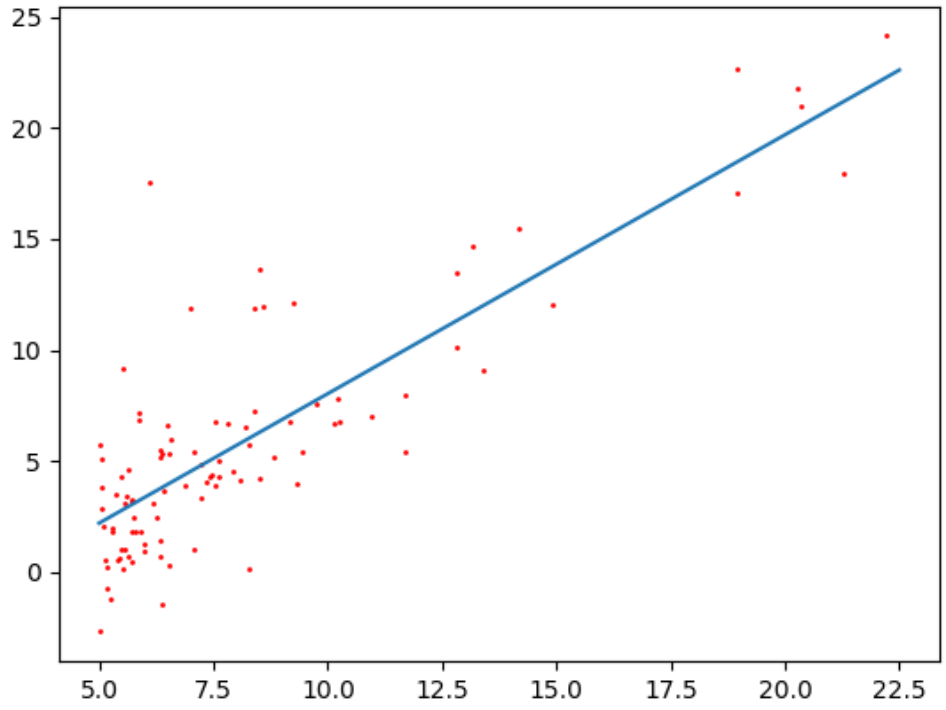
Aprendizaje automático: Práctica 1

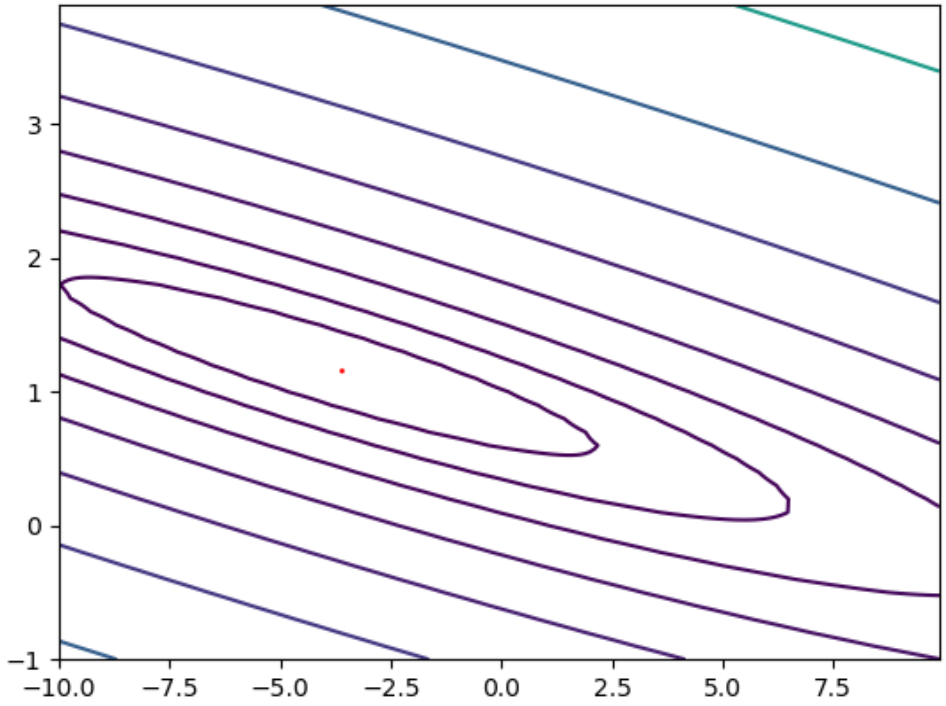
Jorge Rodríguez García y Gonzalo Sanz Lastra

```python
import numpy as np
from pandas.io.parsers import read_csv   # para leer .csv
from matplotlib import pyplot as plt     # para dibujar las graficas
from mpl_toolkits.mplot3d import Axes3D  # para dibujar las graficas en 3D
from matplotlib import cm

def carga_csv(file_name):
    """carga el fichero csv especificado y lo devuelve en un array de numpy"""
    valores = read_csv(file_name, header=None).values

    return valores.astype(float)

def hTransposed(X, O):
    """devuelve la funcion h(x) usando la matriz transpuesta de O"""
    AuxO = O[np.newaxis]
    AuxX = X[np.newaxis]
    return (np.dot(AuxX, np.transpose(AuxO))).sum()

def hLine(X, O):
    """devuelve la funcion h(x) usando la ecuacion de la recta"""
    return O[0] + O[1]*X

def coste(X, Y, O):
    """devuelve la funcion de coste, dadas X, Y, y thetas"""
    H = np.dot(X, O)
    Aux = (H-Y)**2
    return Aux.sum()/(2*len(X)) # lo mismo que hacer la formula con el sumatorio...

def normalizeScales(X):
    """normalizacion de escalas, para cuando haya mas de un atributo"""
    mu = X.mean(0)   # media de cada columna de X
    sigma = X.std(0) # desviacion estandar de cada columna de X

    X_norm = (X - mu)/sigma

    return X_norm, mu, sigma

def normalizeValues(valoresPrueba, mu, sigma):
    """normaliza los valores de prueba con la mu y sigma de los atributos X (al normalizarlos)"""
    return (valoresPrueba - mu)/sigma

def gradientDescendAlgorithm(X, Y, alpha, m, n, loops):
    """minimiza la funcion de coste, hallando las O[0], O[1], ... que hacen el coste minimo, y por tanto, h(x) mas precisa"""

    O = np.zeros(n)      # O[0], O[1], ..., inicialmente todas a 0
    cost = np.zeros(loops)

    # con 1500 iteraciones basta para encontrar las thetas que hacen el coste minimo
    for i in range(loops):
        """for rows in range(m):      # bucle utilizando la ecuacion de la recta para h (obviamos la columna de 1s)
            for cols in range(n):
                sumatorio[cols] += (hLine(X[rows, 1], O) - Y[rows])*X[rows, cols]"""

        """for rows in range(m):   # b utilizando la transpuesta de O
            h = hTransposed(X[rows], O)
            for cols in range(n):
                sumatorio[cols] += (h - Y[rows])*X[rows, cols]

        cost[i] = coste(X, Y, O)
        O = O - alpha*(1/m)*sumatorio # actualizamos thetas"""

        cost[i] = coste(X, Y, O)
        H = np.dot(X, O)                    #X(47,3)*O(3,1) = H(47,1)
        O = O - alpha*(1/m)*(X.T.dot(H-Y)) #X.T(3,47)*(H-Y)(47,1) = SUM(3,1)

    return O, cost

def normalEquation(X, Y):
    """minimiza la funcion de coste, hallando las O[0], O[1], ... que hacen el coste minimo, de forma analitica"""
    x_transpose = np.transpose(X)
    x_transpose_dot_x = x_transpose.dot(X)
    temp_1 = np.linalg.inv(x_transpose_dot_x)
    temp_2 = x_transpose.dot(Y)

    return temp_1.dot(temp_2)

def functionGraphic(X, Y, O):
    """muestra el grafico de la funcion h(x)"""

    x = np.linspace(5, 22.5, 256, endpoint=True)
    y = hLine(x, O)

    # pintamos muestras de entrenamiento
    plt.scatter(X[:, 1], Y, 1, 'red')
    # pintamos funcion de estimacion
    plt.plot(x, y)
    plt.savefig('H(X).png')
    plt.show()

def multiVariableCostGraphics(C):
    """muestra el grafico de la funcion de coste J"""

    x = np.linspace(0, 50, 1500, endpoint=True)
    plt.plot(x, C)
    plt.savefig('J(O).png')
    plt.show()

def twoVariableCostGraphics(X, Y, O):
    """muestra diversas graficas de la funcion de coste"""

    # grafica 3D
```

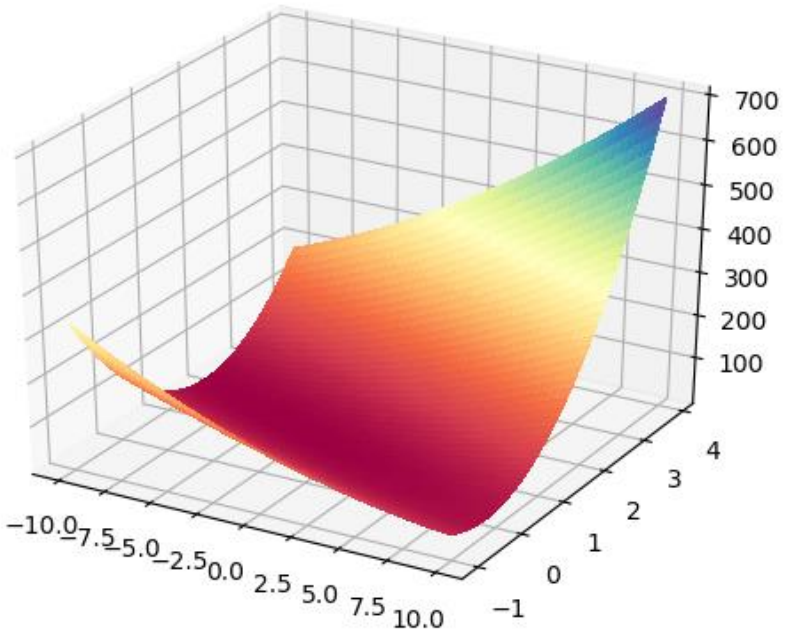
```
    fig = plt.figure()
    ax = fig.gca(projection = '3d')

    Theta0, Theta1, Coste = make_data([-10, 10], [-1, 4], X, Y)

    ax.plot_surface(Theta0, Theta1, Coste, cmap = cm.Spectral, linewidth = 0, antialiased = False)
    plt.savefig('Coste3D.png')
    plt.show()

    # contour
    fig, ax = plt.subplots()
    ax.contour(Theta0, Theta1, Coste, np.logspace(-2, 3, 20)) # lista con los ticks de las curvas de nivel
                                                              # con escala logaritmica de 20 valores entre 10^-2 y 10^-3
    plt.scatter(O[0], O[1], 1, 'red')
    plt.savefig('Contour.png')
    plt.show()

def make_data(t0_range, t1_range, X, Y):
    """Genera las  matrices X (Theta0), Y (Theta1), Z (Coste) para generar un plot en 3D"""

    step = 0.1
    Theta0 = np.arange(t0_range[0], t0_range[1], step)
    Theta1 = np.arange(t1_range[0], t1_range[1], step)

    Theta0, Theta1 = np.meshgrid(Theta0, Theta1)
    #Theta0 y Theta1 tienen las mismas dimensiones, de forma que cogiendo un elemento de cada uno se generan las coordenadas x, y
    # de todos los puntos de la rejilla

    Coste = np.empty_like(Theta0)

    for ix, iy in np.ndindex(Theta0.shape):
        Coste[ix, iy] = coste(X, Y, [Theta0[ix, iy], Theta1[ix, iy]])

    return [Theta0, Theta1, Coste]

def main():
    valores = carga_csv("ex1data2.csv")

    X = valores[:, :-1] # matriz X, con todas las filas y todas las columnas menos la ultima (ys)
    Y = valores[:, -1]  # matriz Y, con todas las filas y la ultima columna

    m = X.shape[0]      # numero de muestras de entrenamiento
    n = X.shape[1] + 1  # numero de variables x que influyen en el resultado y, mas la primera columna de 1s
    alpha = 0.01        # coeficiente de aprendizaje

    X_norm, mu, sigma = normalizeScales(X)

    # Se colocan m filas de 1 columna de 1s al principio (concatenacion de matrices)
    X_norm = np.hstack([np.ones([m, 1]), X_norm])
    X = np.hstack([np.ones([m, 1]), X])
```

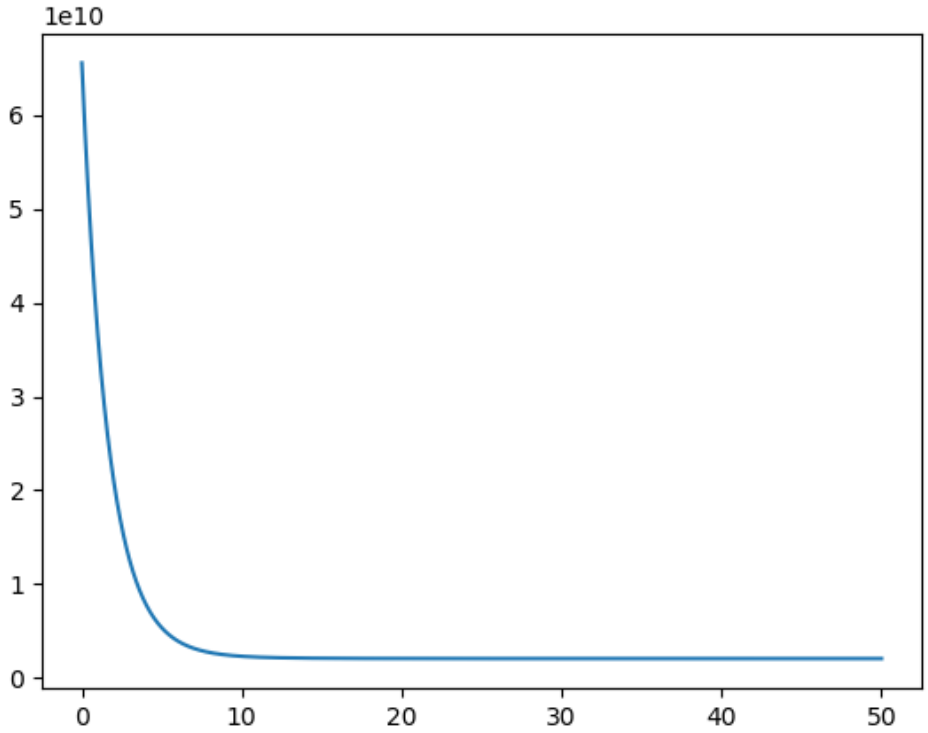
```


    # modelo analitico de minimizar el coste (sin normalizar los atributos)
    ONormalEq = normalEquation(X, Y)

    valoresPrueba = np.array([1, 1650, 3])
    if n < 3:                   # solo lo pintaremos si no tiene mas de dos variables x (pasaria a ser multidimensional)
        O, C = gradientDescendAlgorithm(X, Y, alpha, m, n, 1500)  #  modelo de descenso de gradiente de minimizar el coste con la X sin normalizar (1 x)
        functionGraphic(X, Y, O)                # pintamos la grafica de la funcion h(x)
        twoVariableCostGraphics(X, Y, O)        # graficos para ver el coste con dos variables

        # pruebas de resultados por ecuacion normal y descenso de gradiente (deben dar resultados similares)
        valoresPrueba = np.array([1, 10])
        print(np.dot(O[np.newaxis], np.transpose(valoresPrueba[np.newaxis])).sum())

    else:
        O, C = gradientDescendAlgorithm(X_norm, Y, alpha, m, n, 1500)#  modelo de descenso de gradiente de minimizar el coste con la X normalizada(mas de 1 x)

        # pruebas de resultados por ecuacion normal y descenso de gradiente (deben dar resultados similares)
        valoresPruebaNorm = normalizeValues(valoresPrueba[1:], mu , sigma) # valores de prueba normalizados para el descenso de gradiente
        valoresPruebaNorm = np.insert(valoresPruebaNorm, 0, [1])
        print(np.dot(O[np.newaxis], np.transpose(valoresPruebaNorm[np.newaxis])).sum())

    print(np.dot(ONormalEq, np.transpose(valoresPrueba[np.newaxis])).sum())

    multiVariableCostGraphics(C)

main()
```


25
20
15
10
5
0
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5


3
2
1
0
−1
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

o


700
600
500
400
300
200
100
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
−1
0
1
2
3
4


1e10
6
5
4
3
2
1
0
0
10
20
30
40
50