Práctica 1

Alejandro Aizel Boto
Miguel Robledo Blanco

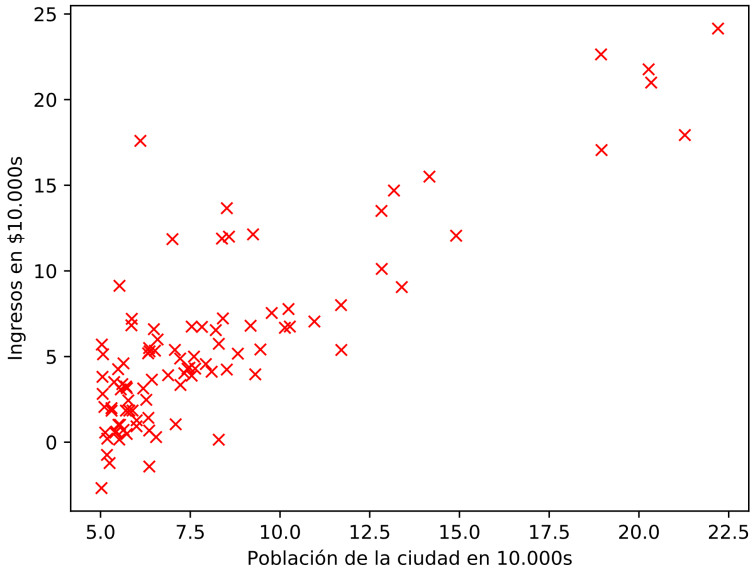
Índice

1. Descripción General

Esta práctica consta de dos partes. En la primera parte de la práctica se nos pedía aplicar el método de regresión lineal con una sola variable para estimar el beneficio de una compañía de distribución de comida en distintas ciudades, en base a su población.

La regresión lineal es un método matemático que en función de unos datos de entrada (datos de entrenamiento) se intenta estimar con un margen de error mínimo una posible solución. Esta estimación se realiza mediante el cálculo de una recta que se ajuste más a nuestros datos de entrenamiento.

En este caso se nos daba un archivo `ex1data1.csv` con los datos para desarrollar la solución:



En la segunda parte de la práctica se nos pedía aplicar el método de regresión lineal a los datos del archivo ex1data2.csv que contienen datos sobre el precio de casas vendidas en Portland, Oregon, incluyendo para cada casa el tamaño en pies cuadrados, el número de habitaciones y el precio.

Para ello vamos a resolverlo de dos formas, primeramente aplicando el método del descenso de gradiente generalizado y con el método de la ecuación normal que obtiene en un solo paso el valor óptimo para theta. Además comprobaremos que las soluciones que hemos obtenido son correctas comparándolas con algún ejemplo.

2. Explicación de la Solución

En este caso la práctica como hemos comentado anteriormente se divide en dos apartados, la regresión lineal con una sola variable y la regresión lineal con varias variables.

El código comienza con una llamada al método `main()` que contiene dos funciones, una por cada parte de la práctica. Vamos a comenzar explicando la primera parte.

```
def main():
    one_var_regression()
    mult_var_regression()
```

La función `one_var_regression()` comienza con la llamada a la función `load_csv()` que nos va a devolver, en un array de Numpy, los datos contenidos en el archivo csv que le indiquemos por parámetro. El código de esta función lo tenemos a continuación.

```
def one_var_regression():
    Data = load_csv('Práctica 1/Recursos/ex1data1.csv')

    X = Data[:, :-1]
    Y = Data[:, -1]
    X = np.hstack([np.ones([len(X), 1]), X])

    Theta, Costs = gradient_descent(X, Y, alpha=0.01)

    show_graph(X[:,1:], Y, Theta)
    show_costs(X, Y, Theta, Costs)
```

```
def load_csv(file_name):
    values = read_csv(file_name, header=None).values

    return values.astype(float)
```

Una vez tengamos los datos leídos lo que hacemos es separarlos en un array de `X` y un array de `Y`. Añadimos una columna de unos con a `X` para poder posteriormente hallar el valor de la hipótesis como el producto de dos vectores.

Una vez tengamos los datos listos vamos a calcular la theta óptima y el array de costes, para lo que llamamos a la función `gradient_descent()` que nos devolverá el Theta óptimo después de todas las iteraciones y un array de costes con el coste en cada iteración. Finalmente una vez tengamos `Theta` y `Cost` generamos las gráficas correspondientes llamando a las funciones `show_graph()` y `show_costs()` que comentaremos más adelante. Empezamos con la función `gradient_descent()`:

```
def gradient_descent(X, Y, alpha=0.3):
    theta0, theta1, Thetas, Costs = 0, 0, [], []

    while len(Costs) < 2 or Costs[len(Costs) - 2] - Costs[len(Costs) - 1] >
         0.00001:
        Theta = [theta0, theta1]

        theta0 = theta0 - alpha / len(X) * (np.dot(X, Theta) - Y).sum()
        theta1 = theta1 - alpha / len(X) * ((np.dot(X, Theta) - Y) *
             np.squeeze(X[:, 1:])).sum()

        Thetas.append([theta0, theta1])
        Costs.append(cost(X, Y, [theta0, theta1]))

    return [Thetas[len(Thetas) - 1], Costs]
```

Como sólamente tenemos una variable x sabemos el tamaño que van a tener los datos. En este caso necesitamos dos variables theta, `theta0` y `theta1`, inicializadas a 0, que vamos a ir actualizando en cada iteración. Creamos también un array de thetas para ir almacenando las thetas de cada iteración y un array de costes para lo mismo. El ejercicio al principio lo resolvimos utilizando un bucle con 1500 iteraciones (que es la solución que hemos dejado en la regresión de mútliples variables) pero luego lo adaptamos a la realidad y lo sustituimos por un bucle que deja de avanzar cuando la diferencia entre las dos últimas iteraciones es menor que un valor muy pequeño, en este caso 10^{-5}. En todo caso, si se quisiera volver a lo anterior habría que sustituir el bucle while por:

```
for i in range(1500):
```

El valor de α es un parámetro fijo que indica el tamaños de los "saltos" que realizará nuestro algoritmo en cada vuelta. En cada iteración se va a actualizar el valor de las thetas con el fin de ir minimizando la función de coste:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Actualizamos cada theta y llamamos a la función que nos calcula el coste:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

La función que nos hace esta operación se llama `cost()` y tiene la siguiente forma:

```
def cost(X, Y, Theta):
    H = np.dot(X, Theta)

    aux = (H - Y) ** 2

    return aux.sum() / (2 * len(X))
```

Por último guardamos este valor en el array de costes y el valor de la theta actual en el array de thetas. En el caso de hacer la solución con las 1500 iteraciones no necesitaríamos crear un array de thetas ya que al terminar las iteraciones tendremos los valores óptimos en `theta0` y `theta1`.

Ahora que ya tenemos todo lo necesario vamos a ir mostrando las gráficas. Primero mostramos la gráfica con la recta de regresión óptima y los casos de entrenamiento. La función se llama `show_graph()` y tiene esta forma:

```
def show_graph(X, Y, Theta, name='graph'):
    X_l = [np.amin(X), np.amax(X)]
    Y_l = [Theta[0] + Theta[1] * X_l[0], Theta[0] + Theta[1] * X_l[1]]

    plt.figure()
    plt.plot(X, Y, 'x', c='r')
    plt.plot(X_l, Y_l, c='b')
    plt.xlabel('Población de la ciudad en 10.000s')
    plt.ylabel('Ingresos en $10.000s')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)
```

Le pasamos por parámetro las coordenadas `X` e `Y` de los puntos que vamos a mostrar y las thetas para poder pintar la recta. Lo que hacemos es buscar los dos puntos máximo y mínimo en `X` y calcular su respectiva 'y' en la recta. De este modo ya tendremos los dos puntos extremos del segmento perteneciente a la recta de regresión que vamos a pintar. Como vemos pintamos los puntos como 'x' de color rojo y la recta de color azul. Con `xlabel()` e `ylable()` damos nombre a cada eje. Guardamos la gráfica en el la ruta deseada con el nombre que le pasamos por parámetro ('graph' por defecto) y con dpi le asignamos la resolución con la que queremos que guarde la imagen.

La siguiente función que llamamos es `show_costs()` que llama a otras tres funciones para representar otras gráficas.

```
def show_costs(X, Y, Theta, Costs):
    t0_range, t1_range = [-10, 10], [-1, 4]

    X, Y, Z = make_data(t0_range, t1_range, X, Y)

    show_2d_costs(Costs)
    show_3d_costs(X, Y, Z)
    show_contour_cost(X, Y, Z, Theta)
```

A esta función le llegan por parámetro `X`, `Y`, `Theta` y el array de costes `Cost`. Para representar las gráficas en 3D necesitamos preparar los datos, para ello llamamos a la función `make_data()` a la que le pasamos los rangos en los que queremos ver la función representada y los arrays `X` e `Y`. La función `make_data()` tiene esta forma:

```
def make_data(t0_range, t1_range, X, Y):
    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)

    Cost = np.empty_like(Theta0)

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

    return [Theta0, Theta1, Cost]
```

Esta función nos va a devolver la `Theta0`, `Theta1` y el `Cost` como matrices para poder representarlas en 3 dimensiones.

Una vez tenemos los datos listos podemos ir llamando a las funciones. La primera de ellas es `show_2d_cost()`:

```
def show_2d_costs(Costs, name='cost'):
    X = np.linspace(0, len(Costs), len(Costs))

    plt.figure()
    plt.plot(X, Costs, '.', c='r')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)
```

Como su propio nombre indica nos va a mostrar el coste en dos dimensiones, es decir, cómo va cambiando el coste a medida que avanzan las iteraciones. Para ello vamos a crear un array de coordenadas `X` llamando a la `funciónnp.linspace()` que nos va a devolver un array de puntos desde 0 hasta el numero de costes que haya con un tamaño del número de costes. Esto resulta en un punto por cada unidad en la gráfica. Por último mostramos los puntos, damos nombres a las coordenadas y guardamos la gráfica en la ruta especificada con el nombre `name` ('cost' por defecto) con dpi 200.

La siguiente función que llamamos es `show_3d_cost()` que tiene la siguiente forma:

```
def show_3d_costs(X, Y, Z, name='3d_cost'): # DONE
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0,
         antialiased=False)

    ax.view_init(elev=15, azim=230)
    fig.colorbar(surf, shrink=0.5, aspect=5)

    plt.xlabel('θ₀')
    plt.ylabel('θ₁')
    plt.locator_params(axis='x', nbins=5)
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)
```

Esta función nos va a mostrar el coste en 3 dimensiones. Es decir, va a mostrar por cada par de valores theta0 y theta1 un coste determinado. Para ello creamos el `plot_surface` pasándole las matrices de `X`, `Y` y `Z`. También le hemos colocado un barra de color para ver de forma visual como varía el valor de coste en una zona determinada de la gráfica. Por último guardamos la gráfica en la ruta especificada con el nombre `name` ('3d_cost por defecto').

La última función a la que llamamos es `show_contour_cost()`:

```
def show_contour_cost(X, Y, Z, Theta, name='contour_cost'):
    plt.figure()
    plt.contour(X, Y, Z, np.logspace(-2, 3, 20), cmap=cm.coolwarm)
    plt.plot(Theta[0], Theta[1], 'x', c='r')
    plt.xlabel('θ₀')
    plt.ylabel('θ₁')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)
```

Esta función nos va a mostrar el contorno de la gráfica. Le pasamos `X`, `Y` y `Z` a la función `contour()`. La hemos generado utilizando una escala logarítmica para el eje z, donde se representa J(theta), con 20 intervalos entre 0.01 y 100, lo que se consigue pasando `np.logspace(-2, 3, 20)` como cuarto argumento. Además se muestra el mínimo obtenido por el descenso de gradiente que hemos pintado con una 'x' y de color rojo. Por último guardamos la gráfica en la ruta especificada con el nombre `name` ('contour_cost' por defecto).

La segunda parte de la práctica vamos a resolver el problema de dos formas diferentes. Primero utilizando el método de regresión lineal con varias variables y luego utilizando el método de la ecuación normal. Comenzamos con la llamada a `mult_var_regression()`. Comenzamos leyendo los datos y separando las `X` y las `Y` y añadiendo la columna de unos a la `X` como vimos en la parte uno de la práctica:

```
def mult_var_regression():
    Data = load_csv('Práctica 1/Recursos/ex1data2.csv')

    X = Data[:, :-1]
    Y = Data[:, -1]
    X = np.hstack([np.ones([len(X), 1]), X])

    X_norm, Mu, Sigma = normalize(X[:, 1:])
    X_norm = np.hstack([np.ones([len(X_norm), 1]), X_norm])

    Th_grad, Cost = gradient_descent_mult(X_norm, Y, alpha=0.1)
    Th_norm = normal_ecuation_method(X, Y)

    check_ex(Th_grad, Th_norm, Mu, Sigma, [1.0, 1650, 3])
    show_2d_costs(Cost, name='mult_cost')
    show_diff_alpha(X_norm, Y)
```

A la hora de hacer el descenso de gradiente podemos tener variables con valores muy dispares al ser de distinta naturaleza, por ejemplo metro cuadrados y número de habitaciones, que es el caso de los datos de entrenamiento. Para solucionar esto tenemos que normalizar los datos. Vamos a generar unas `X_norm` con los valores de `X` normalizados, una variable `Mu` con las medias de cada variable de `X` y una variable `Sigma` con la desviación estándar de cada variable X. Para ello llamamos a la función `normalize()` que nos da este resultado (naturalmente obviando la primera columna de unos, que añadiremos después de obtener `X_norm`).

```
def normalize(X):
    Mu = np.mean(X, axis=0)
    Sigma = np.std(X, axis=0)
    X_norm = (X - Mu) / Sigma

    return [X_norm, Mu, Sigma]
```

Para hallar estos valores utilizamos las funciones `mean()` y `std()` de Numpy que nos calculan la media y la desviación respectivamente. Posteriormente hacemos este cálculo para normalizar los valores:

$$Z = \frac{x_i - \bar{X}}{\sigma}$$

Tenemos que devolver también `Mu` y `Sigma` ya que a la hora de hacer la prueba con nuevos datos tenemos que normalizarlos. Ahora si podemos llamar a `gradient_descent_mult()`:

```
def gradient_descent_mult(X, Y, alpha=0.3):
    Theta, Costs, m = np.zeros(len(X[0])), [], len(X)

    for i in range(1500):
        Aux = Theta.copy()

        Theta = Aux - alpha / m * (np.dot((np.dot(X, Aux) - Y), X))
```

```
        Costs.append(cost(X, Y, Theta))

    return [Theta, Costs]
```

Esta función nos va a resolver el mismo problema que la anterior pero para un caso más general con múltiples variables. Como vemos en este caso hemos utilizado un bucle for con 1500 iteraciones pero podríamos haber usado, como en el caso anterior, un bucle que parase cuando la diferencia entre las dos ultimas thetas fuese menor que un valor muy pequeño. Al haber hecho este cambio ya no necesitamos el vector de thetas. Realizamos la operación para actualizar todas las thetas y calculamos el coste llamando a la función `cost()` que es la misma que utilizábamos en la primera parte.

Una vez calculado el vector de costes y la theta vamos a calcular el la theta óptima utilizando la ecuación normal llamando a la función `normal_ecuation_method()`:

```
def normal_ecuation_method(X, Y):
    return np.matmul(np.matmul(np.linalg.pinv(np.matmul(np.transpose(X),
        X)), np.transpose(X)), Y)
```

Esta función nos va a devolver las thetas óptimas mediante la siguiente operación:

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

Una vez obtenido todo solo queda probar que nuestros cálculos han sido correctos. Para ello hemos creado una función llamada `check_ex()` que lo que va a hacer es, dado un ejemplo, calcular la predicción usando ambos métodos:

```
def check_ex(Th_grad, Th_norm, Mu, Sigma, X):
    X_norm = X.copy()
    X_norm[1:] = (X_norm[1:] - Mu) / Sigma

    print('Gradient descent result: {}'.format(np.dot(X_norm, Th_grad)))
    print('Normalized funcn result: {}'.format(np.dot(X, Th_norm)))
```

Esta función recibe las thetas calculadas mediante el descenso del gradiente y mediante la función normal, el vector de medias, el vector de desviaciones típicas y el ejemplo. Como hemos mencionado anteriormente para poder calcular el resultado de un ejemplo utilizando el método del descenso del gradiente tenemos que normalizarlo. Seguidamente imprimimos por consola ambos resultados y comparamos.

Por último queda mostrar las gráficas. Lo primero que hacemos es llamar a la función `show_2d_costs()` para mostrar la evolución del coste como en la parte uno. Después, como

indica el enunciado de la práctica, queremos mostrar para distintos valores de alfa la evolución del coste. Para esto hemos creado una función llamada `show_diff_alpha()`:

```
def show_diff_alpha(X, Y):
    Alphas = np.array([0.3, 0.1, 0.03, 0.01])
    Colors = np.array(['salmon', 'gold', 'violet', 'aquamarine'])

    plt.figure()

    for i in range(4):
        Theta, Cost = gradient_descent_mult(X, Y, Alphas[i])

        X_plot = np.linspace(0, len(Cost), len(Cost))

        plt.plot(X_plot, Cost, '.', c=Colors[i], label='α
             {}'.format(Alphas[i]))

    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.legend()
    plt.savefig('Práctica 1/Recursos/mul_alphas.png', dpi=200)
```

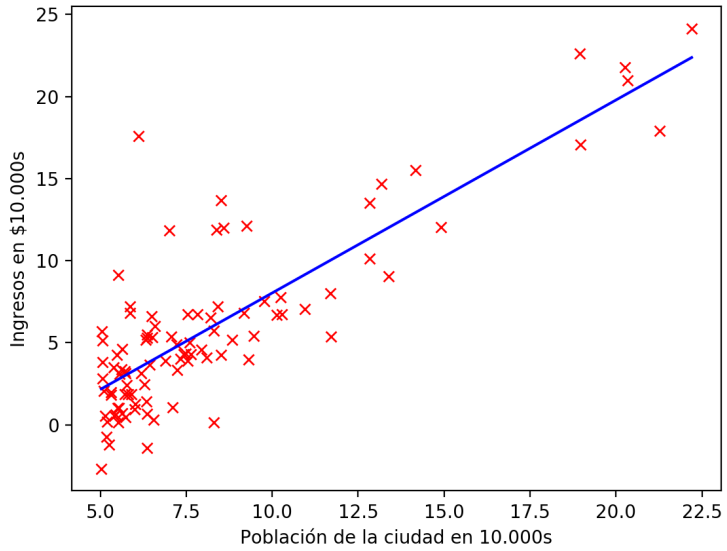
Lo que hacemos es crear dos arrays, uno llamado `Alphas` y otro llamado `Colors`. El primero tendrá los valores de alfa que queremos probar, y el segundo los colores que queremos asignar a cada alfa a la hora de dibujarlo. Mediante un bucle for vamos recorriendo el vector de alfas y calculando el vector de costes. Lo pintamos y pasamos a la siguiente. Finalmente nombramos los ejes y guardamos la gráfica en la ruta especificada.

Todas las gráficas y resultados de ejecución del código se muestran a continuación en el apartado resultados.

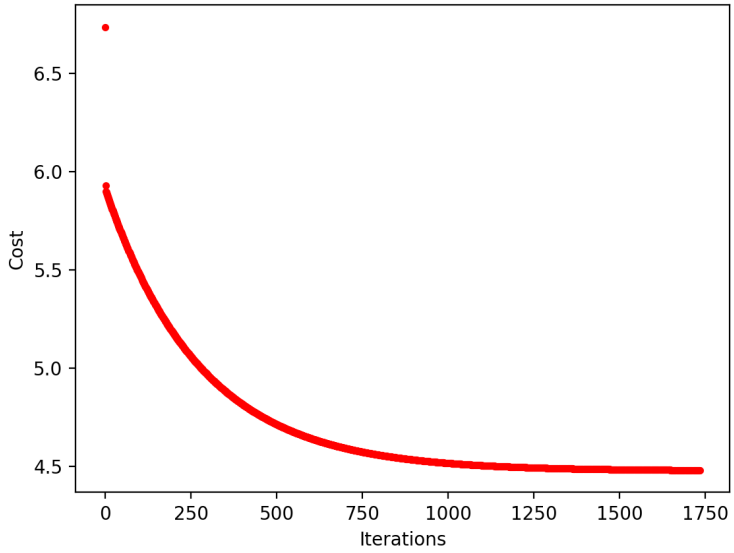
3. Resultados

En este apartado vamos a ir comentando los distintos resultados que hemos obtenido.

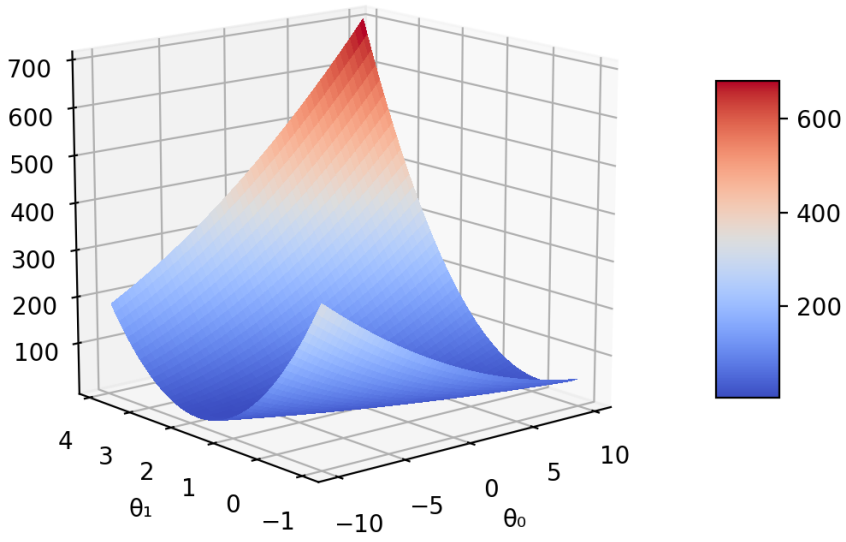
Vamos a comenzar con el primer apartado, correspondiente a la compañía de distribución de comida.



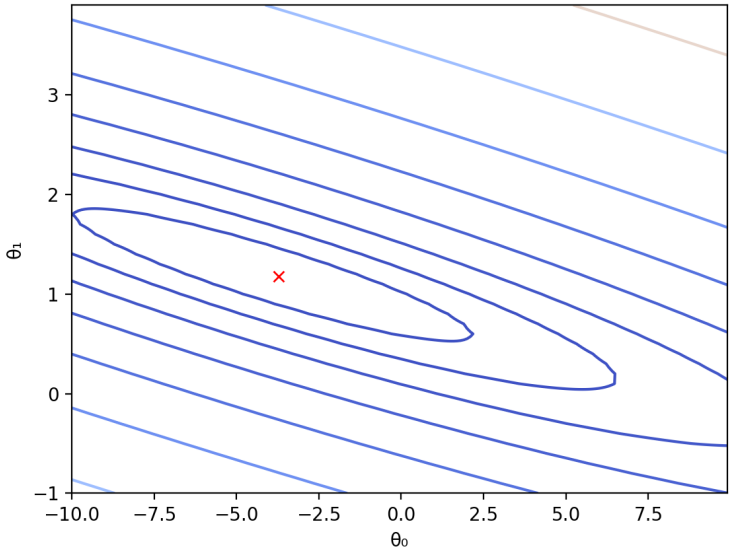
Tras iterar más de 1500 veces aplicando el descenso del gradiente obtuvimos que la mejor recta que se ajustaba a nuestros datos de entrenamiento fue la formada por los parámetros θ_0= -3.7217 y θ_1 = 1.17554. Como podemos ver en la siguiente gráfica el valor del coste va descendiendo por cada iteración:



Obtuvimos también una gráfica tridimensional en la que para cada valor de θ_0 y θ_1 se muestra su coste correspondiente en el eje Z:

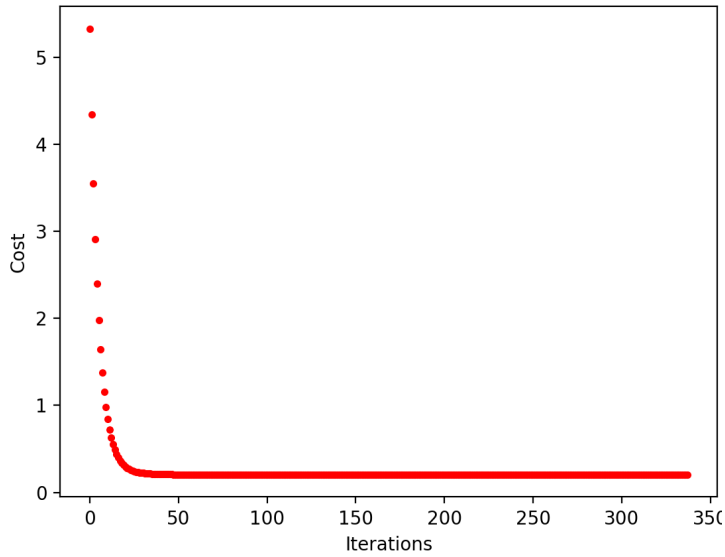


A su vez, gracias a las librerías y operaciones de python pudimos obtener un corte transversal de la gráfica en 3D en la que se puede ver efectivamente los valores óptimos de θ_0 y θ_1 escritos arriba que pintamos con una x en la gráfica:

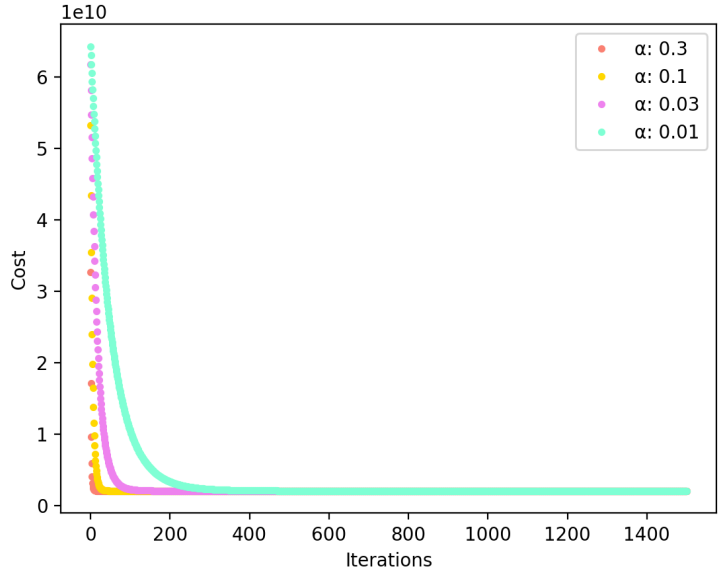


Ahora pasamos a comentar los resultados obtenidos en la segunda parte de la práctica.

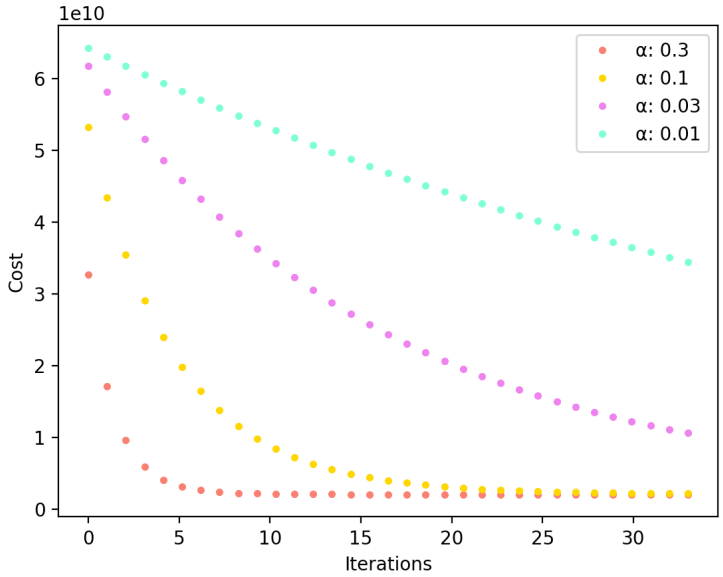
Mediante el descenso del gradiente obtuvimos para un $\alpha = 0.1$ y 1500 iteraciones la siguiente función de coste. Observamos además que habiendo utilizado el método de parar las iteraciones cuando la diferencia entre los dos últimos costes es menor que el anterior nos hubiese ahorrado muchas iteraciones, ya que la gráfica a partir de aproximadamente el valor 40 es prácticamente constante:



Como menciona el resultado, era conveniente mostrar la gráfica para distintos valores de α y compararlos. En nuestro caso lo hemos hecho para $\alpha = 0.3$, $\alpha = 0.1$, $\alpha = 0.03$ y $\alpha = 0.01$. A continuación se muestra la progresión del coste por cada iteración para cada α:



Para distinguir mejor la diferencia hemos imprimido una gráfica únicamente con 33 iteraciones:



Como podemos observar elegir un valor de α adecuado es importante para intentar que nuestro algoritmo sea lo más eficiente posible.

Por último hemos comprobado que nuestro código funcionaba probando con un ejemplo dado en el enunciado de la práctica. Al llamar a la función `check_ex()` con el ejemplo `[1650, 3]` el resultado que hemos obtenido ha sido el siguiente:

```
> check_ex(Th_grad, Th_norm, Mu, Sigma, [1.0, 1650, 3])
```

```
  Gradient descent result: 293081.46433489595
  Normalized funcn result: 293081.4643349892
```

Obtenemos para 1500 iteraciones prácticamente el mismo valor por lo que podemos concluir que nuestro código funciona correctamente.

4. Conclusión

En esta práctica hemos aprendido a realizar el descenso del gradiente tanto para una como para varias variables. Nos ha servido para acabar de familiarizarnos con el lenguaje Python y sus librerías así como la representación de funciones tanto en dos como en tres dimensiones.

En el siguiente apartado se muestra el código completo por si se quiere ver seguido o hacer alguna prueba compilada.

5. Código

```
import numpy as np
import matplotlib.pyplot as plt
from pandas.io.parsers import read_csv
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import random

def load_csv(file_name):
    values = read_csv(file_name, header=None).values

    return values.astype(float)

def cost(X, Y, Theta):
    H = np.dot(X, Theta)

    aux = (H - Y) ** 2

    return aux.sum() / (2 * len(X))

def make_data(t0_range, t1_range, X, Y):
    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)

    Cost = np.empty_like(Theta0)

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

    return [Theta0, Theta1, Cost]

def show_contour_cost(X, Y, Z, Theta, name='contour_cost'):
    plt.figure()
    plt.contour(X, Y, Z, np.logspace(-2, 3, 20), cmap=cm.coolwarm)
    plt.plot(Theta[0], Theta[1], 'x', c='r')
    plt.xlabel('θ₀')
    plt.ylabel('θ₁')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)

def show_3d_costs(X, Y, Z, name='3d_cost'):
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0,
antialiased=False)

    ax.view_init(elev=15, azim=230)
    fig.colorbar(surf, shrink=0.5, aspect=5)

    plt.xlabel('θ₀')
    plt.ylabel('θ₁')
    plt.locator_params(axis='x', nbins=5)
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)

def show_2d_costs(Costs, name='cost'):
```

```
    X = np.linspace(0, len(Costs), len(Costs))

    plt.figure()
    plt.plot(X, Costs, '.', c='r')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)

def show_costs(X, Y, Theta, Costs):
    t0_range, t1_range = [-10, 10], [-1, 4]

    X, Y, Z = make_data(t0_range, t1_range, X, Y)

    show_2d_costs(Costs)
    show_3d_costs(X, Y, Z)
    show_contour_cost(X, Y, Z, Theta)

def show_graph(X, Y, Theta, name='graph'):
    X_l = [np.amin(X), np.amax(X)]
    Y_l = [Theta[0] + Theta[1] * X_l[0], Theta[0] + Theta[1] * X_l[1]]

    plt.figure()
    plt.plot(X, Y, 'x', c='r')
    plt.plot(X_l, Y_l, c='b')
    plt.xlabel('Población de la ciudad en 10.000s')
    plt.ylabel('Ingresos en $10.000s')
    plt.savefig('Práctica 1/Recursos/{}.png'.format(name), dpi=200)

def gradient_descent(X, Y, alpha=0.3):
    theta0, theta1, Thetas, Costs = 0, 0, [], []

    while len(Costs) < 2 or Costs[len(Costs) - 2] - Costs[len(Costs) - 1] >
0.00001:
        Theta = [theta0, theta1]

        theta0 = theta0 - alpha / len(X) * (np.dot(X, Theta) - Y).sum()
        theta1 = theta1 - alpha / len(X) * ((np.dot(X, Theta) - Y) *
np.squeeze(X[:, 1:])).sum()

        Thetas.append([theta0, theta1])
        Costs.append(cost(X, Y, [theta0, theta1]))

    return [Thetas[len(Thetas) - 1], Costs]

def one_var_regression():
    Data = load_csv('Práctica 1/Recursos/ex1data1.csv')

    X = Data[:, :-1]
    Y = Data[:, -1]
    X = np.hstack([np.ones([len(X), 1]), X])

    Theta, Costs = gradient_descent(X, Y, alpha=0.01)

    show_graph(X[:,1:], Y, Theta)
    show_costs(X, Y, Theta, Costs)


def check_ex(Th_grad, Th_norm, Mu, Sigma, X):
    X_norm = X.copy()
    X_norm[1:] = (X_norm[1:] - Mu) / Sigma
```

```
    print('Gradient descent result: {}'.format(np.dot(X_norm, Th_grad)))
    print('Normalized funcn result: {}'.format(np.dot(X, Th_norm)))

def normalize(X):
    Mu = np.mean(X, axis=0)
    Sigma = np.std(X,axis=0)
    X_norm = (X - Mu) / Sigma

    return [X_norm, Mu, Sigma]

def gradient_descent_mult(X, Y, alpha=0.3):
    Theta, Costs, m = np.zeros(len(X[0])), [], len(X)

    for i in range(1500):

        Aux = Theta.copy()

        Theta = Aux - alpha / m * (np.dot((np.dot(X, Aux) - Y), X))

        Costs.append(cost(X, Y, Theta))

    return [Theta, Costs]

def normal_ecuation_method(X, Y):
    return np.matmul(np.matmul(np.linalg.pinv(np.matmul(np.transpose(X), X)),
np.transpose(X)), Y)

def show_diff_alpha(X, Y):
    Alphas = np.array([0.3, 0.1, 0.03, 0.01])
    Colors = np.array(['salmon', 'gold', 'violet', 'aquamarine'])

    plt.figure()

    for i in range(4):
        Theta, Cost = gradient_descent_mult(X, Y, Alphas[i])

        X_plot = np.linspace(0, len(Cost), len(Cost))

        plt.plot(X_plot, Cost, '.', c=Colors[i], label='α:
{}'.format(Alphas[i]))

    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.legend()
    plt.savefig('Práctica 1/Recursos/mul_alphas.png', dpi=200)

def mult_var_regression():
    Data = load_csv('Práctica 1/Recursos/ex1data2.csv')

    X = Data[:, :-1]
    Y = Data[:, -1]
    X = np.hstack([np.ones([len(X), 1]), X])

    X_norm, Mu, Sigma = normalize(X[:, 1:])
    X_norm = np.hstack([np.ones([len(X_norm), 1]), X_norm])

    Th_grad, Cost = gradient_descent_mult(X_norm, Y, alpha=0.1)
    Th_norm = normal_ecuation_method(X, Y)
```

```
    check_ex(Th_grad, Th_norm, Mu, Sigma, [1.0, 1650, 3])
    show_2d_costs(Cost, name='mult_cost')
    show_diff_alpha(X_norm, Y)


def main():
    one_var_regression()
    mult_var_regression()


main()
```