

Comenzamos con la implementación de la función gradiente descendente, `gd`, y para ello es fundamental entender a la perfección los diferentes parámetros de dicha función:

- `w`: Punto inicial.
- `lr`: *Learning rate* o tasa de aprendizaje.
- `grad_fun`: Gradiente de la función `fun`.
- `fun`: Función (diferenciable) a minimizar.
- `epsilon`: Error permitido. Si el valor de la función está por debajo de éste entonces acabamos y no hace falta agotar `max_iters`.
- `max_iters`: Número máximo de iteraciones.

Los argumentos `epsilon` y `max_iters` son opcionales, es decir, si no le pasamos un valor tomarán el valor por defecto  $-\infty$  y 100000 respectivamente. Mostramos el código del algoritmo:

```
def gd(w, lr, grad_fun, fun, epsilon=-math.inf, max_iters=100000):  
    it = 0  
    while fun(w) > epsilon and it < max_iters:  
        w = w - lr * grad_fun(w)  
        it += 1  
    return w, it
```

La expresión que rige la minimización de gradiente descendente es  $w = w - \eta \nabla E(w)$ . Por tanto los dos ingredientes más importantes para que la minimización sea efectiva es una buena elección de la tasa de aprendizaje ( $\eta$ ), ya que un valor muy grande podría hacer que el algoritmo no convergiese y un valor muy pequeño podría necesitar demasiadas iteraciones para alcanzar el mínimo, y un buen punto inicial. Por supuesto el cálculo del gradiente ha de ser correcto, en otro caso, es como si no hiciéramos nada.

La condición de parada que hemos impuesto es que no se sobrepase el número máximo de iteraciones o que el valor de la función en un punto sea menor o igual que `epsilon`. La función nos devuelve el vector final donde se alcanza el mínimo y el número de iteraciones que se han usado.

## Apartado 2

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

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

Calculamos las derivadas parciales de  $E$  respecto a  $u$  y  $v$  para formar la expresión del gradiente  $\nabla E$ :

$$\nabla E(u, v) = \left( \frac{\partial E}{\partial u}, \frac{\partial E}{\partial v} \right) = (2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}), 2(ue^v - 2ve^{-u})(ue^v - 2e^{-u}))$$

Mostramos el código de la función  $E$ , sus derivadas parciales y  $\nabla E$ :

```
""" Funcion E del apartado 2 """
def E(w):
    return (w[0]*np.exp(w[1]) - 2*w[1]*np.exp(-w[0]))**2

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

""" Derivada parcial de E respecto de v """
def Ev(w):
    return 2 * (w[0]*np.exp(w[1]) - 2*w[1]*np.exp(-w[0]))
    * (w[0]*np.exp(w[1]) - 2*np.exp(-w[0]))

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

- 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).

Para saber el número de iteraciones que tarda el algoritmo, llamamos a la función `gd` implementada, usando  $(u, v) = (1, 1)$  y  $\eta = 0,1$ .

```
w, num_ite = gd(np.array([1.0, 1.0]), 0.1, gradE, E, 1e-14)
print("\nb) Numero de iteraciones: {}".format(num_ite))
```

El número de iteraciones necesarias para alcanzar un error menor que  $10^{-14}$  es 10.

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

```
print("\nc) Coordenadas obtenidas: ({}, {})".format(w[0], w[1]))
print("    Valor en el punto: {}".format(E(w)))
```

Las coordenadas del mínimo son  $(u_{min}, v_{min}) \approx (0,044736, 0,023959)$  y su imagen  $E(u_{min}, v_{min}) \approx 1,208683 \cdot 10^{-15}$  (he redondeado a 6 decimales).

### Apartado 3

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

- a) 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  $\eta$ .

De manera análoga al ejercicio anterior calculamos las derivadas parciales de  $f$  respecto a  $x$  e  $y$  para formar la expresión del gradiente  $\nabla f$ :

$$\nabla f(x, y) = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) = (2(x - 2) + 4\pi \cos(2\pi x) \sin(2\pi y), 4(y + 2) + 4\pi \sin(2\pi x) \cos(2\pi y))$$

Continuamos mostrando el código asociado a la función  $f$ , sus derivadas parciales y  $\nabla f$ :

```
""" Funcion f del apartado 3 """
def f(w):
    return (w[0]-2)**2 + 2*(w[1]+2)**2 +
           2*np.sin(2*np.pi*w[0])*np.sin(2*np.pi*w[1])

""" Derivada parcial de f respecto de x """
def fx(w):
    return 2*(w[0]-2) + 4*np.pi*np.cos(2*np.pi*w[0])*np.sin(2*np.pi*w[1])

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

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

Para el apartado a) se ha implementado una función llamada `gd_grafica` que aplica gradiente descendente pero va guardando el valor de la función en cada iteración para posteriormente construir una gráfica:

```
""" Funcion de GD que almacena los resultados para construir una grafica """
def gd_grafica(w, lr, grad_fun, fun, max_iters = 100000):
    it = 0
    graf = np.zeros(max_iters)
    while it < max_iters:
        graf[it] = fun(w) # Guardamos el resultado de la iteración
        w = w - lr*gradf(w)
        it += 1

    # Dibujamos la gráfica
    plt.plot(range(0, max_iters), graf, 'b-o', label=r'$\eta$ = {}'.format(lr))
    plt.xlabel('Iteraciones'); plt.ylabel('f(x,y)')
    plt.legend()
    plt.title("Curva de gradiente descendente")
    plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 3a')
    plt.show()
    return graf
```

Vamos a preparar un código que construya esas dos gráficas usando como punto inicial  $(x_0, y_0) = (1, -1)$  y tasas de aprendizaje  $\eta_1 = 0,01$  y  $\eta_2 = 0,1$ . Posteriormente juntaremos ambas gráficas sobre los mismos ejes coordenados para comparar mejor.

```
print ("a) Grafica con learning rate igual a 0.01")
g1 = gd_grafica(np.array([1.0, -1.0]), 0.01, gradf, f, 50)
print ("    Grafica con learning rate igual a 0.1")
g2 = gd_grafica(np.array([1.0, -1.0]), 0.1, gradf, f, 50)
# Comparamos las gráficas
plt.plot(g1, 'b-o', label=r"$\eta$ = {}".format(0.01))
plt.plot(g2, 'r-o', label=r"$\eta$ = {}".format(0.1))
plt.xlabel('Iteraciones'); plt.ylabel('f(x,y)');
plt.legend()
plt.title("Comparacion de las curvas de gradiente descendente")
plt.gcf().canvas.set_window_title('Ejercicio 1 - Apartado 3a')
plt.show()
```

Obtenemos las siguientes graficas para  $\eta_1$  y  $\eta_2$ :

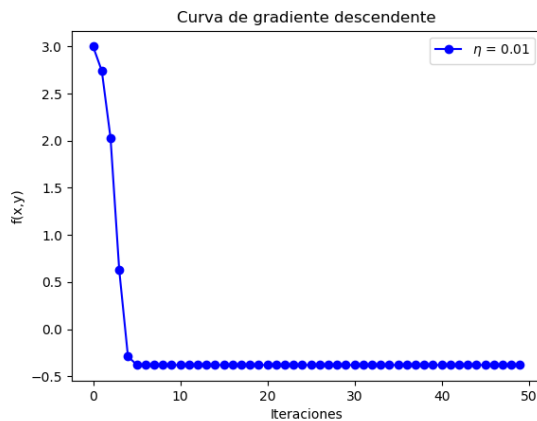


Imagen 1: Curva de GD con  $\eta = 0,01$ .

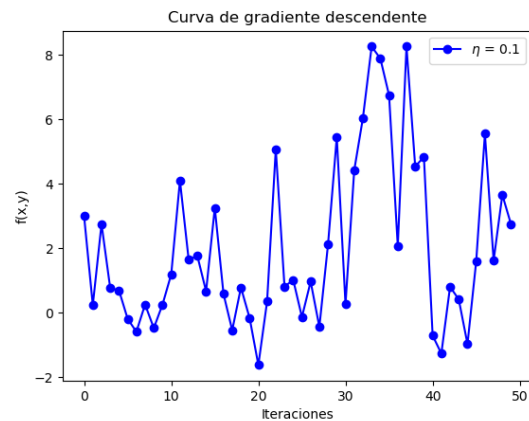


Imagen 2: Curva de GD con  $\eta = 0,1$ .

En la primera como  $\eta_1$  es pequeño se realiza una buena convergencia y en la iteración 10 ya podría haberse parado el algoritmo. En la segunda tenemos una tasa de aprendizaje  $\eta_2$  demasiado grande que impide la convergencia hacia el mínimo y no para de saltar de un lado a otro. Sin duda usar una tasa de aprendizaje como  $\eta_2$  para esta función y valor inicial es un error.

Se muestran a continuación ambas gráficas en la misma imagen para comparar algo mejor y entender el papel tan importante que juega la tasa de aprendizaje en el algoritmo de gradiente descendente.

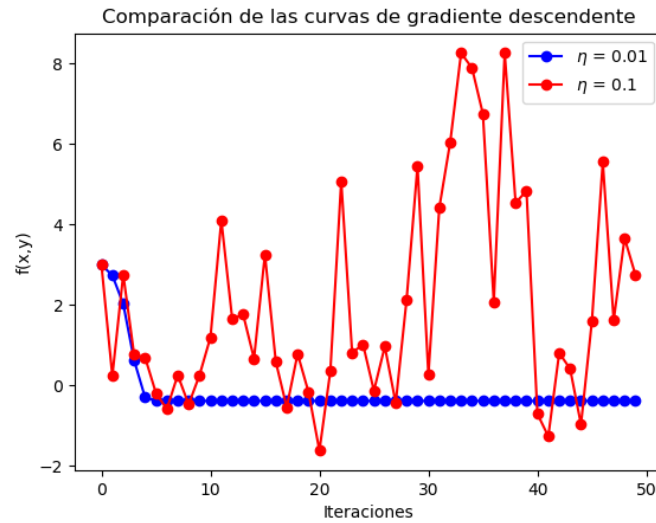


Imagen 3: Comparativa de curvas de GD.

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

He implementado una función `print_gd` que haciendo uso de gradiente descendente imprimirá los datos pedidos (punto inicial, mínimo y valor del mínimo).

```
""" Usando GD muestra el punto inicial, el minimo y el valor del minimo """
def print_gd(w, lr, grad_fun, fun, epsilon, max_iters = 1000000000):
    print("\n Punto de inicio: ({} , {})".format(w[0], w[1]))
    w, _ = gd(w, lr, grad_fun, fun, epsilon, max_iters)
    print(" (x,y) = ({} , {})".format(w[0], w[1]))
    print(" Valor minimo: {}".format(f(w)))
```

Por tanto, para la resolución del apartado sólo tengo que llamar a esta función cuatro veces, una por cada punto que me dan. En este caso la tasa de aprendizaje es  $\eta = 0,01$ . He establecido como condiciones de parada un número máximo de 1000 iteraciones o un error menos que 0,0001.

```
print("\nb) Minimo y valor donde se alcanza segun el punto inicial:")
lrate = 0.01; eps = 0.0001; max_it = 1000
print_gd(np.array([2.1, -2.1]), lrate, gradf, f, eps, max_it)
print_gd(np.array([3.0, -3.0]), lrate, gradf, f, eps, max_it)
print_gd(np.array([1.5, 1.5]), lrate, gradf, f, eps, max_it)
print_gd(np.array([1.0, -1.0]), lrate, gradf, f, eps, max_it)
```

Tal y como se nos pide, generamos la tabla con los resultados pedidos.

$(x_0, y_0)$	$(x_{min}, y_{min})$	$E(x_{min}, y_{min})$
(2.1, -2.1)	(2.1, -2.1)	-0.660983
(3, -3)	(2.766339, -2.745475)	-0.289924
(1.5, 1.5)	(1.779120, 1.0309456)	18.042072
(-1, -1)	(1.233661, -1.254525)	-0.2899238

Tabla 1: Valores iniciales, mínimos alcanzados para Gradiente Descendente.

He redondeado los valores a 6 decimales. En primer lugar, recordar que estamos usando la tasa de aprendizaje “buena”  $\eta = 0,01$  y aún así los resultados varían mucho. Esta batería de ejemplos refleja perfectamente la dependencia de este algoritmo del punto inicial para caer en óptimos locales y no converger a los globales.

Hay que destacar de igual manera que el peor punto inicial es el  $(1,5, 1,5)$  ya que el mínimo es muy malo comparado con los otros y reflexionar por qué cuándo el punto inicial es  $(2,1, -2,1)$  el mínimo es él mismo. La respuesta es sencilla: la primera parte es  $(x-2)^2$  que para ese  $x$  es prácticamente 0, lo mismo pasa con  $2(y+2)^2$  y por último la parte trigonométrica que al ser tanto  $x$  como  $y$  valores cercanos a un entero estamos ante casi un múltiplo de  $2\pi$  en donde la función seno es 0. Así, ese punto ya es un buen candidato a ser el mínimo.

#### Apartado 4

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

El algoritmo de gradiente descendente es una técnica útil e intuitiva para buscar mínimos de funciones. La primera condición necesaria es la derivabilidad de la función a minimizar. La búsqueda del mínimo global de una función nos dificulta mucho las cosas. Cuando sabemos que la función es convexa entonces tenemos asegurado que el mínimo que encontremos es un mínimo global e incluso si buscamos un mínimo en un cierto intervalo y sabemos que la función en ese intervalo es convexa (además de diferenciable) también tenemos la certeza de encontrar el mínimo global.

Las funciones ejemplificadas en este ejercicio,  $f$  y  $E$ , no son convexas. Analizando los resultados he de destacar que ambas funciones son muy diferentes ya que  $f$  tiene muchos mínimos locales debido a la presencia de funciones sin en su expresión lo cual hace que la elección del punto inicial sea determinante en la obtención del resultado. Para  $E$  decir que la convergencia al mínimo es rápida (pocas iteraciones), no sabemos si ese mínimo es global pero debido al cuadrado en la expresión de  $E$  deducimos  $E(u, v) \geq 0$  y el valor obtenido en el apartado 2c) de este ejercicio es prácticamente 0.

Por tanto, después de lo explicado concluyo diciendo que la dificultad está en la elección de parámetros como la tasa de aprendizaje y el punto inicial y para hacer una buena elección podemos hacer múltiples pruebas además de pararnos y analizar la función en caso de que no sea excesivamente compleja. Con estas herramientas deberíamos de ser capaces de no quedar atrapados en mínimos locales y tener una velocidad de convergencia adecuada.



## 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.

### Apartado 1

Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números (Intensidad promedio, Simetria) usando tanto el algoritmo de la pseudo-inversa 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)).

Escribir

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

""" Derivada del error """
def dErr(x, y, w):
    return 2/len(x)*(x.T.dot(x.dot(w) - y))
```

Escribir

```
""" Gradiente Descendente Estocastico.
- x: datos en coordenadas éhomogneas.
- y: etiquetas asociadas {-1,1}.
- lr: tasa de aprendizaje.
- max_iters: número máximo de iteraciones.
- tam_minibatch: tamaño del minibatch. """
def sgd(x, y, lr, max_iters, tam_minibatch):
    w = np.zeros(3)
    it = 0
    ind_set = np.random.permutation(np.arange(len(x))) # conjunto de índices
    begin = 0 # Comienzo de la muestra

    while it < max_iters:
        if begin > len(x): # Nueva época
            begin = 0
            ind_set = np.random.permutation(ind_set)

        minibatch = ind_set[begin:begin + tam_minibatch] # Escogemos el minibatch
        w = w - lr*dErr(x[minibatch, :], y[minibatch], w)
        it += 1 # Actualizo iteraciones
        begin += tam_minibatch # Cambio minibatch
    return w
```

Escribir

```
""" Calcula w usando el algoritmo de la pseudoinversa """  
def pseudoinverse(x, y):  
    x_pinv = np.linalg.inv(x.T.dot(x)).dot(x.T) # inv(xT * x) * xT  
    return np.dot(x_pinv, y) # w = pseudoinversa(x) * y
```

Escribir

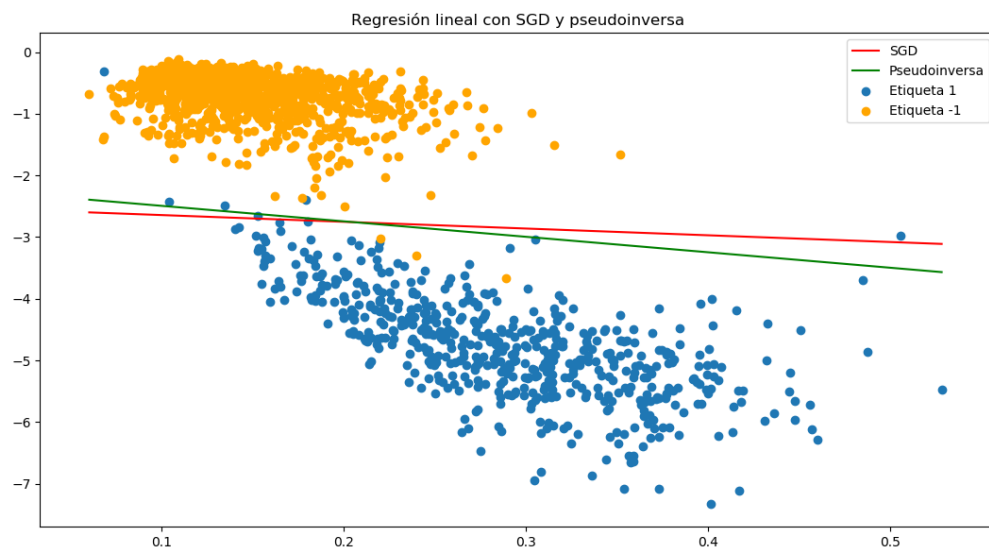


Imagen 4: Comparativa de curvas de GD.

## Apartado 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  $[-size, size] \times [-size, size]$ .

### ■ EXPERIMENTO

- a) Generar una muestra de entrenamiento de  $N = 1000$  puntos en el cuadrado  $\mathcal{X} = [-1, 1] \times [-1, 1]$ . Pintar el mapa de puntos 2D. (ver función de ayuda)

Escribir.

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

print ("\n### Apartado 2 ###\n")
# a) Muestra de entrenamiento N = 1000, cuadrado [-1,1]x[-1,1]
print ("a) Muestra N = 1000, cuadrado [-1,1]x[-1,1]")
x = simula_unif(1000, 2, 1)
plt.scatter(x[:,0], x[:,1])
plt.title("Muestra de entrenamiento N = 1000, cuadrado [-1,1]x[-1,1]")
plt.gcf().canvas.set_window_title('Ejercicio 2 - Apartado 2a')
plt.show()
```

Escribir.

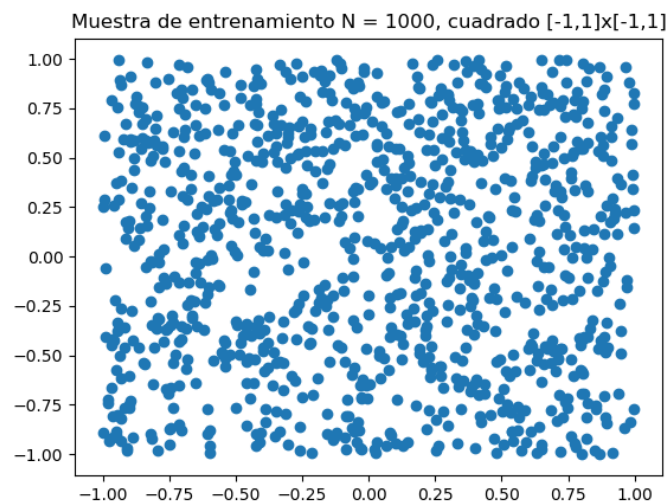


Imagen 5: Muestra de 1000 puntos en  $[-1, 1] \times [-1, 1]$ .

- b) Consideremos la función  $f(x_1, x_2) = \text{sign}((x_1 - 0,2)^2 + x_2^2 - 0,6)$  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.

Escribir.

```
""" Calcula la ófuncin f dada sobre dos vectores x1 y x2.
    Introduce ruido al 10% de los datos (cambia las etiquetas). """
def f(x1, x2):
    # Calulamos la predicción de todos los puntos 2D
    res = np.sign((x1 - 0.2)**2 + x2**2 - 0.6)
    # Introducimos ruido en el 10%
    ind = np.random.choice(len(res), size=int(len(res)/10), replace=False)
    for i in ind:
        res[i] = -res[i]
    return res

# b) Mapa de etiquetas usando la función f y con un 10 % de ruido
print ("\nb) Mapa de etiquetas")
y = f(x[:,0], x[:,1])
plt.scatter(x[y==1][:,0], x[y==1][:,1], label="Etiqueta 1")
plt.scatter(x[y==-1][:,0], x[y==-1][:,1], c='orange', label="Etiqueta -1")
plt.legend()
plt.title("Mapa de etiquetas sobre la muestra en [-1,1]x[-1,1]")
plt.gcf().canvas.set_window_title('Ejercicio 2 - Apartado 2b')
plt.show()
```

Escribir.

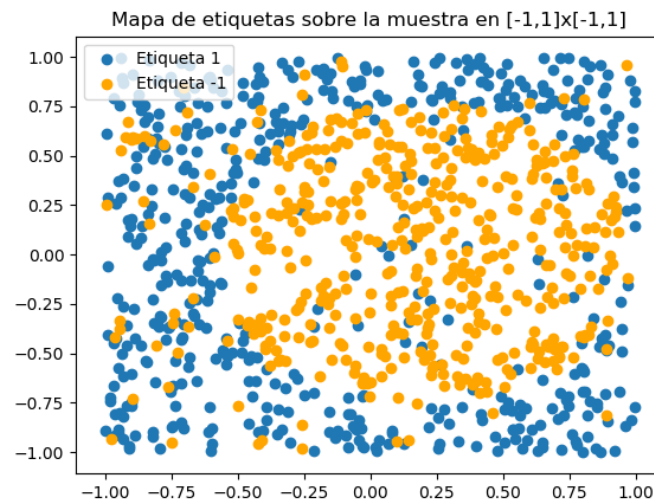


Imagen 6: Mapa de etiquetas de los 1000 puntos con un 10 % de ruido.

- 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).

Escribir.

```
# c) Usando (1, x1, x2) ajustar un modelo de regresion lineal al conjunto de
# datos y estimar los pesos w. Estimar el error de ajuste Ein usando SGD.
x_ones = np.hstack((np.ones((1000, 1)), x)) # columna de 1s
w = sgd(x_ones, y, 0.01, 1000, 32)
print("\nc) Bondad del resultado para SGD:")
print("    Ein:  {}".format(Err(x_ones, y, w)))
input("—— Pulsar tecla para continuar ——")
```

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.

Escribir.

```
""" Experemiento a ejecutar 1000 veces """
def experiment():
    x = np.hstack((np.ones((1000, 1)), simula_unif(1000, 2, 1)))
    y = f(x[:,0], x[:,1])
    x_test = np.hstack((np.ones((1000, 1)), simula_unif(1000, 2, 1)))
    y_test = f(x_test[:,0], x_test[:,1])
    w = sgd(x, y, 0.01, 1000, 32)
    Ein = Err(x, y, w)
    Eout = Err(x_test, y_test, w)
    return np.array([Ein, Eout])
```

Escribir.

```
# d) Ejecutar el experimento 1000 veces
print ("\nd) Errores Ein y Eout medios tras 1000reps del experimento:")
N = 1000; errs = np.array([0.,0.])
for _ in range(N):
    errs = errs + experiment()
Ein_media, Eout_media = errs/N
print ("    Ein media: ", Ein_media)
print ("    Eout media: ", Eout_media)
input("—— Pulsar tecla para continuar ——\n")
```

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  $E_{out}$ .

Escribir.

### 3. Bonus

**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.

- Generar un gráfico de como descende el valor de la función con las iteraciones.

Escribir.

```
""" Hessiana de f """
def hessf(w):
    return np.array([
        2 - 8*np.pi**2*np.sin(2*np.pi*w[0])*np.sin(2*np.pi*w[1]),
        8*np.pi**2*np.cos(2*np.pi*w[0])*np.cos(2*np.pi*w[1]),
        8*np.pi**2*np.cos(2*np.pi*w[0])*np.cos(2*np.pi*w[1]),
        4 - 8*np.pi**2*np.sin(2*np.pi*w[0])*np.sin(2*np.pi*w[1])
    ]).reshape((2, 2))
```

Escribir.

```
""" Newton. Devuelve el ínnimo el °n de iteraciones usadas.
- w: vector de pesos inicial.
- lr: tasa de aprendizaje.
- grad_fun: gradiente de 'fun'.
- fun: ófuncin (diferenciable) a minimizar.
- hess_fun: ófuncin hessiana.
- max_iters: ámximo únmero de iteraciones.
"""
def newton(w, lr, grad_fun, fun, hess_fun, max_iters = 100000):
    w_list = [w]
    it = 0

    while it < max_iters:
        w = w - lr*np.linalg.inv(hess_fun(w)).dot(grad_fun(w))
        w_list.append(w)
        it += 1

    return np.array(w_list)
```

Escribir.

```
# Representación de curva de decrecimiento para Newton
print("\nCurva de decrecimiento usando Newton")
g3 = np.apply_along_axis(f, 1, newton(np.array([1.0, -1.0]), 0.01, gradf, f, hessf, 50))
g4 = np.apply_along_axis(f, 1, newton(np.array([1.0, -1.0]), 0.1, gradf, f, hessf, 50))
plt.plot(g3, 'g-o', label=r"Newton, $\eta$ = 0.01")
plt.plot(g4, 'c-o', label=r"Newton, $\eta$ = 0.1")
plt.legend()
plt.title("Curva de decrecimiento usando Newton")
plt.gcf().canvas.set_window_title('Bonus')
plt.show()
input("—— Pulsar tecla para continuar ——")
```

Escribir.

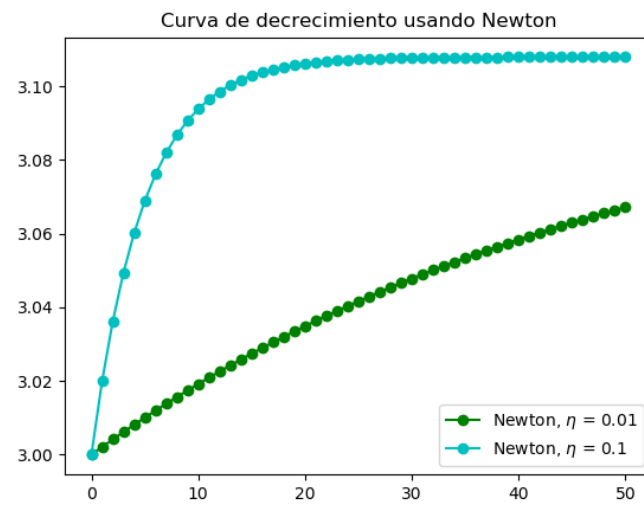


Imagen 7: Curva de Newton

- 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.

Escribir.

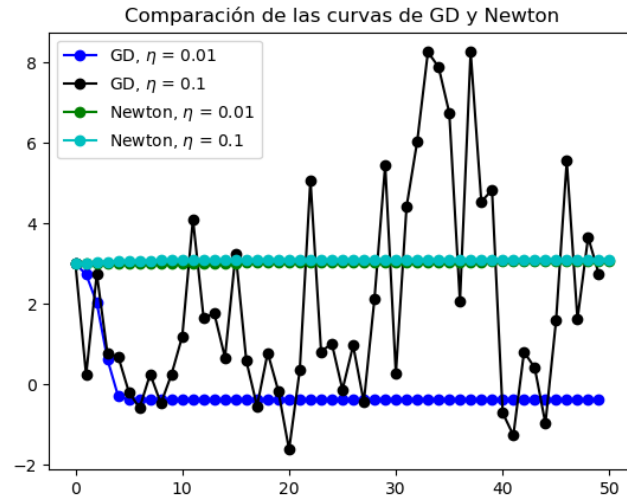


Imagen 8: Comparación de las curvas de Gradiente Descendente y Newton.

Escribir.

$(x_0, y_0)$	$(x_{min}, y_{min})$	$E(x_{min}, y_{min})$
(2.1, -2.1)	(2.1, -2.1)	-0.660983
(3, -3)	(2.766339, -2.745475)	-0.289924
(1.5, 1.5)	(1.779120, 1.0309456)	18.042072
(-1, -1)	(1.233661, -1.254525)	-0.2899238

Tabla 2: Valores iniciales, mínimos alcanzados para Newton.