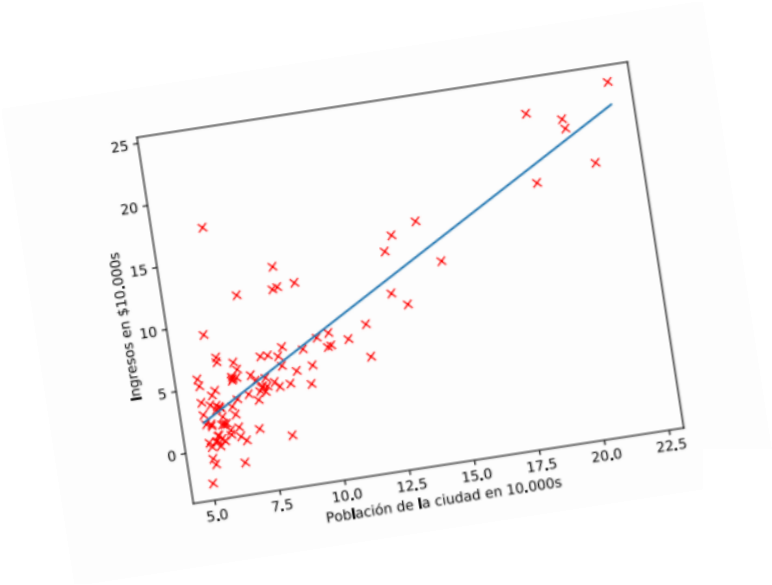
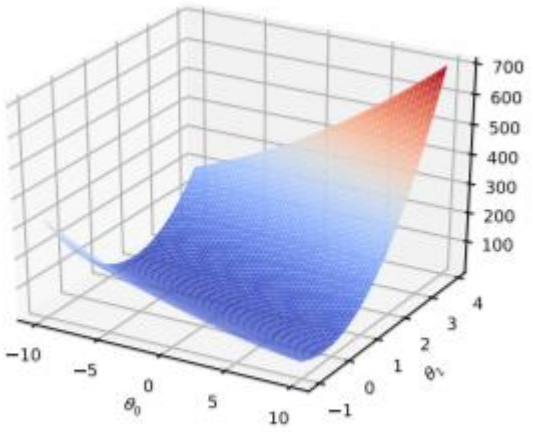
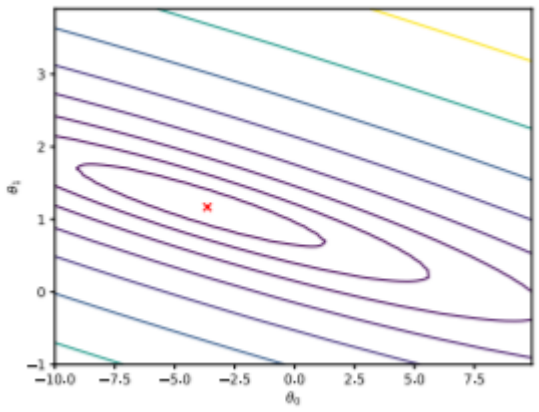
REGRESIÓN LINEAL:

A) CON UNA VARIABLE

B) CON VARIAS VARIABLES







Realizado por:
Montserrat Sacie Alcázar y
Tomás Golomb Durán
Universidad Complutense de Madrid
Asignatura: Aprendizaje Automático y Big Data
Profesor: Pedro Antonio González Calero
Curso: 2018 - 2019

A) REGRESIÓN LINEAL CON UNA VARIABLE

En esta primera parte de la práctica, debemos aplicar el método de regresión lineal para aproximar el beneficio obtenido de una empresa de comida en relación a la población de una ciudad (variable única de la que depende el beneficio).

Para la resolución en python de este problema se utilizarán las librerías numpy y matplotlib para realizar operaciones vectorizadas y dibujar las gráficas correspondientes.

```
def gradiente(alpha, filename, numiteraciones):
    datos = np.array(lee_csv(filename))

    teta0 = 0
    teta1 = 0
    yvect = datos[0:, 1]
    xvect = datos[0:, 0]
    vTeta0 = [teta0]
    vTeta1 = [teta1]
    for i in range (0, numiteraciones):
        hvect = teta0 + teta1* xvect
        vresta = hvect - yvect
        teta0aux = teta0 - alpha * (1/ float(datos.size)) * np.sum(vresta)
        teta1aux = teta1 - alpha *(1/float(datos.size)) * np.sum(vresta * xvect)
        teta0 = teta0aux
        teta1 = teta1aux
        vTeta0 += [teta0]
        vTeta1 += [teta1]

    grafica2d(datos, teta0, teta1, xvect, yvect)
    graficas3d(datos, vTeta0, vTeta1)

    return [teta0, teta1]
```

Función gradiente

Esta función que recibe el ratio de aprendizaje (α), el archivo con los ejemplos de aprendizaje (*filename*) y el número de iteraciones (*numiteraciones*). Antes de empezar con el algoritmo de descenso del gradiente inicializamos los *θ* de la función *h(x)* que se desea calcular a 0. Además, extraemos de la matriz de datos la variable *x* (población de la ciudad) e *y* (ingresos) para poder aplicar operaciones vectorizadas y de esta forma reducir el coste computacional del cálculo. Por último, inicializamos dos vectores que llevarán el historial del "refinamiento" de los *θ* con el fin de comprobar y representar el funcionamiento del algoritmo.

El objetivo del algoritmo es hallar la recta de regresión *h(x)* formada por los *θ* que reducen el error que produce esta sobre los ejemplos de entrenamiento y que viene descrita por la función coste *J(θ) (media de la diferencia del coste al cuadrado).* Con esta recta se podrán realizar predicciones sobre el ingreso medio de la población dado el número de ciudadanos de un área urbana. Para ello, se deben ir ajustando los *θ* (*θ0 y θ1*) restando la derivada parcial de la función coste por el ratio de aprendizaje. Para calcular el coste de estas derivadas de forma óptima realizamos operaciones vectorizadas básicas de la librería numpy (+, -, *, sum).

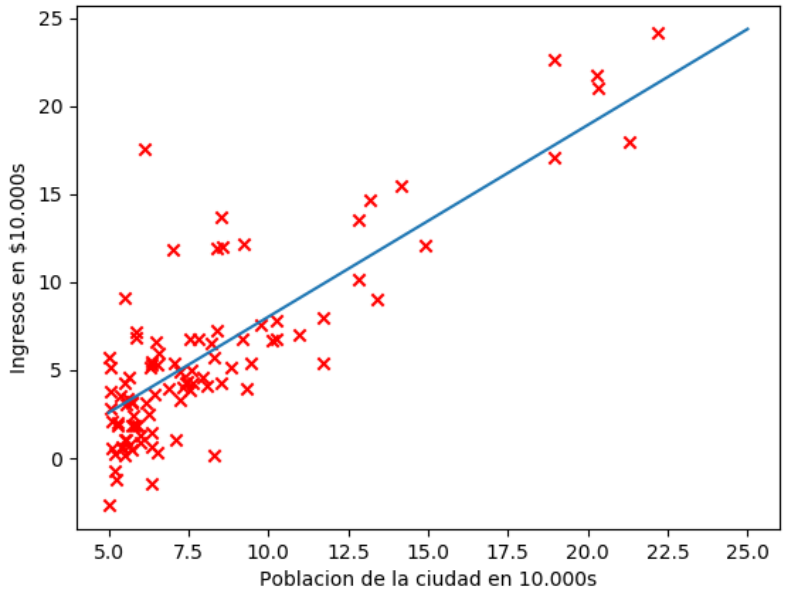
Por último, dibujamos los gráficos de la solución y devolvemos un par con el historial de los *θ* calculados.

```
def grafica2d(datos, teta0, teta1, xvect, yvect):
    pl.figure()
    pl.xlabel('Poblacion de la ciudad en 10.000s')
    pl.ylabel('Ingresos en $10.000s')
    pl.scatter(np.array(xvect), np.array(yvect), marker='x', c='red')
    pl.plot(np.array([5, 25]), np.array([teta0+teta1*5, teta0+teta1*25]), '-')
    pl.savefig('grafica2d.png')
```

Función grafica2d

Se dibuja la gráfica 2D con la librería matplotlib que contiene los ejemplos de aprendizaje (*xvect* e *yvect)* y la recta de regresión *h(x) = θ0 + θ1*x*

Resultado:

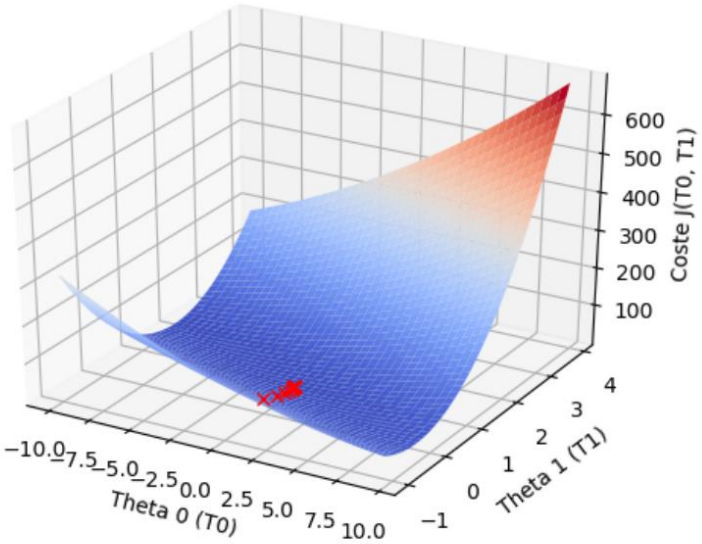


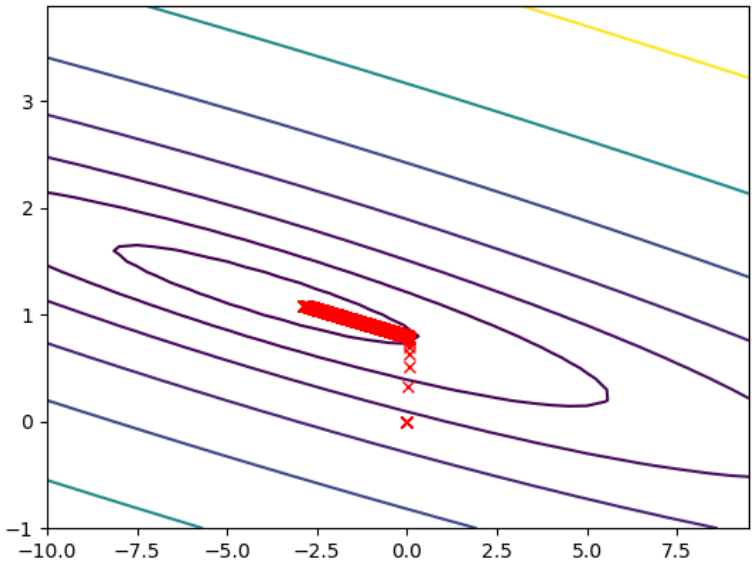

```
def graficas3d(datos, vT0, vT1):
    fig = pl.figure()
    ax = fig.gca(projection='3d')
    X = np.arange(-10, 10, 0.5)
    Y = np.arange(-1, 4, 0.1)
    X, Y = np.meshgrid(X, Y)
    Z = X * 0
    datosX = datos[0:, 0]
    datosY = datos[0:, 1]
    for i in range (0,len(X)):
        for j in range (0, len(X[i])):
            Z[i][j]= coste(X[i][j], Y[i][j], datosX, datosY, len(datosX))
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=True)
    for i in range(0, 50):
        z = coste(vT0[i], vT1[i], datosX, datosY, len(datosX))
        pl.plot([vT0[i]], [vT1[i]], [z], marker='x', c="red")

    pl.savefig('grafica_surface.png')
    pl.figure()
    pl.contour(X, Y, Z, np.logspace(-3, 3, 20))
    for i in range(0, len(vT0)):
        z = coste(vT0[i], vT1[i], datosX, datosY, len(datosX))
        pl.plot([vT0[i]], [vT1[i]], [0], marker='.', c="red")
    pl.savefig('grafica_contour.png')
```

Función grafica3d

Representación del algoritmo mediante el dibujo de la gráfica de coste en 3D y en un mapa topográfico. Los gráficos contienen el dibujo de la

función coste y el cálculo de este para el historial de tetas explicado anteriormente (*vT0* y *vT1*) que demuestran el descenso al mínimo.
Resultado:






```python
def coste(teta0, teta1, datosX, datosY, size):
    vH = (teta0 + teta1 *datosX)
    return (1/float(2*size)) * np.sum((vH - datosY)**2)
```

Función coste
Cálculo de la función coste J(θ) dado un par de tetas y el conjunto de datos X e Y utilizando operaciones vectorizadas.

B) REGRESIÓN LINEAL CON VARIAS VARIABLES

Imports:

```
import matplotlib.pyplot as pl
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
import numpy as np
from pandas.io.parsers import read_csv
```

En la segunda parte de la práctica tendremos que aplicar el método de regresión lineal para estimar el precio de las casas (variable y), en función del tamaño(variable x1) y el número de habitaciones (variable x2).
Los datos los leemos en la función:

```
def lee_csv(a):
    valores = read_csv(a,header = None).values
    return valores.astype(float)
```

Función descensoGradiente

En este caso tenemos dos variables que representan los datos de entrada de nuestro algoritmo, pero podríamos tener más variables que ayuden a estimar el precio. Así, implementamos una función *descensoGradiente* genérica que permita calcular los coeficientes *θ,* en un bucle aplicando el descenso de gradiente.
Como **parámetros** recibe un α, los datos normalizados guardados en una matriz(cada fila de *datosN* es un ejemplo de entrenamiento con los valores de las xi normalizados y la columna *y* borrada de la matriz), un vector *y* con todos los precios exactos que corresponde a cada fila de *datosN* y el número de iteraciones del bucle *for* , en el que actualizamos los *θ.*
En el vector vthetas guardamos varias tuplas de *θ calculados para comprobar* como disminuye su valor. Como se realizan demasiadas

```
def descensoGradiente(alpha, datosN, y, numiteraciones):
    thetas = np.zeros(len(datosN[0]))
    vthetas = []
    for i in range (0, numiteraciones):
        hMat = thetas * datosN
        sumMat = np.dot(hMat, np.ones((len(hMat[0]), 1)))
        mResta = sumMat - y
        mResta = mResta.transpose()
        mResta = np.dot(mResta, datosN)
        sumatorios = alpha/float(datosN.size) * mResta
        if (i % 100 == 0):
            vthetas += [thetas]
        thetas = thetas - sumatorios
    print(len(vthetas))
    return vthetas
```

iteraciones, decidimos guardar cada 100 iteraciones el valor en ese momento de los θ, dando lugar a un vector de 15 elementos que usaremos para dibujar la gráfica .

Función normalizar

Como función auxiliar para normalizar los datos de entrada y permitir que los cálculos de las h(*θ),* para cada ejemplo de entrenamiento, sean más precisos (los θ no disten tanto del rango de datos) usamos la siguiente función:

```
def normalizar(datos):
    mu = []
    sigma = []
    for j in range(0, len(datos[0])):
        mu += [np.mean(datos[0:, j])]
        sigma += [np.std(datos[0:,j])]
        datos[:,j] = (datos[0:,j]-mu[j])/sigma[j]
    return (datos, mu, sigma)
```

En la función anterior llamamos *mu* a la media de los valores que toma cada variable *xi* en todos los ejemplos de entrenamiento y *sigma* es la desviación típica de los mismos

Devolvemos los datos normalizados y también mu y sigma para usarlos posteriormente en la normalización de datos de entrada de nuestro programa de los que queramos estimar el valor de *y* aproximado.

Función predicción

La función que aproxima un valor de *y* para unos valores de las variables de entrada cualesquiera es:

```
def prediccion(vars, mu, sigma, thetas):
    vars = (vars - mu) / sigma
    prediccion = sum(vars * thetas)
    return prediccion
```

Función gradienteOptimo

En la función *descensoGradiente* implementamos una forma de calcular los θ que implica varios cálculos con matrices, un buble for y no es tan eficiente. Usando el método de la ecuación normal en la función *gradienteÓptimo,* vemos otra forma de hacer los cálculos de las y, más eficiente:

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

La función recibe como parámetros los datos de ejemplos de entrenamiento y las *y* (no normalizados) y devuelve los valores de las *θ*

```
def gradienteOptimo(datos, y):
    transpose = datos.transpose()
    thetas = np.dot(np.dot(np.linalg.pinv(np.dot(transpose, datos)), transpose), y)
    return thetas
```

Función comparaPredicciones

En la función *comparaPredicciones,* recibimos como entrada un ejemplo de datos, calculamos los θ que hacen el coste mínimo tanto con *descensoGradiente* como con *gradienteOptimo* y finalmente imprimimos los resultados de las predicciones de la *y* en ambos casos; comprobando que el resultado es el mismo y no afecta el método empleado en la corrección de la predicción.

```
def comparaPredicciones(datos, y, mu, sigma, alpha, filename, numiteraciones):
    datos = np.delete(datos, len(datos[0])-1, 1)
    datos = np.c_[np.ones(len(datos)), datos]
    thetasOpt = gradienteOptimo(datos, y)
    thetas = descensoGradiente(alpha, filename, numiteraciones)
    print([0] + [mu])
    print prediccion([1, 1650, 3], np.array([0] + mu), np.array([1] + sigma), thetas)
    print prediccion(np.array([1,2104,3]), np.array([0, 0, 0]), np.array([1, 1, 1]), thetasOpt.transpose()[0])
```

Función coste

Una vez obtenidos los parámetros *θ* óptimos, calculamos los costes mediante:

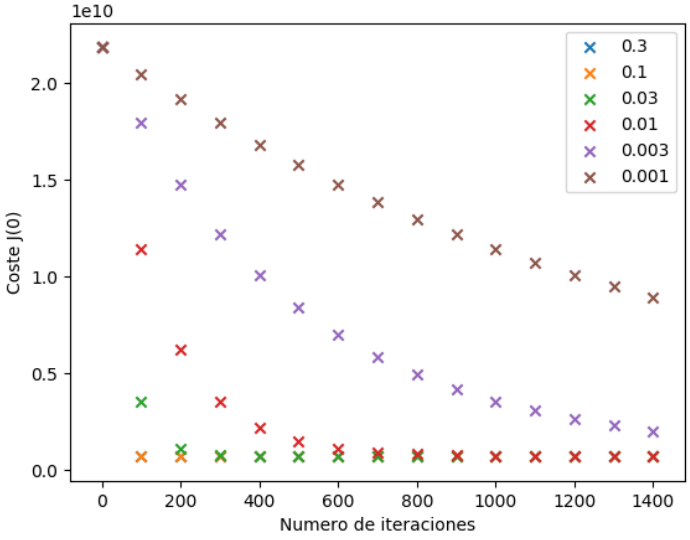
```
def coste(datos, y, thetas):
    hMat = thetas * datos
    sumMat = np.dot(hMat, np.ones((len(hMat[0]), 1)))
    mResta = sumMat - y
    return (1/(2*datos.size))*np.dot(mResta.transpose(), mResta)
```

Función comprobacion

Tras implementar los algoritmos para resolver nuestro problema, definimos la función principal *comprobación.* En ella leemos los datos de fichero, normalizamos los datos y preparamos la llamada a *descensoGradiente* para diferentes valores de α. A la vez que hacemos los cálculos para cada α, representamos en una gráfica como desciende el coste para *θs* más ajustados (mientras más iteraciones actualizando los *θ,* más ajustados son sus valores) .

```
def comprobacion(filename, numIteraciones):
    datos = np.array(lee_csv(filename))
    datosX = np.delete(datos, len(datos[0])-1, 1)
    y = np.array([datos[:,len(datos[0]) - 1]])
    y = y.transpose()
    norm = normalizar(datosX)
    datosN = np.c_[np.ones(len(datosX)), datosX]
    pl.figure()
    pl.xlabel('Numero de iteraciones')
    pl.ylabel('Coste J(0)')
    for alpha in [0.3, 0.1, 0.03, 0.01, 0.003, 0.001]:
        vthetas = descensoGradiente(alpha, datosN, y, numIteraciones)
        vCostes = []
        for i in range (0,len(vthetas)):
            vCostes += [coste(datosN, y, vthetas[i])]
        print(len(np.linspace(0,numIteraciones, numIteraciones/100, endpoint = False)))
        print(len(vCostes))
        pl.scatter(np.linspace(0,numIteraciones, numIteraciones/100, endpoint = False), vCostes, marker = 'x', label = alpha)
    pl.legend()
    pl.savefig('comprobacionFinal.png')
```

La gráfica resultante es la siguiente:



(Cada curva de un color está asociada a un α)