UNIVERSIDAD
DE GRANADA

Aprendizaje Automático
grado en ingeniería informática

MEMORIA PRÁCTICA 1

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

Curso 2018-2019

Índice

1. Ejercicio sobre la búsqueda iterativa de óptimos

Apartado 1

Implementar el algoritmo de gradiente descendente.

A continuación se muestra el código en Python implementado:

In [3]:
```python
def descent_gradient(initial_w, function, gradient, eta=0.01, threshold=None, iterations=100):
    """
    Función para el cálculo del gradiente descendente

    :param initial_w: Pesos iniciales
    :param function: Función a evaluar
    :param gradient: Función gradiente a utilizar
    :param eta: Valor de la tasa de aprendizaje (por defecto 0.01)
    :param threshold: Valor umbral con el que parar (por defecto None)
    :param iterations: Número máximo de iteraciones que tiene que hacer el bucle
                       (por defecto 100)

    :returns: Devuelve el peso final (w), el número de iteraciones que ha llevado
              conseguir llegar hasta éste, un array con todos los w y un array con
              los valores de w evaluados en function
    """

    w = np.copy(initial_w)                  # Se copia initial_w para evitar modificarlo
    iter = 0                                # Se inicializan las iteraciones a 0
    w_list = []                             # Se inicializa una lista vacía con los valors de w
    func_values_list = []                   # Se inicializa una lista vacía con los valors de la función

    w_list.append(w)                        # Añadir valor inicial de w
    func_values_list.append(function(*w))   # Añadir valor inicial de w evaluado en function

    # Se realiza el cálculo de la gradiente descendente mientras no se superen
    # el número máximo de iteraciones.
    while iter < iterations:
        iter += 1
        w = w - eta * gradient(*w)              # Actualización de w con los nuevos valores

        w_list.append(w)                        # Añadir nuevo w
        func_values_list.append(function(*w))   # Añadir nueva evaluación de w en function

        # Si se ha especificado un umbral en el que parar y se ha pasado
        # se sale del bucle
        if threshold and function(*w) < threshold:
            break

    return w, iter, np.array(w_list), np.array(func_values_list)
```

Se ha intentado que esta implementación sea lo más general posible para poder utilizarla en los ejercicios posteriores, parametrizando la función que recibe (parámetro **function**), la cuál puede ser tanto $f(x, y)$ como $E(u, v)$. Con este motivo, también se ha parametrizado el error con el que se quiere ajustar, ya que en un caso no será necesario utilizar un error como criterio de parada (de ahí que su valor por defecto sea **None**). Y, adicionalmente, se ha parametrizado el gradiente (parámetro **gradient**), para que también se pueda especificar a la hora de la llamada cuál se usará.

Apartado 2

Considerar la función $E(u, v) = (u^2e^v - 2v^2e^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta funcón, comenzando desde el punto $(u, v) = (1, 1)$ y usanto una tasa de aprendizaje $\eta = 0{,}01$.

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

Para calcular el gradiente, vamos a calcular antes $\frac{\partial E}{\partial u}$ y $\frac{\partial E}{\partial v}$. Las derivadas, al aplicar la regla de la cadena, quedarían de la siguiente forma:

$$\frac{\partial E}{\partial u} = \frac{\partial}{\partial u}\Big((u^2e^v - 2v^2e^{-u})^2\Big) = 2(u^2e^v - 2v^2e^{-u})\frac{\partial(u^2e^v - 2v^2e^{-u})}{\partial u} = \\ = 2(u^2e^v - 2v^2e^{-u})(2ue^v + 2v^2e^{-u}) \tag{1}$$

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

Con esto, tenemos que la expresión del gradiente es la siguiente:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial u} \\ \\ \frac{\partial E}{\partial v} \end{bmatrix} \tag{3}$$

$$\nabla E = \begin{bmatrix} 2(u^2e^v - 2v^2e^{-u})(2ue^v + 2v^2e^{-u}) \\ \\ 2(u^2e^v - 2v^2e^{-u})(u^2e^v - 4ve^{-u}) \end{bmatrix} \tag{4}$$

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)

```
In [5]: # Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
        # (w inicial, num. iteraciones, valor mínimo)
        initial_w = np.array([1.0, 1.0])
        max_iter = 10000000000
        error = 1e-14

        w, it, w_array, func_val = descent_gradient(initial_w, E, gradient_E, threshold=error, iterations=max_iter)

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

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

Figura 1: Cálculo del mínimo mediante el Gradiente Descendente para la función $E(u, v)$.

Como se puede ver en la figura anterior, donde también se incluye el código, el algoritmo tarda 33 iteraciones en encontrar por primera vez un valor de $E(u, v)$ inferior a 10^{-14}.

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 donde se alcanzó un valor inferior a 10^{-14} son $(0{,}619, 0{,}968)$ (redondeadas a 3 cifras decimales).

Apartado 3

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

a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = 0{,}1, y_0 = 0{,}1)$, tasa de aprendizaje $\eta = 0{,}01$ y un máximo de 50 iteraciones. Repetir el experimento pero usando $\eta = 0{,}1$, comentar las diferencias y su dependencia de η.

Antes de mostrar las gráficas, vamos a calcular el gradiente de la función $f(x, y)$. Primero, vamos a calcular $\frac{\partial f}{\partial x}$ y $\frac{\partial f}{\partial y}$:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 2x + 4\pi\cos(2\pi x)\sin(2\pi y) \tag{5}$$

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 4y + 4\pi\sin(2\pi x)\cos(2\pi y) \tag{6}$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{7}$$

$$\nabla f = \begin{bmatrix} 2x + 4\pi\cos(2\pi x)\sin(2\pi y) \\ \\ 4y + 4\pi\sin(2\pi x)\cos(2\pi y) \end{bmatrix} \tag{8}$$

Una vez calculada la expresión del gradiente, vamos a ejecutar el código y a realizar la comparación de los cálculos del gradiente al cambiar el valor de η:

In [6]:
```
# Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
# en los dos casos
# w inicial, eta del segundo caso a estudiar y número máximo de iteraciones
initial_w = np.array([0.1, 0.1])
eta = 0.1
max_iter = 50

# Primer caso: w_inicial = (0.1, 0.1), eta 0.01, iteraciones = 50
w_1, it_1, w_array_1, func_val_1 = descent_gradient(initial_w, f, gradient_f, iterations=max_iter)

# Segundo caso: w_inicial = (0.1, 0.1), eta 0.1, iteraciones = 50
w_2, it_2, w_array_2, func_val_2 = descent_gradient(initial_w, f, gradient_f, eta=eta, iterations=max_iter)

# Mostrar por pantalla los resultados obtenidos
print('eta = 0.01')
print('Coordenadas obtenidas = ({}, {})'.format(w_1[0], w_1[1]))
print('Valor de la función = {}\n'.format(func_val_1[-1]))

print('eta = 0.1')
print('Coordenadas obtenidas = ({}, {})'.format(w_2[0], w_2[1]))
print('Valor de la función = {}'.format(func_val_2[-1]))
```

```
eta = 0.01
Coordenadas obtenidas = (0.24380496936478835, -0.23792582148617766)
Valor de la función = -1.8200785415471563

eta = 0.1
Coordenadas obtenidas = (0.10039167365942725, -1.0157510051441512)
Valor de la función = 1.9570333596159941
```

Figura 2: Valores de los óptimos para la función $f(x, y)$ cambiando η-

A continuación, se muestra el gráfico comparativo:

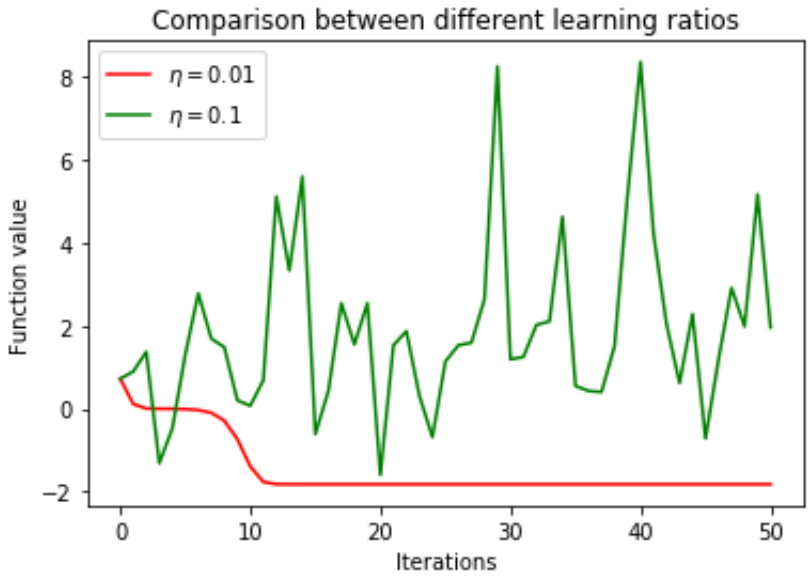


Figura 3: Comparativa gráfica de la Gradiente Descendente con diferentes valores de η.

Como se puede ver, después de 50 iteraciones, al utilizar un $\eta = 0{,}01$ se consigue converger a un óptimo local, mientras que al utilizar un $\eta = 0{,}1$ no se consigue.

En el primer caso, los valores de w van modificándose poco a poco con cada iteración, y por eso en el gráfico que puede ver que tiene una forma muy suavizada el cambio que sufre w entre iteración e iteración. En cambio, en el segundo caso, al tener un η mayor, se le da mucho peso al gradiente, y por tanto, los valores de w van modificándose muy bruscamente, como si estuviese oscilando entre la parte previa al mínimo y la posterior, sin llegar a converger en ningún momento.

Por tanto, el valor de η juega un factor clave en el algoritmo del descenso de gradiente. Si es muy bajo, los valores de w van descendiendo poco a poco, pero si el número de iteraciones son suficientes, se asegura llegar a un óptimo local. Sin embargo, si el η es muy grande, no se asegura en ningún momento que se pueda llegar al óptimo local, ya que puede ir saltando entre la parte anterior y la posterior al mínimo indefinidamente, llegando incluso a poder alejarse de éste.

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.

A continuación se muestra una captura de pantalla con el código que permite generar los datos y la salida en formato tabla, donde se muestran los 4 puntos con sus coordenadas iniciales, finales y valor del punto final.

In [7]:
```
# Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
# en los dos casos
# w inicial de cada caso y número máximo de iteraciones
initial_w_1 = np.array([0.1, 0.1])
initial_w_2 = np.array([1.0, 1.0])
initial_w_3 = np.array([-0.5, -0.5])
initial_w_4 = np.array([-1.0, -1.0])
max_iter = 50

# Cálculo del gradiente descendente para cada caso
w_1, it_1, w_array_1, func_val_1 = descent_gradient(initial_w_1, f, gradient_f, iterations=max_iter)
w_2, it_2, w_array_2, func_val_2 = descent_gradient(initial_w_2, f, gradient_f, iterations=max_iter)
w_3, it_3, w_array_3, func_val_3 = descent_gradient(initial_w_3, f, gradient_f, iterations=max_iter)
w_4, it_4, w_array_4, func_val_4 = descent_gradient(initial_w_4, f, gradient_f, iterations=max_iter)

# Mostrar por pantalla los resultados obtenidos usando pandas
# Crear una lista con los nombres de las columnas
column_header = ['x_0', 'y_0', 'x_f', 'y_f', 'Valor punto final']
row_header = ['Punto 1', 'Punto 2', 'Punto 3', 'Punto 4']

# Crear un array con los valores de cada fila
rows = np.array([[initial_w_1[0], initial_w_1[1], w_array_1[-1, 0], w_array_1[-1, 1], func_val_1[-1]],
                 [initial_w_2[0], initial_w_2[1], w_array_2[-1, 0], w_array_2[-1, 1], func_val_2[-1]],
                 [initial_w_3[0], initial_w_3[1], w_array_3[-1, 0], w_array_3[-1, 1], func_val_3[-1]],
                 [initial_w_4[0], initial_w_4[1], w_array_4[-1, 0], w_array_4[-1, 1], func_val_4[-1]]])

# Crear un nuevo DataFrame
df = pandas.DataFrame(rows, index=row_header, columns=column_header)

# Mostrarlo por pantalla
print(df)
```

```
         x_0  y_0       x_f       y_f  Valor punto final
Punto 1  0.1  0.1  0.243805 -0.237926          -1.820079
Punto 2  1.0  1.0  1.218070  0.712812           0.593269
Punto 3 -0.5 -0.5 -0.731377 -0.237855          -1.332481
Punto 4 -1.0 -1.0 -1.218070 -0.712812           0.593269
```

Apartado 4

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

Encontrar el máximo global de una función arbitraria puede llegar a ser difícil y depende de una serie factores:

- **Derivabilidad de la función**: Para poder encontrar un mínimo global, la función tiene que ser lo primero de todo derivable, y no todas las funciones lo son.
- **Forma de la función**: Si la función es convexa, como solo tiene un mínimo, es fácil encontrarlo con algoritmos como por ejemplo el Gradiente Descendente. Si la función tiene muchos óptimos locales o muchas curvaturas puede llegar a ser difícil dar con el óptimo global, y los algoritmos iterativos pueden llegar a quedarse en óptimos locales.
- **Punto de inicio**: El punto desde el que se empieza puede influir en el óptimo al que se llegue. En algunos casos, el punto inicial puede hacer que al aplicar algoritmos se llegue a un óptimo local y no se consiga salir de ahí. En otros casos, el punto puede estar más cerca del óptimo global, y por tanto se puede llegar a éste más fácilmente. Por tanto, hay que elegir con cuidado en qué punto se debería empezar.
- **El ratio de aprendizaje** (η): En algunos algoritmos, como por ejemplo el Gradiente Descendente y sus variantes, se utiliza un ratio de aprendizaje que permite dar más o menos peso al gradiente a la hora de acutalizar los valores de w. En caso de escoger un valor muy pequeño, la velocidad a la que se va a converger a un óptimo será más lenta, y por tanto se necesitarán más iteraciones. Un ratio muy elevado hará que sea muy difícil converger, ya que el valor de w irá pegando muchos saltos. Por tanto, un ratio de aprendizaje adecuado hará que se pueda llegar a un óptimo de mejor o peor forma. Dicho esto, ninguno de los dos garantiza que el óptimo al que se acabe llegando sea global, ya que perfectamente puede llegar a un óptimo local y quedarse ahí.

2. Ejercicio sobre Regresión Lineal

Este ejercicio ajusta modelos de regresión a vectores de características extraidos de imágenes de digitos manuscritos. En particular se extraen dos característcas 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 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)).

Primero, vamos a observar las funciones implementadas para realizar los ajustes:

In [7]:
```python
# Funcion para calcular el error
def Err(x, y, w):
    error = np.square(x.dot(w) - y.reshape(-1, 1))  # Calcular el error cuadrático para cada vector de caracterísi
    error = error.mean()                            # Calcular la media de los errors cuadráticos (matriz con una c

    return error

# Derivada de la función del error
def diff_Err(x,y,w):
    d_error = x.dot(w) - y.reshape(-1, 1)           # Calcular producto vectorial de x*w y restarle y
    d_error =  2 * np.mean(x * d_error, axis=0)     # Realizar la media del producto escalar de x*error y la media

    d_error = d_error.reshape(-1, 1)                # Cambiar la forma para que tenga 3 filas y 1 columna

    return d_error
```

Figura 4: Función de error y derivada de ésta para el gradiente.

In [10]:
```python
# Pseudoinversa
def pseudoinverse(X, y):
    """
    Función para el cálculo de pesos mediante el algoritmo de la pseudoderivada

    :param X: Matriz que contiene las caracterísiticas
    :param y: Matriz que contiene las etiquetas relacionadas a las características

    :returns w: Pesos calculados mediante ecuaciones normales
    """

    X_transpose = X.transpose()                     # Guardamos la transpuesta de X
    y_transpose = y.reshape(-1, 1)                  # Convertimos y en una matriz columna (1 fila con n columnas)

    # Aplicamos el algoritmo para calcular la pseudoinversa
    w = np.linalg.inv(X_transpose.dot(X))
    w = w.dot(X_transpose)

    # Hacemos el producto de matrices de la pseudoinversa y la matriz columna y
    w = w.dot(y_transpose)

    return w
```

Figura 5: Implementación del algoritmo de la pseudoinversa.

In [8]:
```python
# Gradiente Descendente Estocastico
def sgd(X, y, eta, M=64, iterations=200):
    """
    Función para calcular el Gradiente Descendente Estocástico.
    Selecciona minibatches aleatorios de tamaño M de la muestra original
    y ajusta en un número de iteraciones los pesos.

    :param X: Muestra de entrenamiento
    :param y: Vector de etiquetas
    :param eta: Ratio de aprendizaje
    :param M: Tamaño de un minibatch (64 por defecto)
    :param iterations: Número máximo de iteraciones

    :return w: Pesos ajustados
    """

    # Crear un nuevo vector de pesos inicializado a 0, establecer el número de iteraciones
    # inicial y obtener el número de elementos (N)
    w = np.zeros((3, 1), np.float64)
    N = X.shape[0]
    iter = 0

    # Mientras el número de iteraciones sea menor al máximo, obtener un minibatch
    # de tamaño M con valores aleatorios de X y ajustar los pesos con estos valores
    while iter < iterations:
        iter += 1

        # Escoger valores aleatorios de índices sin repeticiones y obtener los elementos
        index = np.random.choice(N, M, replace=False)
        minibatch_x = X[index]
        minibatch_y = y[index]

        # Actualizar w
        w = w - eta * diff_Err(minibatch_x, minibatch_y, w)

    return w
```

Figura 6: Implementación del Gradiente Descendente Estocástico.

Para realizar los ajustes utilizando el Gradiente Descendente Estocástico vamos a suponer una serie de cosas, ya que no se especifican en el enunciado del problema:

- Como no se especifica cuántas iteraciones tiene que dar el algoritmo, vamos a suponer que, por defecto, da **200 iteraciones**.
- Como tampoco se especifica que η utilizar, vamos a utilizar $\eta = 0{,}05$, ya que en combinación con el número de iteraciones, ofrece un buen resultado. También se ha probado con un $\eta = 0{,}01$ y se han conseguido resultados muy similares, pero con la contra de tener que realizar un mayor número de iteraciones.
- Como no se especifica ningún criterio para crear los *batches*, vamos a escoger en cada iteración una parte aleatoria de la muestra de tamaño M, siendo $M = 64$ por defecto (y también el valor utilizado).
- Como tampoco se especifica un valor inicial para w, vamos a suponer que $w = \{0, 0, 0\}$.

Una vez dicho esto, vamos a ver los resultados de cada ajuste, comenzando con el Gradiente Descendente Estocástico:

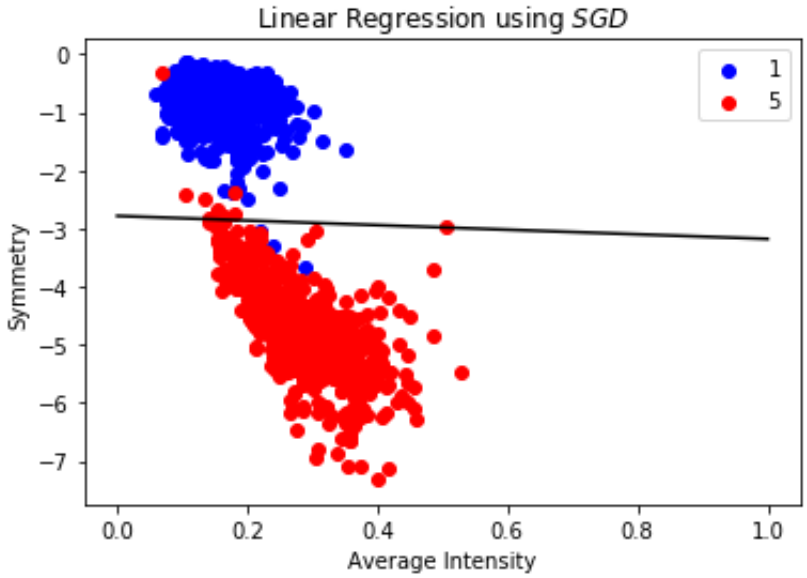


Figura 7: Gráfica del ajuste de regresión mediante SGD.

```
Bondad del resultado para grad. descendente estocastico:

Ein:  0.08273424444121182
Eout:  0.1330378970482456
```

Como se puede comprobar, con los datos especificados anteriormente ($\eta = 0{,}05$, 200 iteraciones y $M = 64$) se obtiene un ajuste muy bueno, si bien es cierto que van a haber algunos datos que no sean clasificados correctamente, ya que los datos no son linealmente independientes del todo (se puede ver claramente que hay algunos datos cuyos valores no se corresponden con la clase que supuestamente les toca).

Para poder pintar la recta, se ha pintado en el rango $[0, 1]$ (eje X) según los valores de w. Para ello, se ha resuleto la ecuación $y = w_0 + w_1 x_1 + w_2 x_2$, donde y es la clase obtenida, x_1 el valor en el eje X y x_2 el valor en el eje Y. En este caso se quería obtener el valor de x_2, ya que se han supuesto los casos en los que $x_1 = 0$ (extremo inferior del intervalo en el que se quiere pintar la recta) y $x_1 = 1$ (extremo superior del intervalo en el que se quiere pintar la recta). Suponiento que $y = 0$ en ambos casos, se ha obtenido lo siguiente:

$$x_1 = 0,\ x_2 = \frac{-w_0}{w_2}$$

$$x_1 = 1,\ x_2 = \frac{-w_0 - w_1}{w_2}$$

Por tanto, los valores en el eje Y tendrán en los extremos $[0, 1]$ los valores $[\frac{-w_0}{w_2}, \frac{-w_0 - w_1}{w_2}]$, interpolando el resto de valores para poder formar la recta. Este

mismo proceso se ha seguido para la pseudoinversa.

A continuación, vamos a observar los resultados del ajuste de regresión mediante la pseudoinversa:

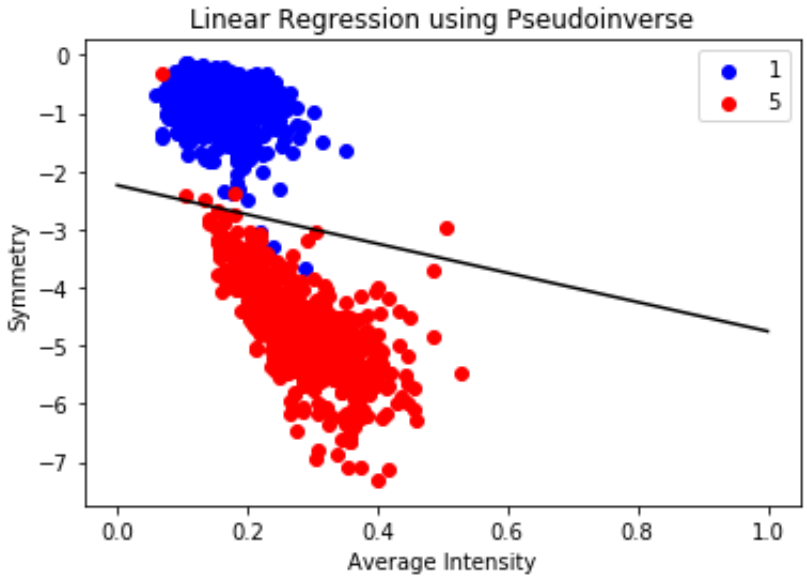


Figura 8: Gráfica del ajuste de regresión mediante el algoritmo de la pseudoinversa.

```
In [15]: # Cálculo de w mediante la pseudoinversa
         w = pseudoinverse(x, y)

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

         Bondad del resultado para pseudoinversa:

         Ein:  0.07918658628900396
         Eout:  0.13095383720052578
```

Figura 9: Resultados de los errores obtenidos con el ajuste de regresión para E_{in} y E_{out}.

Como se puede ver, existen unas pequeñas diferencias. En este caso, la recta obtenida tiene un poco más de pendiente, y permite obtener un error un poco menor en E_{in}, conservando sin embargo un error similar en E_{out}.

Ambas técnicas permiten obtener buenos resultados. Sin embargo, en este caso, la pseudoinversa permite obtener unos resultados algo mejores, ya que intenta calcular los valores de w resolviendo un sistema de ecuaciones, mientras que en el caso del Gradiente Descendente Estocástico se intentan ajustar los valores de w de

forma iterativa y, al seleccionar elementos aleatorios de la muestra, no se asegura que se realice un ajuste con todos los datos, lo cuál puede hacer que sea necesario dar más iteraciones.

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 $[-size, size] \times [-size, size]$

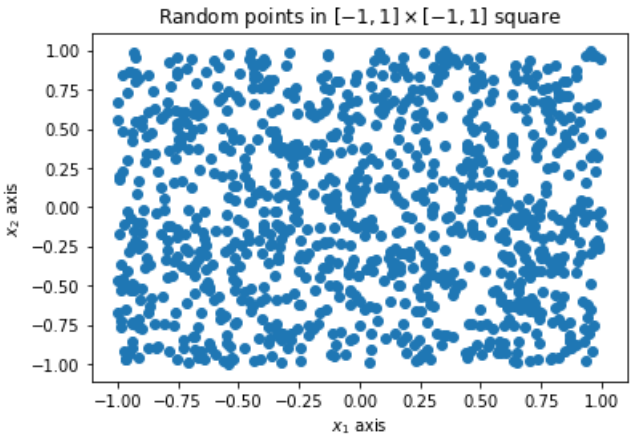
- EXPERIMENTO

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

  Para generar los datos, hemos utilizado esta función:

```
In [16]: # Simula datos en un cuadrado [-size,size]x[-size,size]
         def simula_unif(N, d, size):
             return np.random.uniform(-size,size,(N,d))
```

  Al generar los 1000 puntos, hemos obtenido el siguiente gráfico:

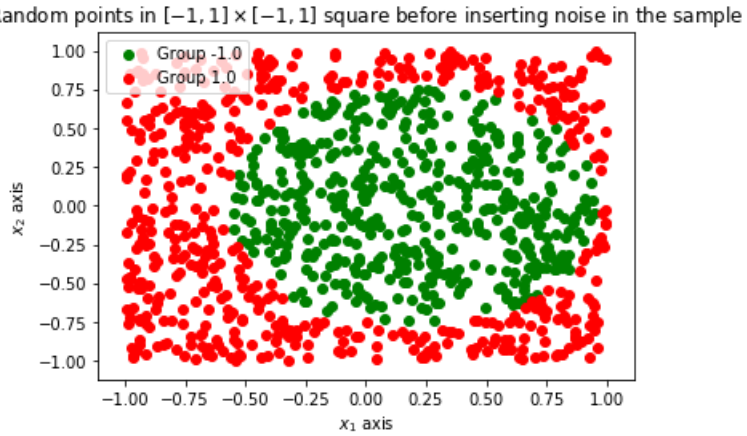


  b) Consideremos la función $f(x_1, x_2) = \text{sign}((x_1 - 0{,}2)^2 + x_2^2 - 0{,}6)$ que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas obtenido.

  La implementación de la función es la siguiente:

In [17]:
```
# Crea una matriz de etiquetas a partir de las entradas
def sign_labels(X):
    return np.sign(np.square(X[:, 0] - 0.2) + np.square(X[:, 1]) - 0.6)
```

Al generar el signo para los puntos, se ha obtenido el siguiente gráfico:



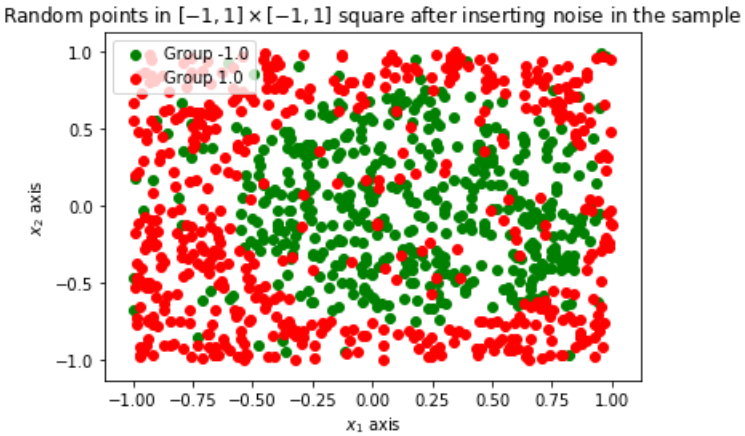
La función para insertar ruido es la siguiene:

In [20]:
```
# Inserta ruido en un porcentaje de las etiquetas de la población
def insert_noise(labels, ratio=0.1):
    # Calcular el número de elementos con ruido según el ratio
    n = labels.shape[0]
    noisy_elements = int(n * ratio)

    # Obtener una muestra aleatoria entre [0, n) de n * ratio elementos sin repeticiones
    index = np.random.choice(np.arange(n), noisy_elements, replace=False)

    # Cambiar el signo del porcentaje de etiquetas
    labels[index] = -labels[index]
```

Tras insertar ruido, se ha obtenido el siguiente gráfico:



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).

Al realizar el ajuste de regresión con el Gradiente Descendente Estocástico, hemos obtenido lo siguiente:

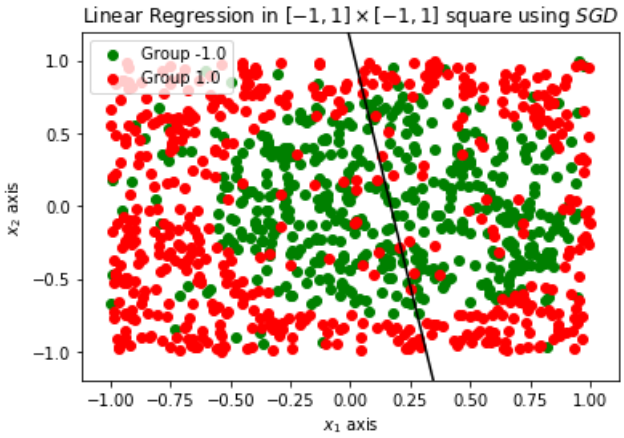


Figura 10: Gráfico del ajuste por regresión.

```
Bondad del resultado para grad. descendente estocastico:

Ein:  0.9493372939157104
```

Figura 11: Error para E_{in}.

Para dibujar la recta, se ha resuelto la misma ecuación que en el ejercicio anterior, solo que esta vez en el rango $[-1, 1]$, y se han obtenido los siguientes valores:

$$x_1 = -1,\ x_2 = \frac{-w_0 + w_1}{w_2}$$

$$x_1 = 1,\ x_2 = \frac{-w_0 - w_1}{w_2}$$

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.

Para el experimento repetido 1000 veces, se han obtenido los siguientes resultados:

```
Valor medio de la bondad del resultado para grad. descendente estocastico:

Ein:  0.927351060956626
Eout:  0.897456517009633
```

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}.

Los dos valores de los errores son muy malos, ya que están próximos a 1. Esto se debe a que una parte de los datos está dentro de la otra, y los modelos lineales están muy limitados en este aspecto, ya que una recta no puede dividir perfectamente los datos por mucho que se intente. Habría que utilizar otras funciones, como por ejemplo las hiperbólicas para intentar obtener un mejor ajuste.

En el caso de E_{in}, el error obtendo es mayor que el de E_{out}, ya que al haber introducido ruido en la muestra de entrenamiento, los datos no serán linealmente independientes, y por tanto va a fallar más que de costumbre. En E_{out}, como no se ha metido ruido en las etiquetas, los datos no estarán tan dispersos, pero igualmente, no se podrá obtener un ajuste bueno por los motivos comentados anteriormente.

3. Bonus

Apartado 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.

- 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 el gradiente descendente.

Antes de empezar, vamos a calcular analíticamente el valor de la *Hessiana*. Para ello, necesitamos calcular antes $\frac{\partial^2 f}{\partial x^2}$, $\frac{\partial^2 f}{\partial y^2}$ y $\frac{\partial^2 f}{\partial xy}$:

$$\frac{\partial^2 f}{\partial x^2} = \frac{\partial^2}{\partial x^2}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 2 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) \tag{9}$$

$$\frac{\partial^2 f}{\partial y^2} = \frac{\partial^2}{\partial y^2}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 4 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) \tag{10}$$

$$\frac{\partial^2 f}{\partial xy} = \frac{\partial^2}{\partial xy}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 8\pi^2 \cos(2\pi x)\cos(2\pi y) \tag{11}$$

Con lo cuál, tenemos que:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial xy} \\ \\ \frac{\partial^2 f}{\partial xy} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} \tag{12}$$

$$H = \begin{bmatrix} 2 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) & 8\pi^2 \cos(2\pi x)\cos(2\pi y) \\ \\ 8\pi^2 \cos(2\pi x)\cos(2\pi y) & 4 - 8\pi^2 \sin(2\pi x)\sin(2\pi y) \end{bmatrix} \tag{13}$$

La implementación del algoritmo es la siguiente:

In [19]:
```python
# Método de Newton
def newtons_method(initial_w, iterations=50, eta=None):
    """
    Función para el cálculo de pesos mediante el Método de Newton
    Es una modificación del algoritmo de Gradiente Descendente usando
    la invertida de la matriz Hessiana como ratio de aprendizaje

    :param initial_w: Valor de w inicial
    :param iterations: Número máximo de iteraciones (por defecto 50)
    :param eta: Ratio de aprendizaje (por defecto None)

    :return Devuelve los pesos ajustados (w), el número de iteraciones
            para llegar a esos pesos (iter), un array con los valores
            de w (w_list) y un array con los valores de la función
            (f_list)
    """

    w = np.copy(initial_w)        # Copiar los w iniciales para no modificarlos
    iter = 0                      # Iniciar las iteraciones a 0
    w_list = []                   # Se inicializa una lista vacía con los valors de w
    f_list = []                   # Se inicializa una lista vacía con los valors de la función

    w_list.append(w)              # Añadir valor inicial de w
    f_list.append(f(*w))          # Añadir valor inicial de w evaluado en function

    # Mientras el número de iteraciones no supere el máximo, calcular
    # la hessiana, invertirla, calcular el gradiente y ajustar w
    # Añadir además a las listas correspondientes los valores de w y de w evaluado en f
    while iter < iterations:
        iter += 1

        # Calcular la Hessiana, invertirla (pseudoinversa) y calcular el gradiente
        hessian = hessian_f(*w)
        hessian = np.linalg.inv(hessian)
        gradient = gradient_f(*w)

        # Calcular theta (producto vectorial del Hessiana invertida y el gradiente)
        theta = hessian.dot(gradient.reshape(-1, 1))
        theta = theta.reshape(-1,)                    # Hacer que theta sea un vector fila

        # Multiplicar theta por eta, en caso de que se haya especificado
        if eta:
            theta = eta * theta

        # Actualizar w
        w = w - theta

        # Añadir w y su valor a las listas correspondientes
        w_list.append(w)
        f_list.append(f(*w))


    return w, iter, np.array(w_list), np.array(f_list)
```

Una vez dicho esto, vamos a mostrar los resultados obtenidos:

	x_0	y_0	x_f	y_f	Valor punto final
Punto 1	0.1	0.1	-47.141424	66.537931	11077.272807
Punto 2	1.0	1.0	0.949129	0.974568	2.900405
Punto 3	-0.5	-0.5	-0.475114	-0.487805	0.725482
Punto 4	-1.0	-1.0	-0.949129	-0.974568	2.900405

Figura 12: Puntos obtenidos al aplicar el Método de Newton.

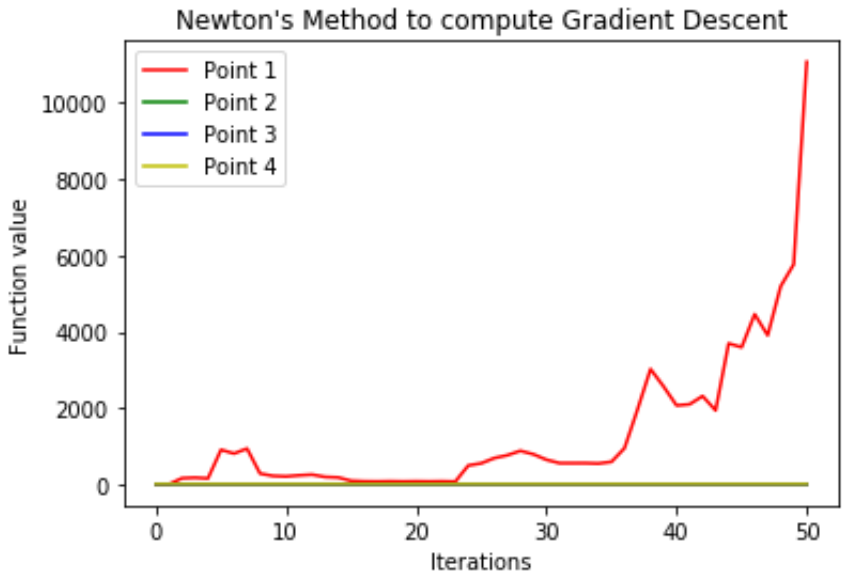


Figura 13: Evolución de los valores de $f(x, y)$ con el Método de Newton.

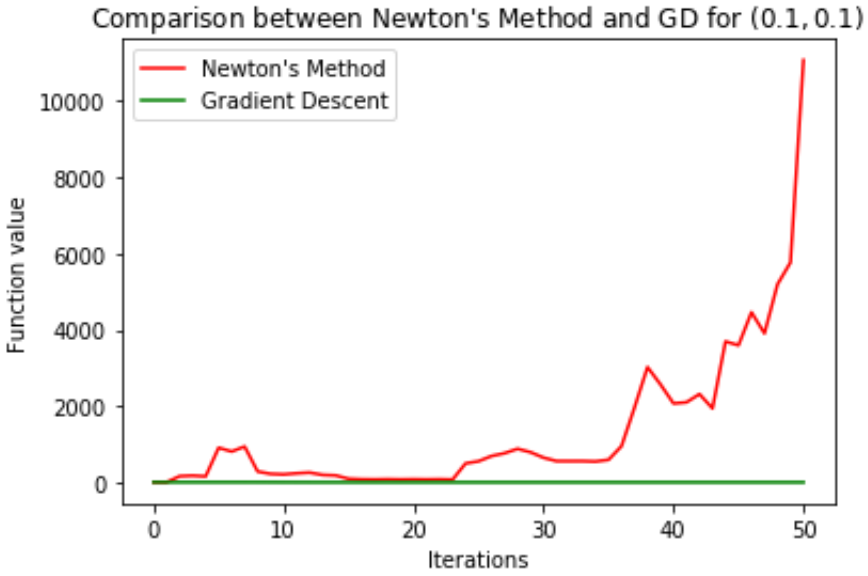


Figura 14: Minimización del punto (0,1, 0,1) del GD comparada con el Método de Newton.

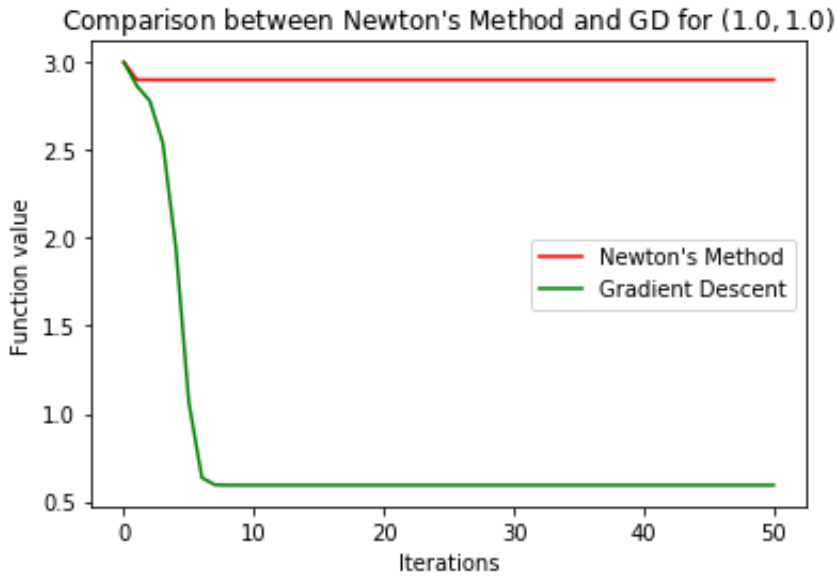


Figura 15: Minimización del punto $(1, 1)$ del GD comparada con el Método de Newton.

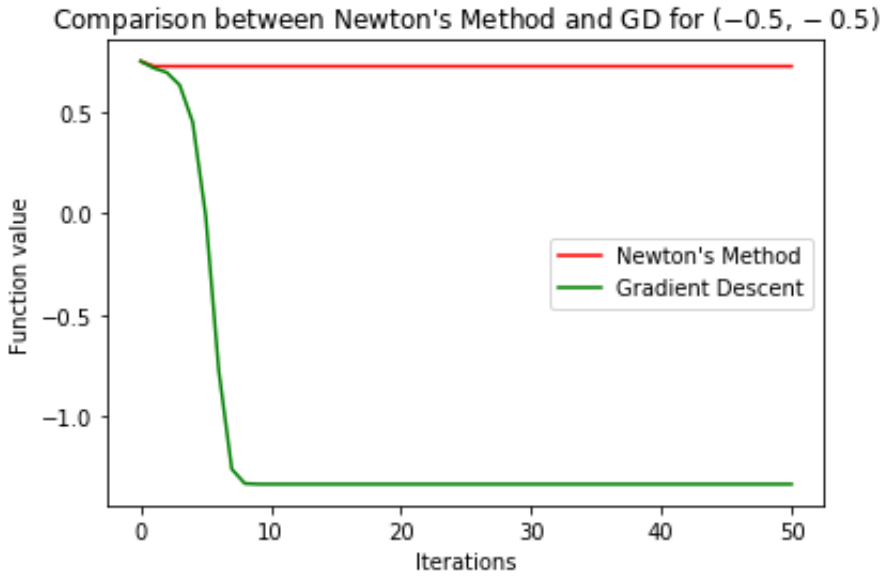


Figura 16: Minimización del punto $(-0{,}5, -0{,}5)$ del GD comparada con el Método de Newton.

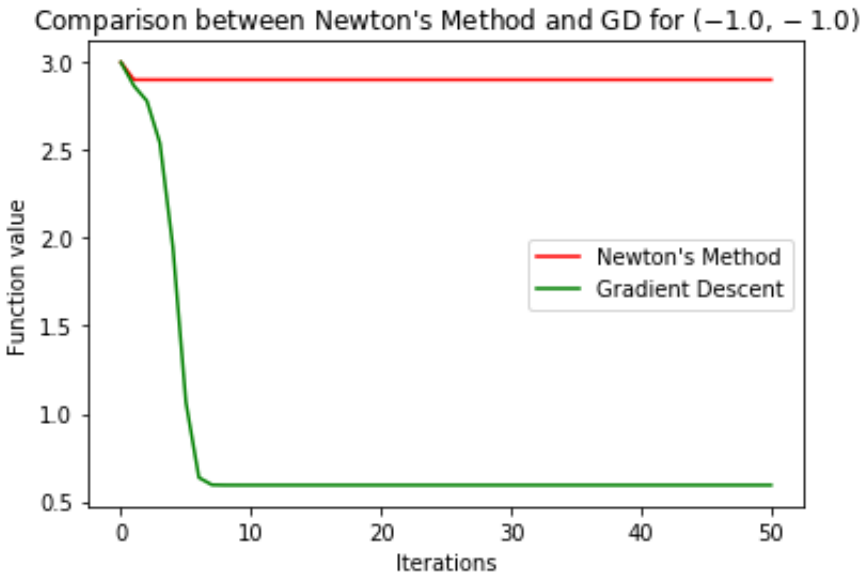


Figura 17: Minimización del punto $(-1, -1)$ del GD comparada con el Método de Newton.

Como se puede ver, las minimizacions no son muy buenas. En el primer caso es pésima ya que el valor de la función crece hasta valores muy elevados, posiblemente debido a que el valor de la segunda derivada es muy elevado justo en el punto donde empieza, con lo cuál sale disparada y empieza a crecer. En los otros casos, los valores de la función van decreciendo, pero a un ritmo muy lento, con lo cuál tampoco se llega a un óptimo local, como en el primer caso.

Para intentar solucionar esto, vamos a meter un *learning rate* $\eta = 0{,}01$ para evitar que en el primer caso el Método de Newton haga que la función se salga disparada. Con esto obtenemos los siguientes resultados:

	x_0	y_0	x_f	y_f	Valor punto final
Punto 1	0.1	0.1	0.049202	0.048900	0.191232
Punto 2	1.0	1.0	0.980392	0.990415	2.937804
Punto 3	-0.5	-0.5	-0.490226	-0.495235	0.734512
Punto 4	-1.0	-1.0	-0.980392	-0.990415	2.937804

Figura 18: Puntos obtenidos al aplicar el Método de Newton con $\eta = 0{,}01$.

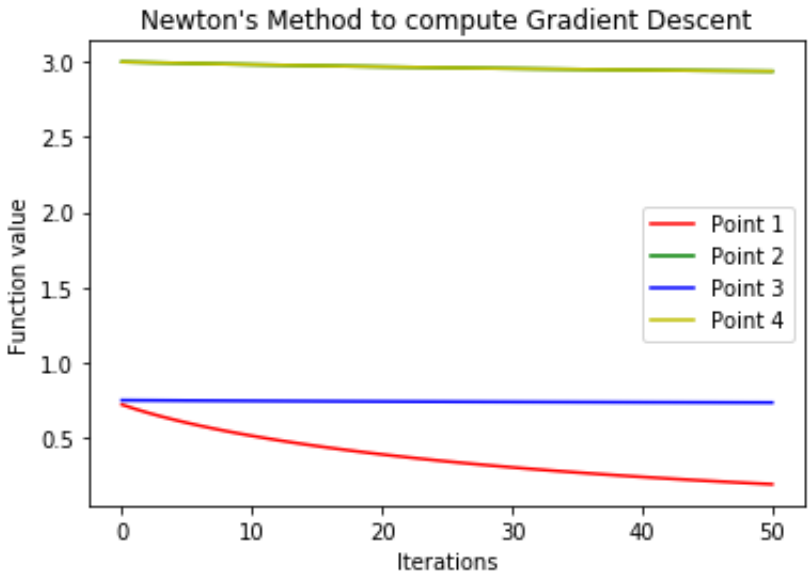


Figura 19: Evolución de los valores de $f(x, y)$ con el Método de Newton con $\eta = 0{,}01$.

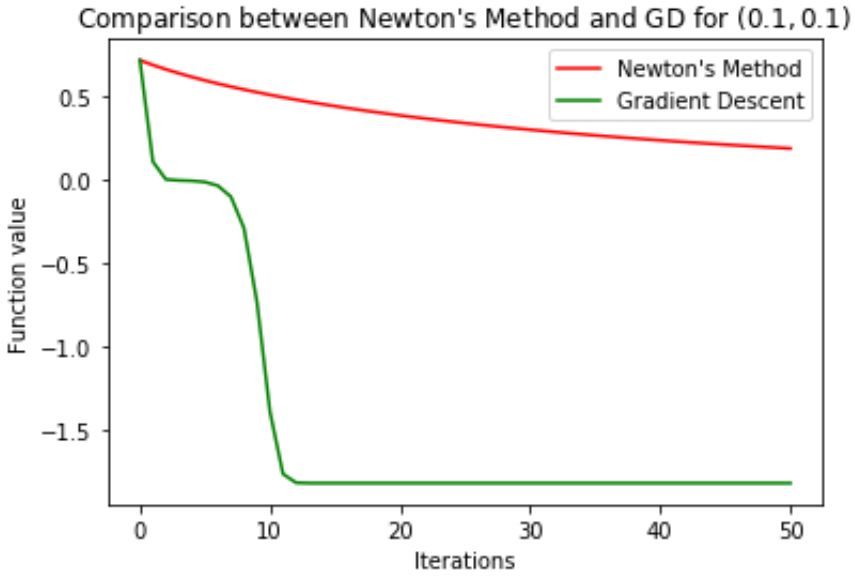


Figura 20: Minimización del punto $(0{,}1, 0{,}1)$ del GD comparada con el Método de Newton con $\eta = 0{,}01$.

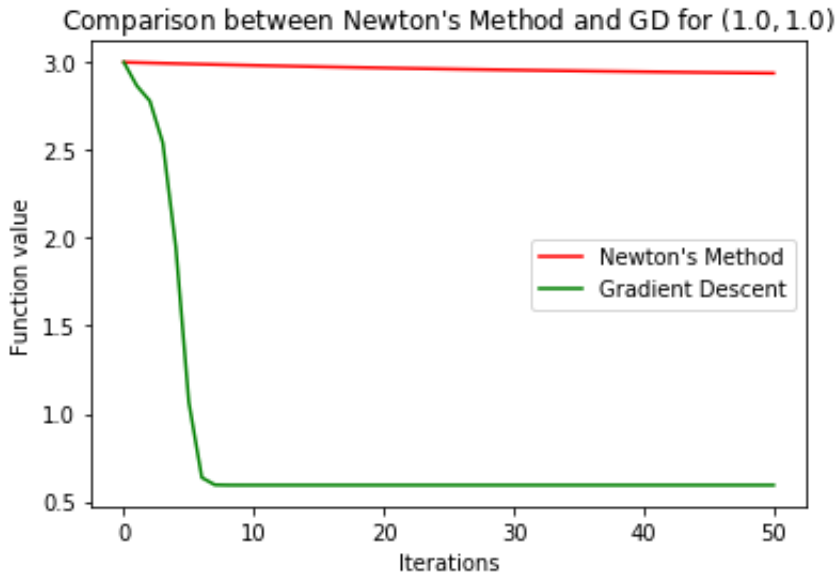


Figura 21: Minimización del punto $(1, 1)$ del GD comparada con el Método de Newton con $\eta = 0{,}01$.

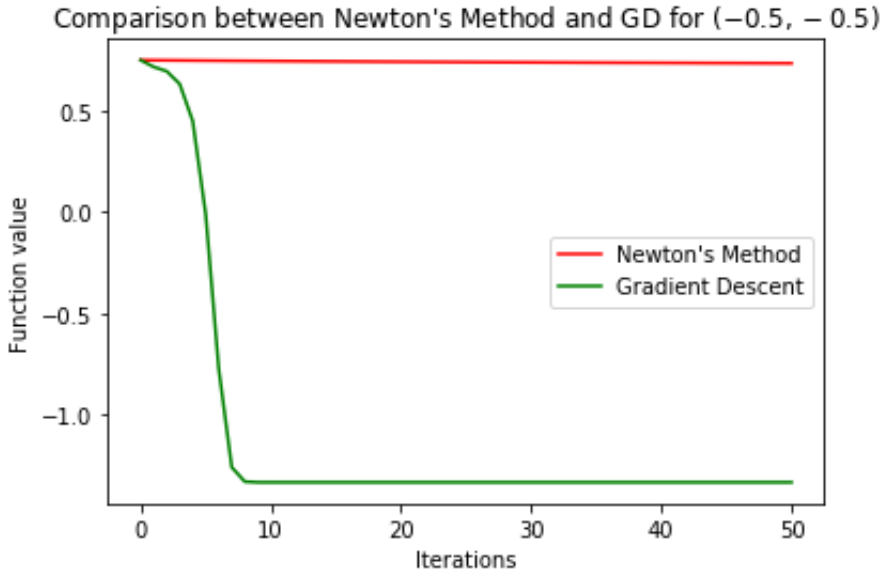


Figura 22: Minimización del punto $(-0{,}5, -0{,}5)$ del GD comparada con el Método de Newton con $\eta = 0{,}01$.

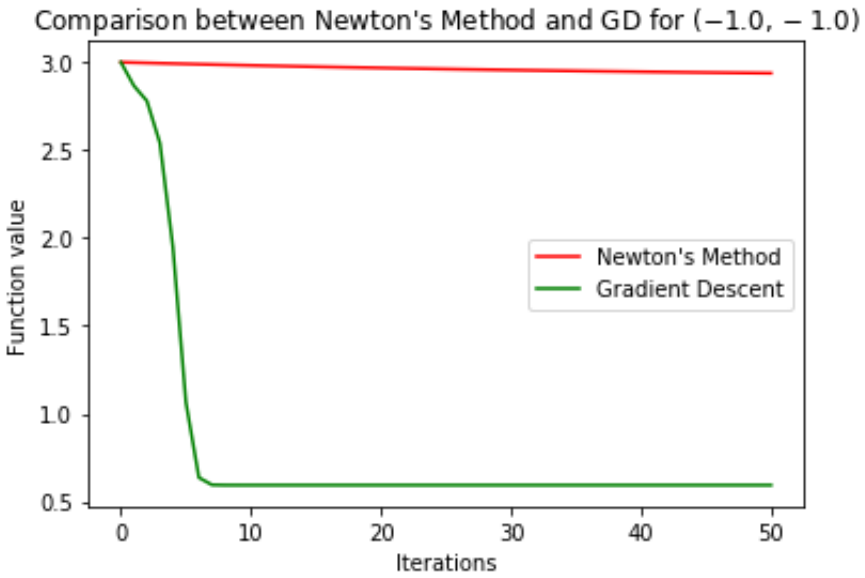


Figura 23: Minimización del punto $(-1, -1)$ del GD comparada con el Método de Newton con $\eta = 0{,}01$.

Como se puede ver, al utilizar un $\eta = 0{,}01$, los valores son algo más normales. Para el primer punto se puede ver que esta vez, en vez de salir disparado para arriba, los valores de la función se intentan acercar al óptimo local. En el resto de casos, sin embargo, el ratio de descenso es aún peor que antes, con lo cuál los valores de la función descienden de una forma mucho más lenta que antes. Además, como en la situación anterior, no se llega a converger en ningún momento. Ninguno de los puntos llega a un óptimo local.

Las conclusiones que se pueden extraer del Método de Newton, en este caso, son las siguientes:

- La función no es la más adecuada, ya que no es completamente convexa. Tiene demasiados óptimos locales, y el punto de inicio condiciona mucho hacia dónde se dirige.
- Los puntos de inicio pueden no ser los más adecuados. En este caso, por ejemplo, el punto $(0{,}1, 0{,}1)$ tiene un valor de la segunda derivada muy próximo a 0. Por eso, al multiplicar el gradiente por la inversa de la Hessiana, se obtiene un valor muy elevado, lo cuál permite salir de la zona de mínimos y hace que los valores de la función obtenidos sean muy elevados y que solo vaya creciendo.
- El criterio de la segunda derivada no asegura que se consiga converger siempre a un óptimo local. En este caso, ninguno de los puntos consiguió llegar a un

óptimo local, a pesar de que el Gradiente Descendente si que consiguió llegar, y que la segunda derivada ofrece una mayor velocidad de convergencia. Lo ideal sería combinar un *learning rate* fijo con la segunda derivada cuando más se acerque al óptimo, para así acelerar la convergencia.

Una vez dicho esto, para conlcuir todo el análisis, podemos confirmar que el algoritmo del Gradiente Descendente ofrece unos mejores resultados, ya que en todos los casos se consigue llegar a un óptimo, mientras que el Método de Newton, aún combinándolo con un *learning rate*, no consigue converger en ningún momento.