Machine Learning

Assignment 2: LINEAR REGRESSION /OPTIMIZATION

Course Instructor: Dr. Wajahat Hussain

Prepared by: Asad khan
Email: akhan.msee19seecs@seecs.edu.pk

Submitted By:

Muhammad Hamza Shakoor
Reg No. 255633

ML Group 1
BEE 10 B
SEECS NUST

Tutorial/ Report

Introduction:

In this exercise I implemented linear regression and gott to see it work on data. Before starting on this programming exercise, we recommend clearing concepts of linear regression, gradient descent and feature normalization

Files included in this exercise :

- ex1P1.ipyb – Google Colab Notebook (gradient descent) script that steps you through the exercise
- ex1P2.ipyb – Google Colab Notebook (Optimization Function) script that steps you through the exercise
- ex1data1.txt – Dataset Text file for linear regression with one variable

Throughout the exercise, you will be using the scripts ex1P1.ipyb and ex1P2.ipyb. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them.

You are only required to modify functions, by following the instructions in this assignment. For this programming exercise, you are only required to complete the first part of the exercise to implement linear regression with one variable.

The second part of the exercise, which is optional, covers linear regression with multiple variables.

Overview:

- Assignment comprises of two parts, first part simply implements linear regression model while second part optimizes cost function by calculating its numerical minimum.
- In part 1, Estimated cost is given, and computed cost is calculated using the program and we have to compare these.
- Gradient descent finds minimum value of cost to find values of thetas.
- Alpha is learning rate as it decides step size.
- For part 2, optimization is implemented using 2 functions of Scikit library.
- Both of these functions use different algorithms

PART 1

1. Simple Python function The first part of ex1P1.ipyb gives you practice with function syntax. In the the warmUpExercise, you will find a predefined function in given space to return a 5x5 or any other value identity matrix you should see output similar to the following:

```
[ ]  ##==================== Part 1: Basic Function ====================
     #Complete warmUpExercise

     # 5x5 matrix with 1's on main diagnol

     import numpy as np
     def iden(a):
       # function definition
       # ==========YOUR CODE HERE============
       #a 2D zero matrix of given number a
       _2d_zero_matrix=np.zeros((a,a))
       for i in range(a):
         for j in range (a):
           if(i==j):
             _2d_zero_matrix[i,j]=1
       print(_2d_zero_matrix)
      # ==================================

     print("Matrix a : \n")
     iden(5)  # function call
```

```
Matrix a :

[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Simple 5 by 5 matrix with all entries zeroes, except diagonal entries are 1

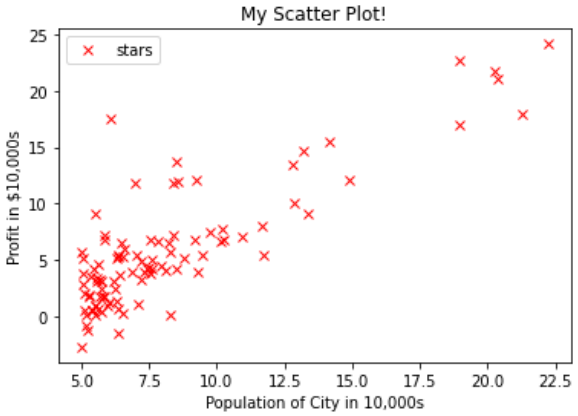
2. Linear regression with one variable

Plotting

```
[ ]  ## ======================== Part 2: Plotting ========================

     import matplotlib.pyplot as plt

     # used for manipulating directory paths
     import os


     # Read comma separated data
     data = np.loadtxt(os.path.join('Data', path ), delimiter=',')
     X, Y = data[:, 0], data[:, 1]


     # Plot Data
     # Note: You have to complete the code in plotData function

     def plotdata(a,b):                          # function def
       # plotting points as a scatter plot
       # ==========YOUR CODE HERE============
       plt.plot(a,b,"rx")
       plt.xlabel('Population of City in 10,000s')
       plt.ylabel('Profit in $10,000s')
       # displaying the title
       plt.title("My Scatter Plot!")
       plt.legend(['stars'])
       # ====================================
```

```
plotdata(X,Y)  # function Call
plt.show()
#In the dataset file, The first column is the population of a city and the second column is the profit of a food truck in that city
```

- This code only plots the already available data of ex1data1.txt file,
- X is population of city in 10,000s
- Y is Profit in $10,000s



```
m = Y.size  # number of training examples
X = np.stack([np.ones(m), X], axis=1) # it used to convert X in to (97
print(X.shape)
# Do NOT execute this cell more than once.
```

- It used to convert X in to (97x2), first column is all ones to get where theta is (2x1). So that dot product of theta and X can be possible.
- theta[0]+theta[1]*X

LINEAR REGRESSION:

Cost Function:

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

the hypothesis h(x) is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Compute Cost function:

```
m = Y.size  # number of training examples
X = np.stack([np.ones(m), X], axis=1) # it used to convert X in to (97x2), first colum is all ones
print(X.shape)
```

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

```
h = np.dot(X, theta)
```

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

Code of cost function J:

```
[ ]
import numpy as np
from numpy import *

def computeCost(X,y , theta):
  m = y.size
  J = 0
  h = np.dot(X, theta)
  # ==========YOUR COST FUNCTION J HERE============
  #squared difference of estimated values and real values of y
  h_theta=h
  J_theta=sum((h_theta-y)**2)
  J_theta=J_theta/(2*m)

 # ===================================
  return J_theta

J = computeCost(X, Y, theta=np.array([0.0, 0.0]))
print('With theta = [0, 0] \nCost computed =', J)
print('Expected cost value (approximately) 32.07\n')
```

Output:

```
With theta = [0, 0]
Cost computed = 32.072733877455676
Expected cost value (approximately) 32.07
```

It is shown that estimated cost and expected cost are same i.e 32.07
So our algorithm is good.

Gradient Descent:

Optimizes cost so that J becomes minimum

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

Code of Gradient Descent Algorithm:

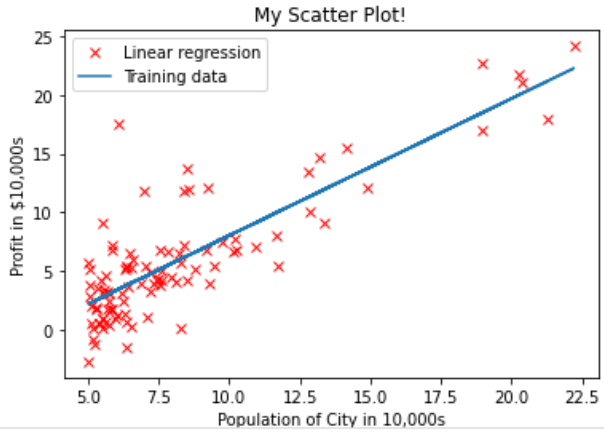
```
def gradientDescent(X, y, theta, alpha, num_iters):
  m = y.shape[0]
  theta = theta.copy()
  J_history = []
  for i in range(num_iters):
    # ==========YOUR GRADIENT DESCENT "theta" HERE============
    h=np.dot(X, theta)
    theta=theta-((alpha)*(1/m)*(X.T.dot(h-y)))

    # ====================================
    J_history.append(computeCost(X, y, theta))


  return theta, J_history
```

Some settings for gradient descent:

```
#=============== initialize fitting parameters
theta = np.zeros(2)
print(theta.shape)
# ==================some gradient descent settings
iterations = 1500
alpha = 0.01

theta, J_history = gradientDescent(X ,Y, theta, alpha, iterations)
print(J_history)


# plot the linear fit
plotdata(X[:, 1],Y)
plt.plot(X[:, 1], np.dot(X, theta))

plt.legend([ 'Linear regression','Training data',]);
```

Output:

```
(2,)
[6.737190464870006, 5.9315935686049555, 5.901154707081388, 5.895228586444221, 5.8900949431173295,
```



2D VISUALIZATION

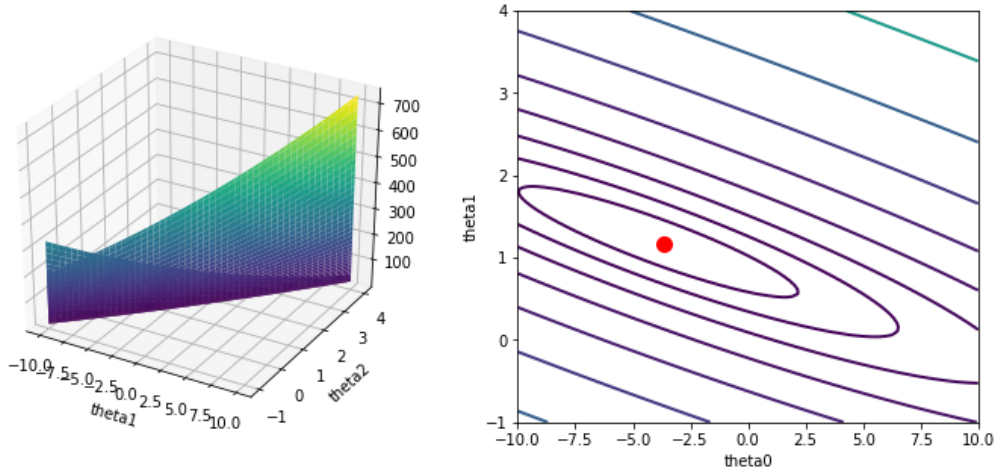
- To understand the cost function J(θ) better, you will now plot the cost over a 2-dimensional grid of θ0 and θ1values.
- You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images. Plot the training data and the linear regression: VISUALIZATION:
- In the next cell, the code is set up to calculate J(θ) over a grid of values using the computeCost function that you wrote.
- After executing the following cell, you will have a 2-D array of J(θ) values. Then, those values are used to produce surface and contour plots of J(θ) using the matplotlib plot surface and contour functions. The plots should look something like the following:

```
[ ]  # ==================================NO NEED CHANGE CODE IN THIS CELL
     # Plotting library
     import matplotlib.pyplot as plt
     from mpl_toolkits.mplot3d import Axes3D  # needed to plot 3-D surfaces

     # grid over which we will calculate J COST
     t0 = np.linspace(-10, 10, 100)
     t1 = np.linspace(-1, 4, 100)

     # initialize J_vals to a matrix of 0's
     J_vals = np.zeros((t0.shape[0], t1.shape[0]))

     # Fill out J_vals
     for i, theta0 in enumerate(t0):
         for j, theta1 in enumerate(t1):
             J_vals[i, j] = computeCost(X,Y, [theta0, theta1])

     # Because of the way meshgrids work in the surf command, we need to
     # transpose J_vals before calling surf, or else the axes will be flipped
     J_vals = J_vals.T
```

```
# Because of the way meshgrids work in the surf command, we need to
# transpose J_vals before calling surf, or else the axes will be flipped
J_vals = J_vals.T

# surface plot
fig = plt.figure(figsize=(12, 5))
ax = fig.add_subplot(121, projection='3d')
ax.plot_surface(t0, t1, J_vals, cmap='viridis')
ax.set_xlabel('theta1')
ax.set_ylabel('theta2')


# contour plot
# Plot J_vals as 15 contours spaced logarithmically between 0.01 and 100
ax2 = plt.subplot(122)
ax2.contour(t0, t1, J_vals, linewidths=2, cmap='viridis', levels=np.logspace(-2, 3, 20))
ax2.set_xlabel('theta0')
ax2.set_ylabel('theta1')
ax2.plot(theta[0], theta[1], 'ro', ms=10, lw=2)

pass
```

We did not change a thing in this part, because it was optional task and just for understanding of 2D gradient descent plotting.

Output:



FEATURE NORMALIZE:

Code :

```
]
    # Read comma separated data
    data = np.loadtxt(os.path.join('Data', path ), delimiter=',')
    X, Y = data[:, 0], data[:, 1]


    def  featureNormalize(X):

    #====================YOUR CODE HERE=========
      mean=np.mean(X)
      mean_diff=X-mean
      standard_deviation=np.std(X)
      X_norm=mean_diff/standard_deviation
      mu=mean
      sigma=standard_deviation
    #==========================================
      return X_norm, mu, sigma

    X, mu, sigma = featureNormalize(X)
    Y, mu, sigma = featureNormalize(Y)

    X = np.stack([np.ones(m), X], axis=1)
```

- This formula is implemented:
- ***X_norm = mean difference(X- mean)/ standard deviation of X***
- This formula is for normalizing errors when population parameters are known.
- Works well for populations that are **normally distributed.**

- **Normalization** means adjusting values measured on different scales to a notionally common scale, often prior to averaging.
- In more complicated cases, normalization may refer to more sophisticated adjustments where the intention is to bring the entire distribution of adjusted values into alignment.

Learning rates:

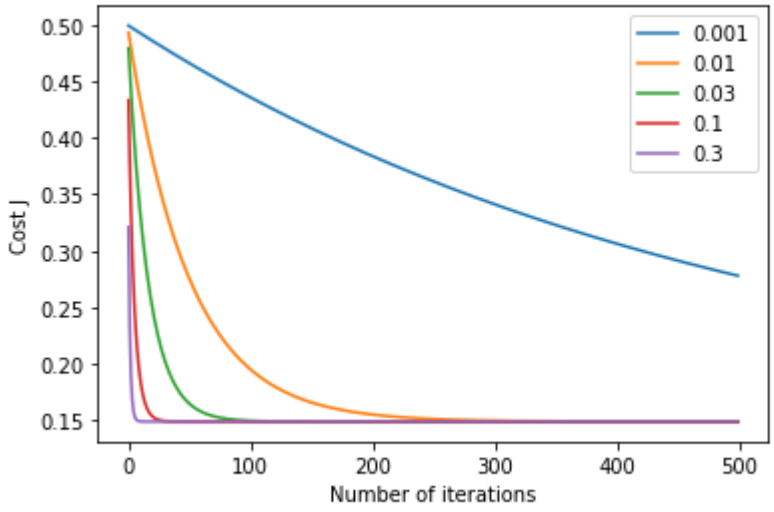
- In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly.
- You can change the learning rate by modifying the following code and changing the Use your implementation of gradient Descent function and run gradient descent for about 500 iterations at the chosen learning rate.
- The function should also return the history of J(θ) values in a vector J. After the last iteration, plot the J values against the number of the iterations.

- If you picked a learning rate within a good range, your plot look similar as the following Figure. part of the code that sets the learning rate

```
# CHANGE THE VALUES of ALPHAS, 5 VALUES OF ALPHA
#PLOT LEARNING RATES FOR FOLLOWING FOR ALPHAS, NO NEED TO CHANGE THE CODE ONLY REQUIRE


# some gradient descent settings
iterations = 500
alpha = [0.001, 0.01, 0.03, 0.1, 0.3] #--------------ENTER YOUR LEARNING RATES
costs=[]

for i in range(5):
  theta = np.zeros(2)
  theta, J_history = gradientDescent(X ,Y, theta, alpha[i], iterations)
          # initialize fitting parameters
  costs.append(J_history)
# Plot the convergence graph

for i in range(5):
  plt.plot(np.arange(len(costs[i])), costs[i], label=str(alpha[i]))
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
plt.legend()
```

Output:

```
<matplotlib.legend.Legend at 0x7f642de3f8d0>
```



- **It is observed from the graph that alpha=0.3 learning rate performs better than other rates, because the cost converges earlier comparatively.**
- **0.3 is optimum value of alpha (learning rate)**
- **Effect of alpha on gradient is that:**

If we keep on increasing its value then gradient descent becomes more effective as it converges with less number of iterations.
But alpha has some limitation, if we keep on increasing alpha even more, then it overfits the algorithm, thus gives unwanted result and it won't converge.

PART 02

OPTIMIZATION FUNCTIONS:

We will use the fmin and fmin_cg function of SciPy in ex1P2.ipyb. DATA initialization will be same as above:

```
res1 = optimize.fmin_cg(J, x0, fprime=gradf,args=args)
```

:tps://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin_cg.html#scipy.optimize.fmin_cg

```
res2 = optimize.fmin(J, x0, args=args)
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html#scipy.optimize.fmin

- **We will be using 2 functions of scipy library :**
- **Fmin_cg and Fmin**
- **Fmin uses downhill algorithm, while fmin_cg uses complex conjugate algorithm**

Cost function:

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

```
import matplotlib.pyplot as plt

# used for manipulating directory paths
import os

from pylab import *
from numpy import *
from numpy.random import normal
from scipy.optimize import fmin_cg

# Read comma separated data
data = np.loadtxt(os.path.join('Data', path ), delimiter=',')
X, Y = data[:, 0], data[:, 1]


args = (X,Y)  # parameter values
m=len(X)
al=1
lr2=[]
```

Cost function code:

```
#--------  COST FUNCTION-------------
def J(t,x,y):
  theta=t
  #============YOUR COST FUNCTION CODE HERE====
  x=np.stack([np.ones(m),x],axis=1)
  h=np.dot(x,theta)
  J=np.divide(np.sum(np.square(h-y)),2*m)
  lr2.append(J)
  return J
#-----------------------------------------------
```

Gradiant function:

Also u have to write definition of "gradf" function for **fmin_cg** which only requires:

$$\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j).$$

Gradient is a bit different algorithm than gradient descent.

Code:

```
# -----------GRADIENT ONLY FUNCTION-----------
def gradf(t, *args):
  theta =t
  #===============GRADIENT ONLY CODE HERE
  x=np.stack([np.ones(Y.size),X],axis=1)
  y=args[1].copy()
  h=np.dot(x,theta)
  loss=h-y
  theta=(1/m)*(x.T.dot(loss))
  #=====================================================================
  #lr2.append(J)
  return theta
```

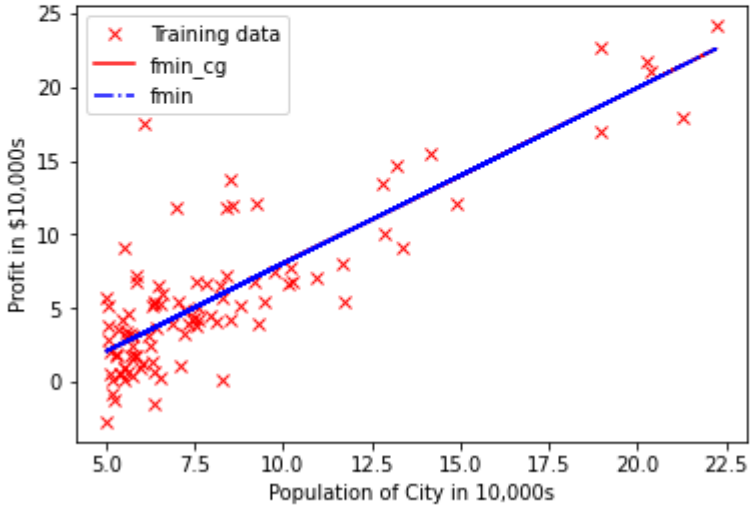
- **Gradient is used in fmin_cg and fmin functions:**

```
#=============================================================
x0 = [0,0]  # Initial guess.
from scipy import optimize

res1 = optimize.fmin_cg(J, x0, fprime=gradf,args=args)
lr1=lr2
lr2=[]
res2 = optimize.fmin(J, x0, args=args)

X = np.stack([np.ones(m), X], axis=1)

plot(X[:,1],Y,'rx', X[:,1],np.dot(X, res1),'r', X[:,1], np.dot(X, res2),'b-.')
plt.ylabel('Profit in $10,000s'); # Set the y ? axis label
plt.xlabel('Population of City in 10,000s'); # Set the x ? axis label
plt.legend([ 'Training data','fmin_cg','fmin']);
plt.show()
print(res1)
```

Output:

This is showing the output of fmin_cg and fmin, that how much iterations both of these functions took to converge:

```
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 6
         Function evaluations: 14
         Gradient evaluations: 14
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 90
         Function evaluations: 172
```

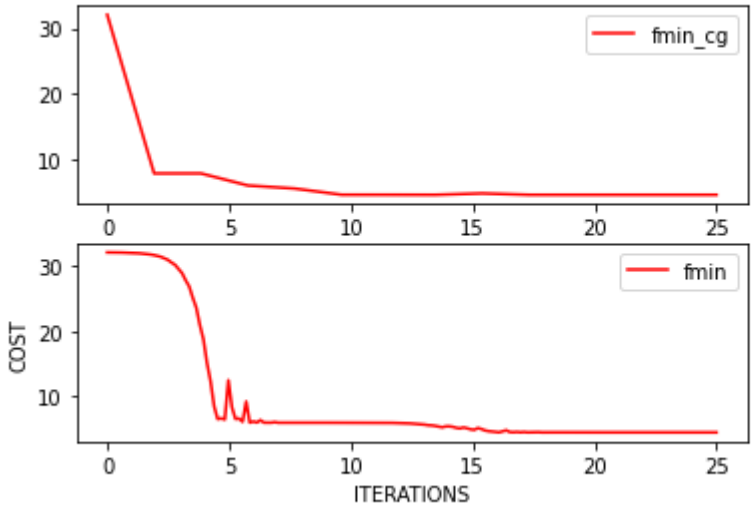



```
[-3.89582548  1.19303812]
```

LEARNING RATES:

```
[ ]  # plot learning rates of fmin and fmin_cg
     xx = linspace(0,25,len(lr1))
     xy = linspace(0,25,len(lr2))
     plt.subplot(2,1,1)

     plt.legend()
     plt.plot( xx, lr1,'r',label='fmin_cg') # (lr1) learning rate 1 of fmin_cg
     plt.legend()
     plt.subplot(2,1,2)
     plt.plot( xy,lr2,'r',label='fmin')  # (lr2)  learning rate 1 of fmin_cg
     plt.ylabel('COST'); # Set the y  axis label
     plt.xlabel('ITERATIONS'); # Set the x  axis label
     plt.legend()
     show()
```

```
No handles with labels found to put in legend.
```



- **It is observed from the graph that fmin_cg performed better than fmin function, because in this case the cost functions converges faster, relatively.**
- **Learning rate converges at around 1.5, while learning rate 2 converges at 5, so Lr 1 is better than Lr 2.**