Question 1

February 10, 2020

1 COL744 : Machine Learning (Assignment 1)

1.1 Question 1

1.1.1 Part (a) : Implementing Batch Gradient Descent

- In this part I have implemented batch SGD for the loss function $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - h_\theta(x^{(i)}))^2$ on the given dataset.
- I have taken η= **0.01**.
- As stopping criteria I am checking **difference of** $J(\theta)$ between two iterations and if it is less than **1e-15** then stopped the iterations. I have also taken an extra parameter **max_iter** = 10^6 describing maximum iterations allowed to bound the number of iterations.
- on given dataset we get (θ_0, θ_1) = **(0.996620, 0.001340)**

[1]:
```
import numpy as np
import numpy as np

import matplotlib as mp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation

from tqdm import tqdm

from sklearn.metrics import mean_squared_error
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
```

- Loading the data

[2]:
```
X_unnormalized=np.genfromtxt('./ass1_data/data/q1/linearX.csv') #loading data
Y_train=np.genfromtxt('./ass1_data/data/q1/linearY.csv')
```

- Normalizing dataset and adding x_0

[3]:
```
X_=(X_unnormalized-X_unnormalized.mean(axis=0))/X_unnormalized.std(axis=0)
X_train=np.vstack((np.ones(X_.shape), X_)).T
```

- Implementing batch gradient descent

[4]:
```
def grad(theta, X, Y):
    '''Function to compute partial differentiation wrt theta'''
    err = (X.dot(theta)) - Y #100x1
    loss_val = ((err**2).sum())/(2*X.shape[0])
    grad_val = (1/X.shape[0])*((X.T).dot(err))
    return (grad_val, loss_val)
def LinearRegressionGD(X,Y, r=0.1, max_iter=10**6):
    theta = np.zeros(X.shape[1])
    loss_lst = []
    theta_list=[]
    for i in (range(max_iter)):
        if np.isnan(theta).any():#Checking for divergence
            print('Diverged to infinity')
            return ([],[])
        (grad_val, loss_val) = grad(theta, X, Y)
        theta_next= theta - r * np.array(grad_val)
        theta_list.append(theta)
        loss_lst.append(loss_val)
        if(i>2 and abs(loss_lst[-1]-loss_lst[-2])<1e-15):
            print('converged in %d iterations'%(i))
            break
        theta=theta_next
    return (theta_list, loss_lst)
```

[5]:
```
(theta_lst, loss_lst) = LinearRegressionGD(X_train, Y_train, r=0.01,␣
↪max_iter=10**6)
```

```
converged in 1490 iterations
```

[6]:
```
theta = theta_lst[-1]
print('theta found by my implementation :(theta0, theta1) = (%f,␣
↪%f)'%(theta[0], theta[1]))
```

```
theta found by my implementation :(theta0, theta1) = (0.996620, 0.001340)
```

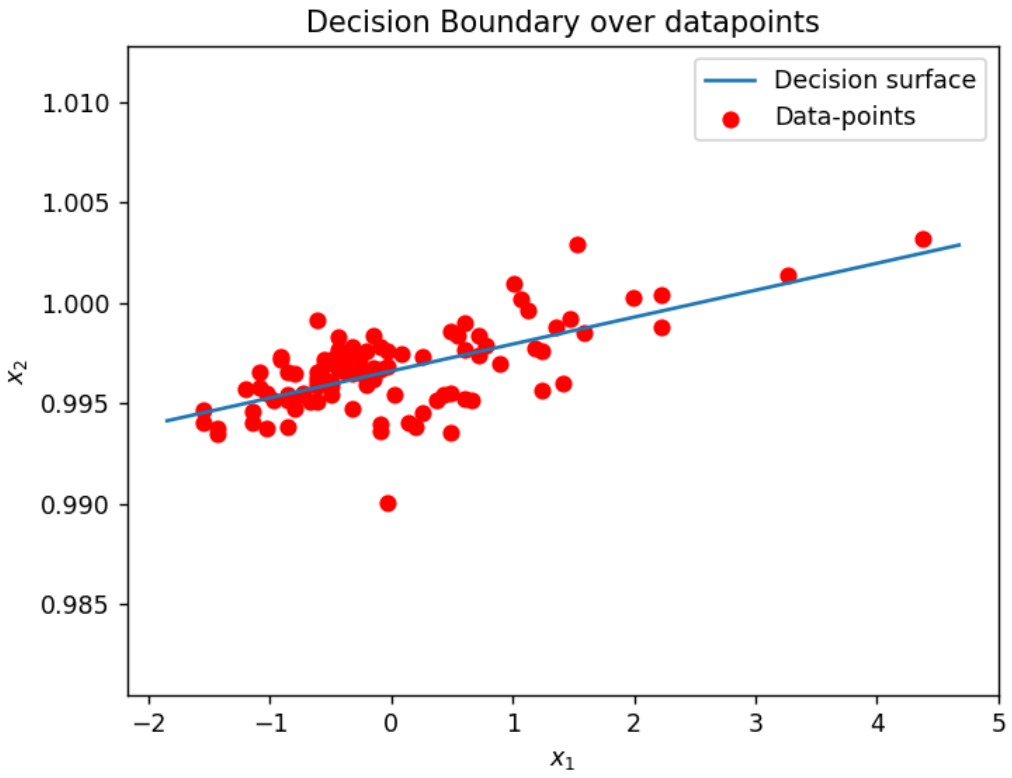
1.1.2 Part (b) Plotting the decision surface

[7]:
```
fig, ax = plt.subplots()
ax.scatter(X_train[:,1:], Y_train, c='r', label='Data-points')
axes = plt.gca()
x_vals = np.array(axes.get_xlim())
y_vals = theta[0] + theta[1] * x_vals
```

```
plt.plot(x_vals, y_vals, label = 'Decision surface')
plt.title('Decision Boundary over datapoints')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```



1.1.3 Part (c) : Plotting movement of θ in 3D

3D Plot

[8]:
```
def plot3d(theta_lst, loss_lst, X_train, Y_train):
    fig = plt.figure(figsize=(8,6))

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

```
    theta1 = np.arange(-1, 1, 10e-3) #X
    theta0 = np.arange(0, 2, 10e-3) #Y
    X, Y = np.meshgrid(theta1, theta0)
    loss = [[(0.5/X_train.shape[0])*np.linalg.norm((X_train[:,1]*t1 + X_train[:
↪,0]*t0 - Y_train), ord=2)**2 for t1 in theta1] for t0 in theta0]
    Z = np.array(loss).T

    # Plot the surface.
    surf = ax.plot_surface(X, Y, Z, cmap='Reds',
                           linewidth=0, antialiased=False,alpha=0.5)#,␣
↪label='loss surface')
    ax.set_xlabel('$\\theta_1$', color='r')
    ax.set_ylabel('$\\theta_0$', color='r')
    ax.set_zlabel('J($\\theta$)', color='r')

    graph, = plt.plot([], [], 'x',markersize=1, c='black', label = '$<\\theta_1,␣
↪\\theta_0, J(\\theta)>$')
    def animate(i):
        graph.set_data(data[:i+1,1], data[:i+1,0])
        graph.set_3d_properties(loss_data[:i+1])
        return graph

    data = np.array(theta_lst)
    loss_data = np.array(loss_lst)
    anim = FuncAnimation(fig, animate, interval=200)
    plt.legend(loc=4)
    plt.title('3Dplot representing movement of theta ')
    plt.show()
    return anim
```

[11]:
```
anim = plot3d(theta_lst, loss_lst, X_train, Y_train)
```

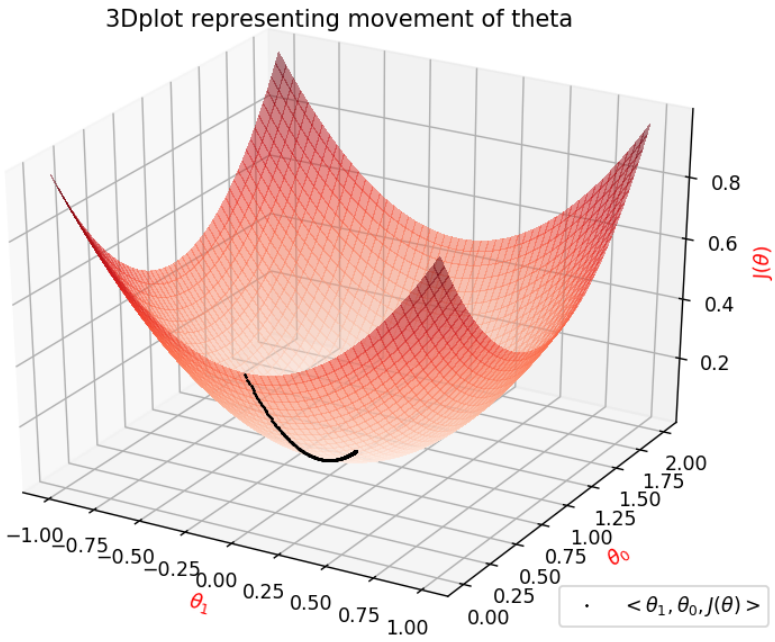
```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.1.4 Part (d) : Plotting contour plot representing movement of θ

Contour Plot

[12]:
```
def plot_contour(theta_lst, X_train, Y_train, learning_rate):
    plt.rcParams.update({'font.size': 7})
    theta1 = np.arange(-1, 1, 10e-2) #X
    theta0 = np.arange(-0.5, 1.5, 10e-2) #Y
    X, Y = np.meshgrid(theta1, theta0)
```




```
   loss = [[np.linalg.norm((X_train[:,1]*t1 + np.ones(X_train.shape[0])*t0 -␣
↪Y_train), ord=2)**2 for t1 in theta1] for t0 in theta0]
   Z = np.array(loss)

   fig, ax = plt.subplots()
   CS = ax.contour(X, Y, Z)
   ax.clabel(CS, inline=1, fontsize=5)
   ax.set_title('countour plot representing movement of theta in each␣
↪iterations for $\\eta$ = %s'%(learning_rate))
   ax.set_xlabel('$\\theta_1$', color='r')
   ax.set_ylabel('$\\theta_0$', color='r')

   graph, = plt.plot([], [], 'x',markersize=1, label = '$<\\theta_1,␣
↪\\theta_0>$')

   def animate(i):
       graph.set_data(data[:i+1, 1:2], data[:i+1, 0:1])
       return graph

   data = np.array(theta_lst)
   anim = FuncAnimation(fig, animate, interval=100)
   plt.legend()
```

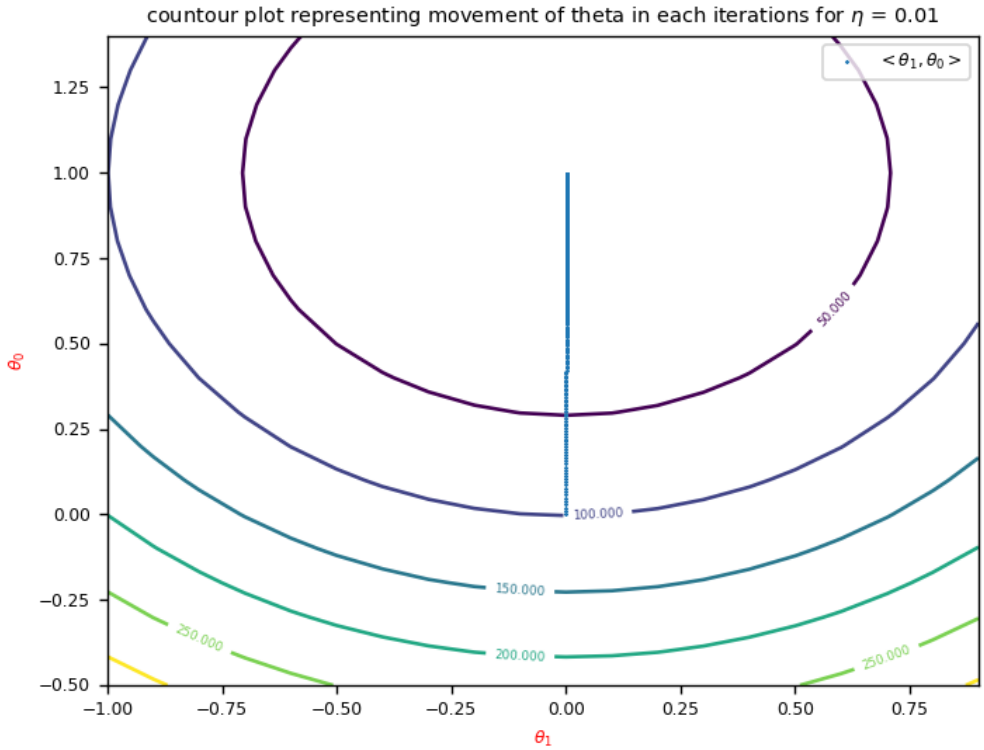
```
    plt.show()
    return anim
```

[11]:
```
anim = plot_contour(theta_lst, X_train, Y_train, 0.01)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[11]:
```
<matplotlib.animation.FuncAnimation at 0x1dde8018988>
```



1.1.5 Part (e) : Comparing various learning rate and plotting Contour Plot for them

- For η=**0.001**

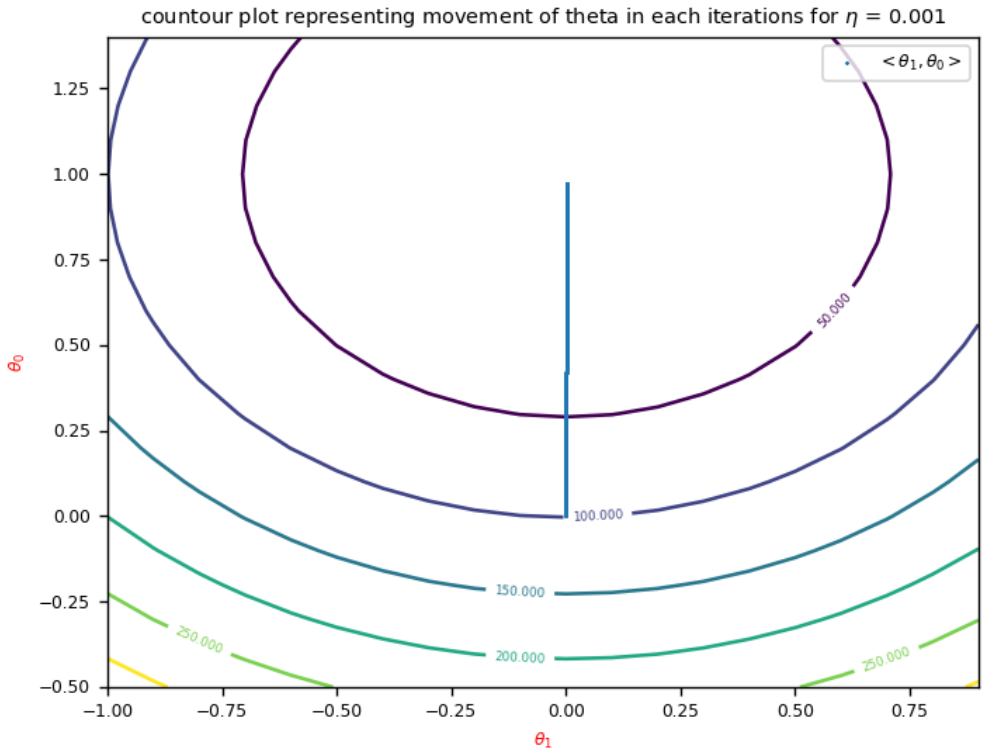
[12]:
```
(theta_lst, loss_lst) = LinearRegressionGD(X_train, Y_train, r=0.001,␣
↪max_iter=10**6)
plot_contour(theta_lst, X_train, Y_train, 0.001)
```

```
converged in 13806 iterations
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[12]: <matplotlib.animation.FuncAnimation at 0x1dde80dde88>



- For η=**0.025**

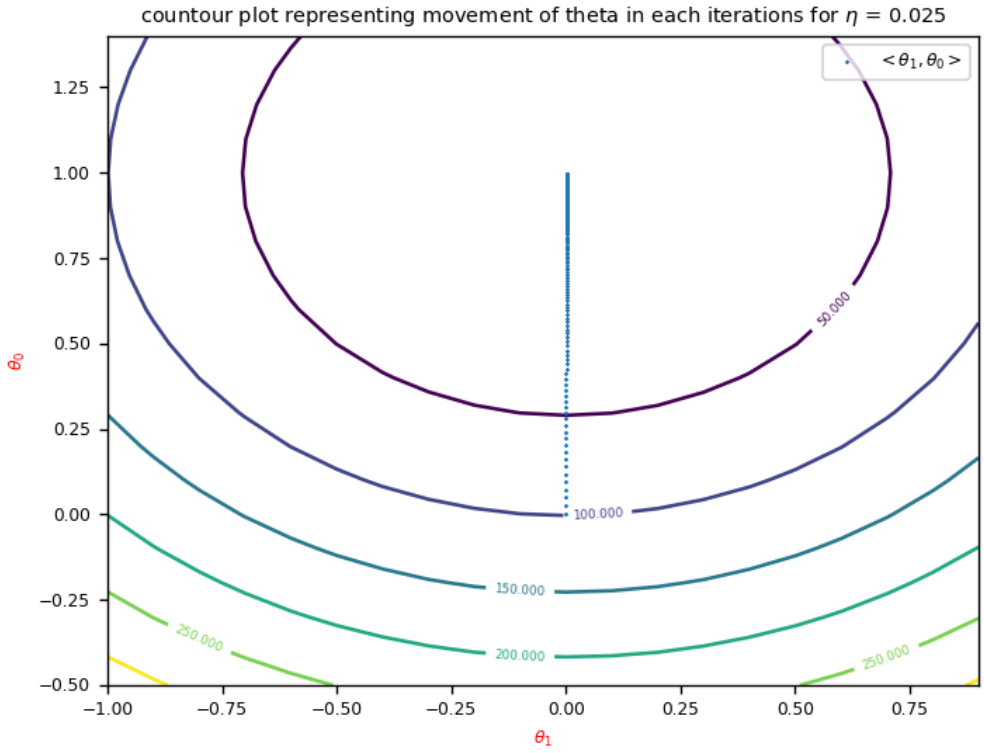
[13]:
```
(theta_lst, loss_lst) = LinearRegressionGD(X_train, Y_train, r=0.025,␣
↪max_iter=10**6)
plot_contour(theta_lst, X_train, Y_train, 0.025)
```

```
converged in 610 iterations
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[13]: <matplotlib.animation.FuncAnimation at 0x1dde8a298c8>

- For η=**0.1**



[14]:
```
(theta_lst, loss_lst) = LinearRegressionGD(X_train, Y_train, r=0.1,␣
↪max_iter=10**6)
plot_contour(theta_lst, X_train, Y_train, 0.1)
```

```
converged in 154 iterations
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[14]:
```
<matplotlib.animation.FuncAnimation at 0x1dde7d4b388>
```

1.1.6 Observations

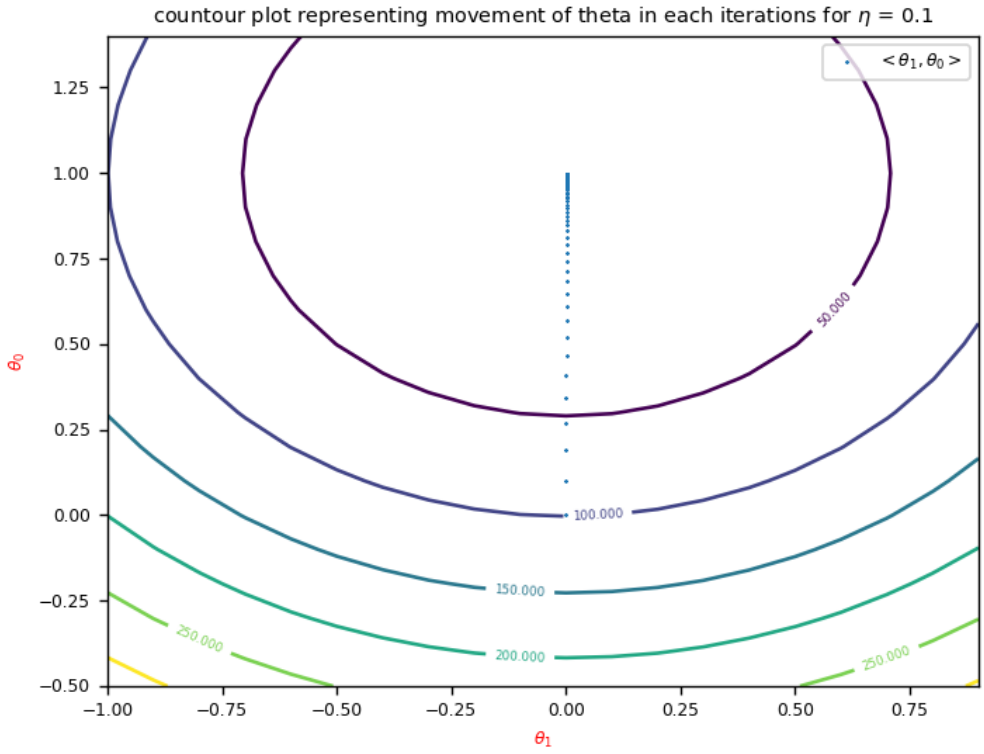
- Here I have mentioned number of iterations per eta in the table below

η	Number of Iterations
0.01	1490
0.001	13806
0.025	610
0.1	154

η	Number of Iterations

Here in this table we can see that number of iterations decreases as be increase eta as the difference would be higher.

- Also in contour plot we can notice that for large η like 0.1 we see big jumps in starting and as θ converges to θ^* we can see smaller jumps, whereas in smaller θ like 0.001 we can see that from the beginning it is taking very small jumps and relatively it also takes more time for θ to reach to optimal value as number of iterations was very large in comparision.


countour plot representing movement of theta in each iterations for η = 0.1
$<\theta_1, \theta_0>$
50.000
100.000
150.000
200.000
250.000
250.000
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
θ_0
θ_1

COL744 : Machine Learning (Assignment 1)

Question 2

Part (a) : Smapling 1 million datapoints

In this part I have sampled 10^6 datapoints with 2 features X_1 and X_2 and then found $y^{(i)}$ by adding error ϵ_i to $\theta^T x^{(i)}$ with θ = [3,1,2]. Here all the variables come from the following distribution.

- $y = \Sigma_{i=1}^{m} \theta_i x_i + \epsilon_i$
- $x1 \sim \mathcal{N}(3, 4)$
- $x2 \sim \mathcal{N}(-1, 4)$
- $\epsilon_i \sim \mathcal{N}(0, 2)$

In [1]:
```
%matplotlib notebook
from tqdm import tqdm
import numpy as np
import pandas as pd
import pickle
import os
import math

import matplotlib as mp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
```

In [2]:
```
theta = np.array([3,1,2])
sampleSize=10**6
```

In [3]:
```
if os.path.isfile('./data/xQ2.pkl') and os.path.isfile('./data/yQ2.pkl'):
    with open('./data/xQ2.pkl', 'rb') as f:
        X = pickle.load(f)
    with open('./data/yQ2.pkl', 'rb') as f:
        Y = pickle.load(f)
elif not os.path.isfile('xQ2.pkl'):
    dataX1 = [np.random.normal(loc=3, scale=2) for i in range(sampleSize)]
    dataX2 = [np.random.normal(loc=-1, scale=2) for i in range(sampleSize)]
    error = [np.random.normal(loc=0, scale=math.sqrt(2)) for i in range(sampleSize)]

    X_ = np.vstack((dataX1,dataX2)) # X_.shape = (2,smpaleSize)
    X = np.vstack((np.ones(X_.shape[1]), X_)).T # X.shape = (sampleSize,3)
    Y = (X.dot(theta)) - error

    if not os.path.isdir('./data'):
        os.mkdir('./data')

    with open('./data/xQ2.pkl', 'wb') as f:
        pickle.dump(X,f)
    with open('./data/yQ2.pkl', 'wb') as f:
        pickle.dump(Y,f)
```

(b) Implementing SGD

- In this part I have implemented SGD with η = 0.001 and tested it with differnt batch size of (1,100,10000,1000000).
- To check the convergence I have taken average of $J(\theta)$ after every 1000 iterations and if their difference is less than 1e-4 then I have stopped the algorithm. Also I have passed a parameter max_iter = 10^6 to stop the algorithm after that many iterations.

In [4]:
```
def grad(theta, X, Y):
    '''Function to compute partial differentiation wrt theta'''
    err = (X.dot(theta)) - Y #100x1
    loss_val = ((err**2).sum())/(2*X.shape[0])
    grad_val = (1/X.shape[0])*((X.T).dot(err))
    return (grad_val, loss_val)

def SGD(X,Y,lr=0.001,r=1, max_iter=10**6, tolerance=1e-4):
    indices = np.random.permutation(X.shape[0])
    X = X.take(indices, axis=0)
    Y = Y.take(indices, axis=0)
    batchNo = 0
    epoch = 0
    currentSum = 0
    previousAvg = np.inf

    theta = np.zeros(X.shape[1])
    totalBatchForOneEpoch = X.shape[0]/r

    loss_lst = []
    theta_lst=[]

    for i in tqdm(range(max_iter)):
        if batchNo == totalBatchForOneEpoch:
            batchNo=0
            epoch+=1
        if i%1000 == 0:
            if abs((currentSum/1000)-previousAvg) <= tolerance:
                print('converged in %d iterations'%(i))
                break
            else:
                previousAvg = currentSum/1000
                currentSum = 0
        X_curr = X[(batchNo*r):((batchNo+1)*r),:]
        Y_curr = Y[(batchNo*r):((batchNo+1)*r)]
        (grad_val, loss_val) = grad(theta, X_curr, Y_curr)
        currentSum += loss_val
        theta_next= theta - lr * np.array(grad_val)
        theta_lst.append(theta)
        loss_lst.append(loss_val)
        theta=theta_next
        batchNo+=1

    return (theta_lst, loss_lst)
```

In [5]:
```
(theta_lst1, loss_lst1) = SGD(X,Y,r=1)
```

```
 54%|                                                                                | 53833
5/1000000 [00:08<00:07, 61841.51it/s]

converged in 543000 iterations

 54%|                                                                                | 53833
5/1000000 [00:08<00:07, 62138.13it/s]
```

In [6]:
```
(theta_lst100, loss_lst100) = SGD(X,Y,r=100)
```

```
100%|                                                                               | 1000000/
1000000 [00:17<00:00, 57024.11it/s]
```

In [7]:
```
(theta_lst10000, loss_lst10000) = SGD(X,Y,r=10000)
```

```
  2%|                                                                               | 1766
9/1000000 [00:02<02:33, 6405.87it/s]

converged in 18000 iterations

  2%|                                                                               | 1766
9/1000000 [00:02<02:36, 6268.92it/s]
```

In [8]:
```
(theta_lst1000000, loss_lst1000000) = SGD(X,Y,r=10**6)
```

```
  2%|                                                                               | 1799
9/1000000 [05:56<5:07:24, 53.24it/s]

converged in 18000 iterations

  2%|                                                                               | 1799
9/1000000 [06:10<5:07:24, 53.24it/s]
```

In [9]:
```
print('for r=1 --> Theta = {}'.format(theta_lst1[-1]))
print('for r=100 --> Theta = {}'.format(theta_lst100[-1]))
print('for r=10000 --> Theta = {}'.format(theta_lst10000[-1]))
print('for r=1000000 --> Theta = {}'.format(theta_lst1000000[-1]))
```

```
for r=1 --> Theta = [2.97919524 1.01608836 2.04332131]
for r=100 --> Theta = [3.00054451 1.00314857 1.99742115]
for r=10000 --> Theta = [2.97844104 1.00384113 1.99781854]
for r=1000000 --> Theta = [2.97843004 1.00397486 1.99797575]
```

Part (c) : Observing various thetas and finding their error on test dataset

- The thetas that we got for different batch size are given in the table below with some other useful information.

Batch Size (r)	θ	Number of iterations	Iterations/sec*	Time(Sec)*	Error on Test Dataset
1	[2.97 1.01 2.04]	5,43,000	61,841	8	1.4727
100	[3.00 1.00 1.99]	10,00,000	57,024	17	1.4028
10000	[2.97 1.00 1.99]	18,000	6,268	2	1.4030
1000000	[2.97 1.00 1.99]	18,000	53	370	1.4030

* - (Reported by tqdm)||||**Error for θ=[3,1,2]**|1.4021

- As we can see that all these theta values are close to [3,1,2] which we used to sampled to dataset.
- Number of iterations for r=1 is less than r=100. And for r=10000 and 1000000 number of iterations are fairly small as they are finding gradient over more number of examples in a given iteration.
- Iterations per second are significanlty reduces as r increases from 1 to 1000000.
- Overall time also increases as r goes from 1 to 1000000.
- Error on test dataset is also very close to the error if we get theta=[3,1,2] as theta we got in every case is much closer to [3,1,2].

In [10]:
```
theta1 = theta_lst1[-1]
theta100 = theta_lst100[-1]
theta10000 = theta_lst10000[-1]
theta1000000 = theta_lst1000000[-1]
```

In [13]:
```
Data = pd.read_csv('./ass1_data/data/q2/q2test.csv').to_numpy()

X = Data[:,:2]
Y = Data[:,2]

def findError(X,Y,theta):
    pred = X.dot(theta[1:3]) + theta[0]
    error = Y - pred
    rmse = np.sqrt((error**2).sum()/(X.shape[0]))
    return rmse
print('Error on test data with (r=1) : %s'%(findError(X[:,:2],Y,theta1)))
print('Error on test data with (r=100) : %s'%(findError(X[:,:2],Y,theta100)))
print('Error on test data with (r=10000) : %s'%(findError(X[:,:2],Y,theta10000)))
print('Error on test data with (r=1000000) : %s'%(findError(X[:,:2],Y,theta1000000)))
print('Error on test data using theta = [3,1,2] : %s'%(findError(X[:,:2],Y,np.array([
3,1,2]))))
```

```
Error on test data with (r=1) : 1.4727354805367239
Error on test data with (r=100) : 1.4028271975941902
Error on test data with (r=10000) : 1.403018409600393
Error on test data with (r=1000000) : 1.4030107997954164
Error on test data using theta = [3,1,2] : 1.4021033638787122
```

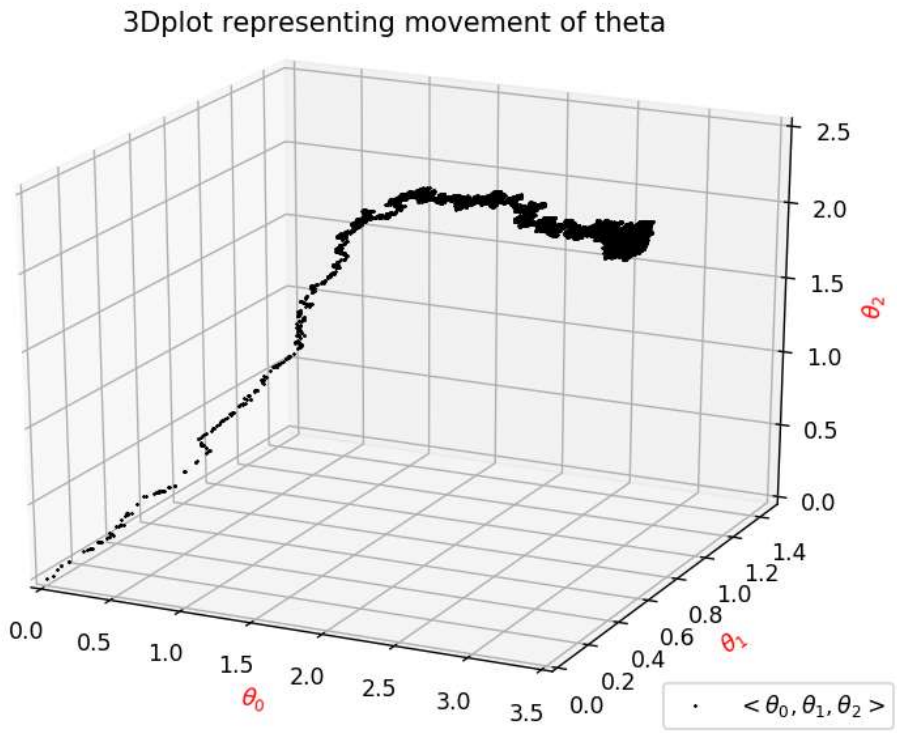
Part (d) : Plotting movement of θ in 3d space

In this part I have plotted movement of theta in 3d space for all 4 sizes.

In [14]:
```
def plotTheta3D(theta_lst):
    fig = plt.figure(figsize=(8,6))
    ax = fig.gca(projection='3d')

    ax.set_xlabel('$\\theta_0$', color='r')
    ax.set_ylabel('$\\theta_1$', color='r')
    ax.set_zlabel('$\\theta_2$', color='r')
    ax.set_zlim(0,2.5)
    ax.set_xlim(0, 3.5)
    ax.set_ylim(0, 1.5)
    graph, = plt.plot([], [], 'x',markersize=1, c='black', label = '$<\\theta_0, \\th
eta_1, \\theta_2>$')
    def animate(i):
        graph.set_data(data[:i+1,0], data[:i+1,1])
        graph.set_3d_properties(data[:i+1,2])
        return graph

    data = np.array(theta_lst)
    anim = FuncAnimation(fig, animate, interval=1)
    plt.legend(loc=4)
    plt.title('3Dplot representing movement of theta ')
    plt.show()
    return anim
```

- Plot Representing movement of θ in 3d space for batch size=1

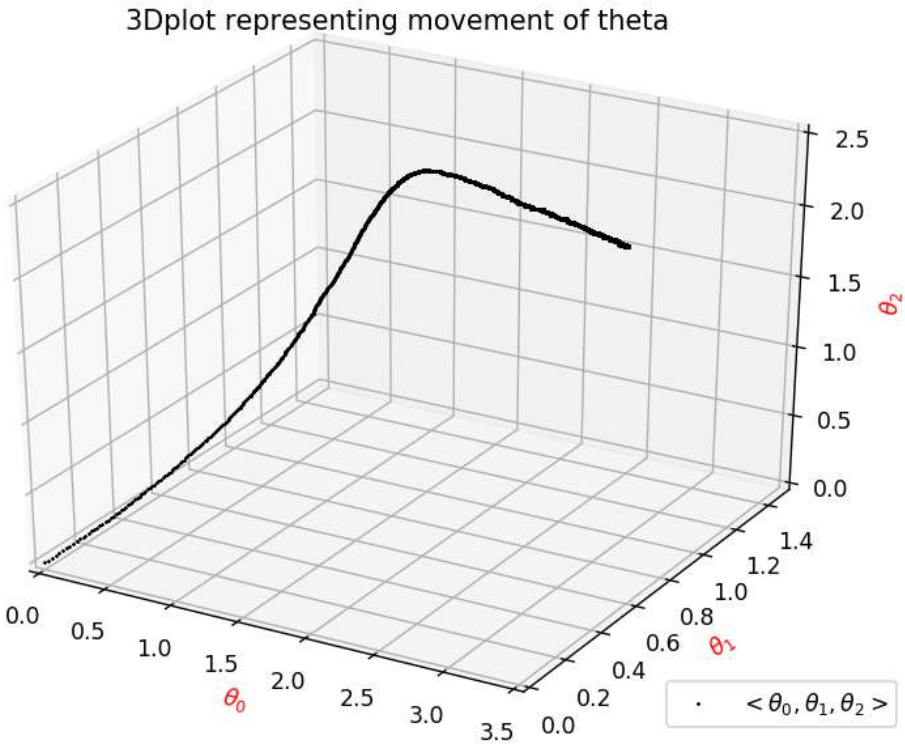
In [23]:
```
plotTheta3D(theta_lst1)
```



Out[23]: <matplotlib.animation.FuncAnimation at 0x25a8db67248>

- Plot Representing movement of θ in 3d space for batch size=100

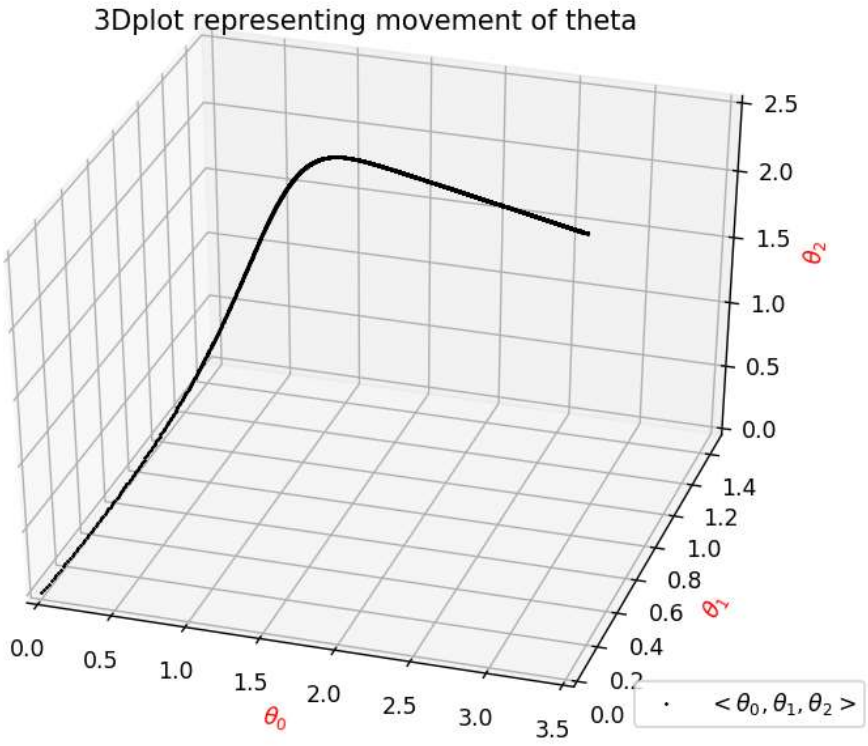
```
In [31]: plotTheta3D(theta_lst100)
```




```
Out[31]: <matplotlib.animation.FuncAnimation at 0x1cbcc778108>
```

- Plot Representing movement of θ in 3d space for batch size=10000

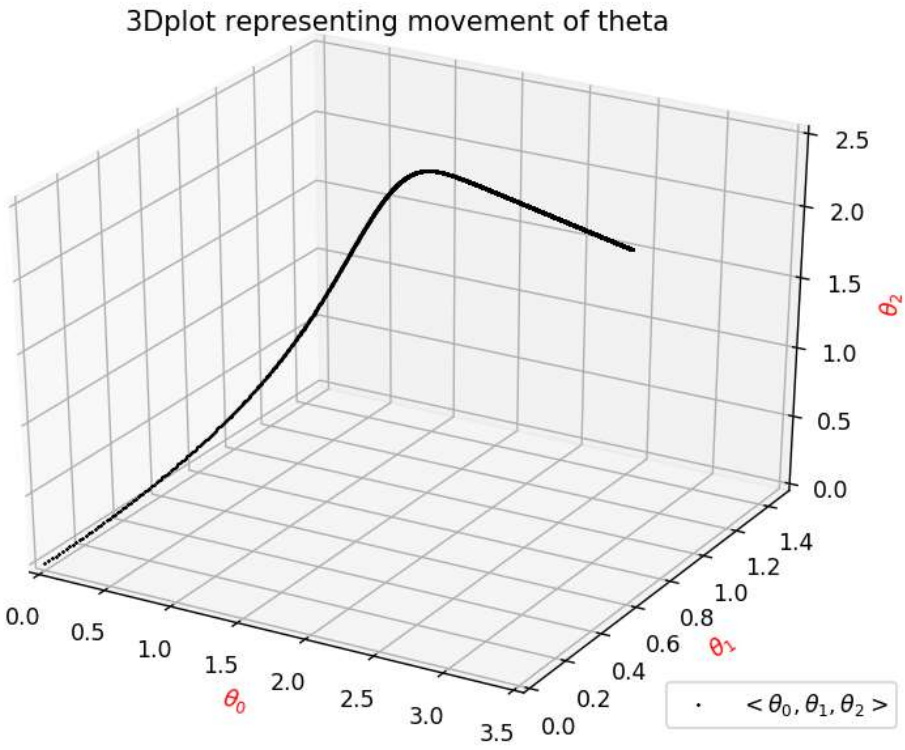
In [21]:
```
plotTheta3D(theta_lst10000)
```



Out[21]:
```
<matplotlib.animation.FuncAnimation at 0x25a8db37c88>
```

- Plot Representing movement of θ in 3d space for batch size=1000000

In [24]:
```
plotTheta3D(theta_lst1000000)
```



Out[24]:
```
<matplotlib.animation.FuncAnimation at 0x25a8dfda7c8>
```

Observations:

In this plots we can see that theta for batch size = 1 moves with lots of jitter in space, and as batch size increases theta moves smoothly in the space. Which is as expected as lower batch size will update theta after seeing fewer datapoints.

Question 3

February 10, 2020

1 COL744 : Machine Learning (Assignment 1)

1.1 Question 3

1.1.1 Part (a) : Implementing logistic regression using newton's method

In this part I have implemented logistic regression using newton't update method to find optimal theta.

[1]:
```
import numpy as np
import numpy as np

import matplotlib as mp
import matplotlib.pyplot as plt

from tqdm import tqdm
import math
```

[2]:
```
X_unnormalized=np.genfromtxt('./ass1_data/data/q3/logisticX.csv',␣
↪delimiter=',') #Loading X
Y=np.genfromtxt('./ass1_data/data/q3/logisticY.csv') #Loading Y

X_=(X_unnormalized-X_unnormalized.mean(axis=0))/X_unnormalized.std(axis=0)␣
↪#Normalizing X
X=np.hstack((np.ones((X_.shape[0],1)),X_)) # Adding X0
```

[3]:
```
def g(x):
    '''Logistic function'''
    return 1/(1+((math.e)**(-1*x)))

def newton(X,Y):
    '''
    Newton's update method to find theta

    ---Parameters
    * X,Y - Training examples
```

```
    ---Returns
    theta

    '''
    theta_lst=[]
    theta = np.zeros(X.shape[1])
    for i in range(100):
        g_thetaX = g(X.dot(theta))
        grad = (X.T).dot(Y)-(X.T).dot(g_thetaX)
        diag = np.diagflat((g_thetaX.T)*(np.ones(g_thetaX.shape)-g_thetaX))
        H = -(((X.T).dot(diag))).dot(X)
        theta_next = theta - (np.linalg.inv(H)).dot(grad)
        theta_lst.append(theta)
        theta=theta_next
        if(i>2 and np.sum(np.abs(theta_lst[-1]-theta_lst[-2]))<1e-10):
            print('converged in %d iterations'%(i))
            break
    return theta_lst[-1]
```

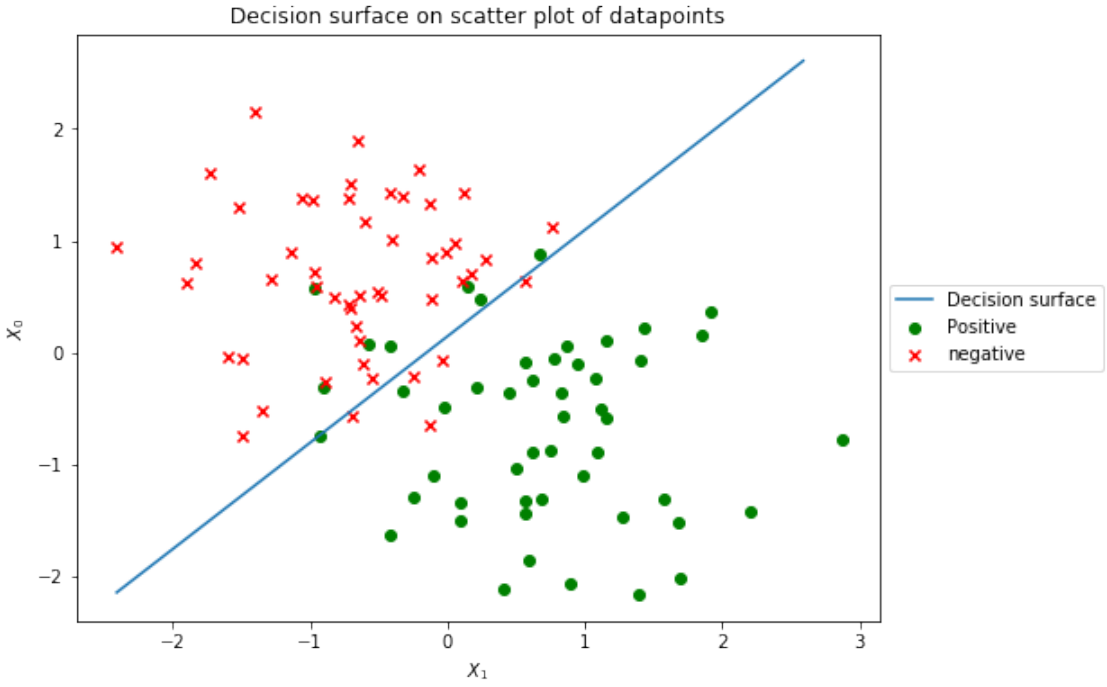
[4]:
```
theta=newton(X,Y)

print(theta)
```

```
converged in 9 iterations
[ 0.40125316  2.5885477  -2.72558849]
```

1.1.2 Part (b) : Ploting decision surface

[5]:
```
plt.figure(figsize=(8,6))

##### Plotting datapoints
pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
plt.scatter(pos_X[:,1], pos_X[:,2], marker='o', c='green', label='Positive')
plt.scatter(neg_X[:,1], neg_X[:,2], marker='x', c='red', label='negative')

##### Plotting decision surface
X_line = np.arange(X[:,1].min(), X[:,1].max())
Y_line = -1*(1/theta[2])*((theta[1]*X_line)+theta[0])
plt.plot(X_line, Y_line, label='Decision surface')


plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlabel('$X_1$')
plt.ylabel('$X_0$')
plt.title('Decision surface on scatter plot of datapoints')
```

```
plt.show()
```



1.1.3 Observations:

- Here we can see that decision boundary found by our algorithm nicely separates positive and negative points.

Question 4

February 10, 2020

1 COL744 : Machine Learning (Assignment 1)

1.1 Question 4

1.1.1 Part (a)

- In this part I have implemented GDA. So I have found values of ϕ μ_0, μ_1 and $\Sigma = \Sigma_0 = \Sigma_1$.

Formula for finding each of these are as below:

$$\phi = \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\}$$

$$\mu_0 = \frac{\Sigma_{i=1}^{m} 1\{y^{(i)} = 0\}x^{(i)}}{\Sigma_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

$$\mu_1 = \frac{\Sigma_{i=1}^{m} 1\{y^{(i)} = 1\}x^{(i)}}{\Sigma_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T$$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

[2]:
```
X_unnormalized=np.genfromtxt('./ass1_data/data/q4/q4x.dat')
categories, Y = np.unique(np.genfromtxt('./ass1_data/data/q4/q4y.dat',␣
↪dtype=str), return_inverse=True)

X = (X_unnormalized - X_unnormalized.mean(axis=0))/X_unnormalized.std(axis=0)
```

[3]:
```
def findParameters(X,Y):
    pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
    neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
    (m,n) = X.shape
    phi = pos_X.shape[0]/m
```

```
    mu_0 = np.sum(neg_X, axis=0)/neg_X.shape[0]
    mu_1 = np.sum(pos_X, axis=0)/pos_X.shape[0]

    Sigma = (np.dot((neg_X - mu_0).T, (neg_X - mu_0)) + np.dot((pos_X - mu_1).T,␣
↪(pos_X - mu_1)))/m
    return (phi, mu_0, mu_1, Sigma)
```

[4]:
```
(phi, mu_0, mu_1, Sigma) = findParameters(X,Y)
```

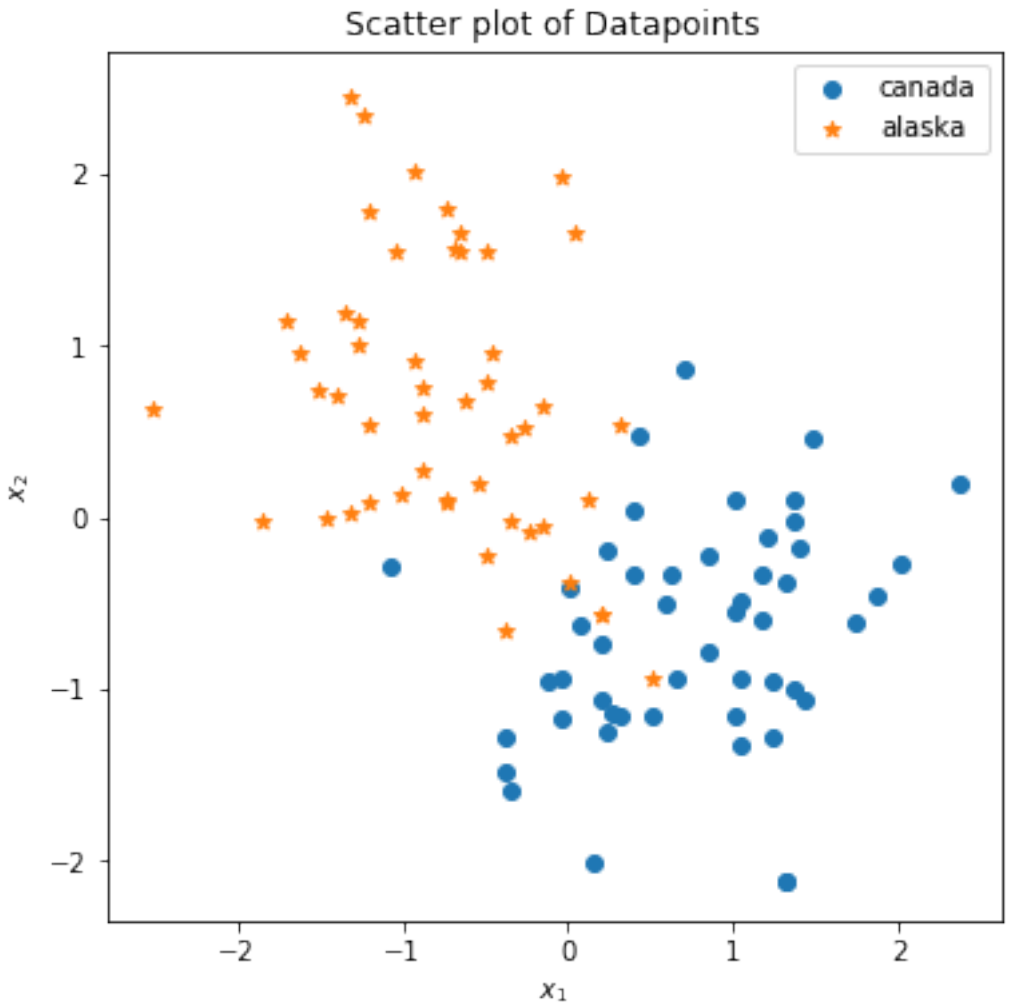
- Reporting values of μ_0, μ_1 and Σ

[5]:
```
print('Value of mu_0 : %s\n'%(mu_0))
print('Value of mu_1 : %s\n'%(mu_1))
print('Value of Sigma : \n%s'%(Sigma))
```

```
Value of mu_0 : [-0.75529433  0.68509431]

Value of mu_1 : [ 0.75529433 -0.68509431]

Value of Sigma :
[[ 0.42953048 -0.02247228]
 [-0.02247228  0.53064579]]
```

1.1.2 Part (b) : Plotting Datapoints

[6]:
```
fig = plt.figure(figsize=(6,6))

pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
plt.scatter(pos_X[:,0], pos_X[:,1], marker='o', label='canada')
plt.scatter(neg_X[:,0], neg_X[:,1], marker='*', label='alaska')
plt.legend()
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Scatter plot of Datapoints')
plt.show()
```



1.1.3 Part (c) : Plotting linear decision boundary

Equation for decision boundary when $\Sigma = \Sigma_1 = \Sigma_2$ is linear in terms of X. The equation is as follows.

$$\log(\frac{1-\phi}{\phi}) + \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0) + (\mu_0^T\Sigma^{-1} - \mu_1^T\Sigma^{-1})x = 0$$

Therefore,

$$\theta_0 = \log(\frac{1-\phi}{\phi}) + \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0)$$

and

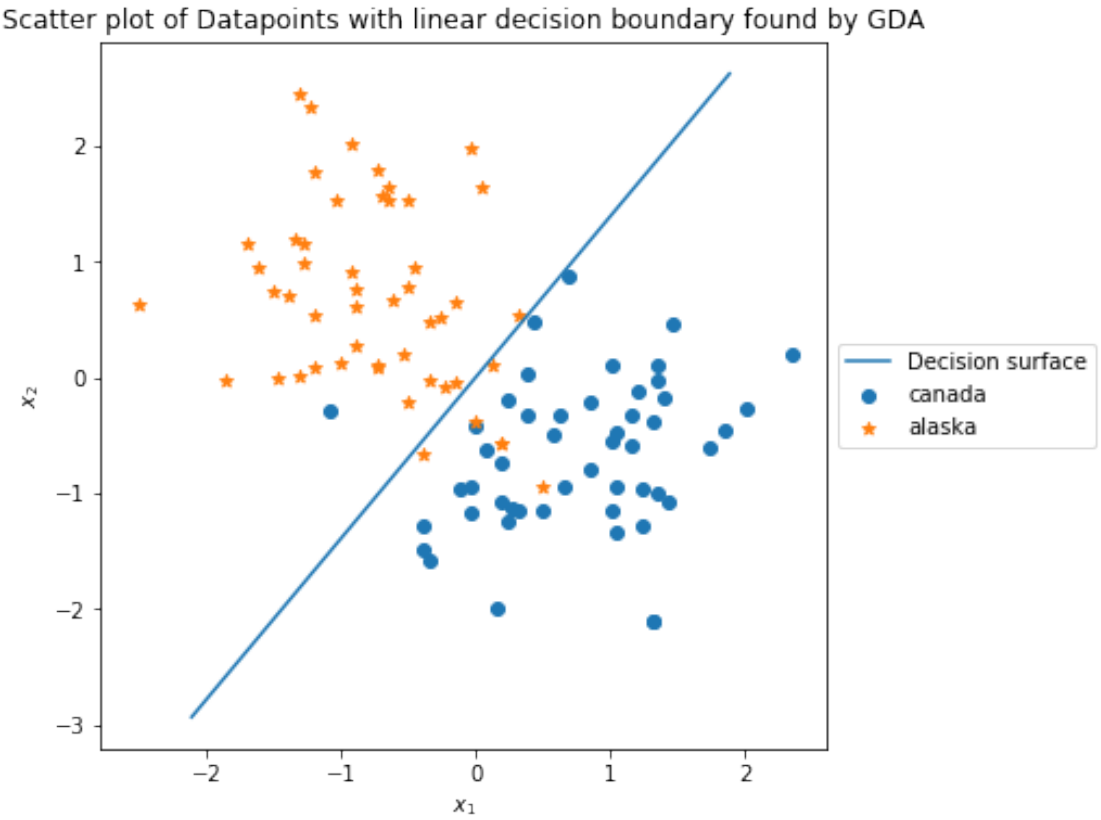
$$\theta = \mu_0^T\Sigma^{-1} - \mu_1^T\Sigma^{-1}$$

So Now computing θ and plotting dicision surface over the datapoints.

[7]:
```
def findTheta(phi, mu_0, mu_1, Sigma):
    Sigma_inv = np.linalg.inv(Sigma)
    theta_0 = np.log((1-phi)/phi) + (1/2)*(mu_1.T.dot(Sigma_inv).dot(mu_1) -␣
 ↪mu_0.T.dot(Sigma_inv).dot(mu_0))
    theta = (mu_0.T - mu_1.T).dot(Sigma_inv)
    return theta_0, theta
```

[8]:
```
(theta_0, theta) = findTheta(phi, mu_0, mu_1, Sigma)
```

[9]:
```
fig = plt.figure(figsize=(6,6))

pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
plt.scatter(pos_X[:,0], pos_X[:,1], marker='o', label='canada')
plt.scatter(neg_X[:,0], neg_X[:,1], marker='*', label='alaska')

X_line = np.arange(X[:,1].min(), X[:,1].max())
Y_line = -1*(1/theta[1])*((theta[0]*X_line)+theta_0)
plt.plot(X_line, Y_line, label='Decision surface')

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Scatter plot of Datapoints with linear decision boundary found by␣
 ↪GDA')
plt.show()
```



1.1.4 Part (d) : finding $\mu_0, \mu_1, \Sigma_0, \Sigma_1$

- Till now I have assumed that $\Sigma = \Sigma_0 = \Sigma_1$, but now we will find Σ_0 and Σ_1 separately using these equations.

$$\Sigma_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}},$$

$$\Sigma_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

[10]:
```
def findParameters2(X,Y):
    pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
    neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
    (m,n) = X.shape
    phi = pos_X.shape[0]/m
```

```
    mu_0 = np.sum(neg_X, axis=0)/neg_X.shape[0]
    mu_1 = np.sum(pos_X, axis=0)/pos_X.shape[0]

    x_mu0 = neg_X - mu_0
    x_mu1 = pos_X - mu_1
    Sigma0 = x_mu0.T.dot(x_mu0)/neg_X.shape[0]
    Sigma1 = x_mu1.T.dot(x_mu1)/neg_X.shape[0]

    return (phi, mu_0, mu_1, Sigma0, Sigma1)
```

[11]:
```
(phi, mu_0, mu_1, Sigma0, Sigma1) = findParameters2(X,Y)
```

- Reporting values of μ_0, μ_1, Σ_0 and Σ_1

[12]:
```
print('Value of mu_0 : %s\n'%(mu_0))
print('Value of mu_1 : %s\n'%(mu_1))
print('Value of Sigma0 : \n%s\n'%(Sigma0))
print('Value of Sigma1 : \n%s'%(Sigma1))
```

```
Value of mu_0 : [-0.75529433  0.68509431]

Value of mu_1 : [ 0.75529433 -0.68509431]

Value of Sigma0 :
[[ 0.38158978 -0.15486516]
 [-0.15486516  0.64773717]]

Value of Sigma1 :
[[0.47747117 0.1099206 ]
 [0.1099206  0.41355441]]
```

1.1.5 Part (e) : Plotting non-linear decision boundary

- Here as we have computed Σ_0 and Σ_1 separately, we get a non-linear decision boundary. Here is the equation for the same.

$$\log(\frac{(1-\phi)|\Sigma_1|^{1/2}}{\phi|\Sigma_0|^{1/2}}) + \frac{1}{2}(\mu_1^T\Sigma_1^{-1}\mu_1 - \mu_0^T\Sigma_0^{-1}\mu_0) + \frac{1}{2}x^T(\Sigma_1^{-1} - \Sigma_0^{-1})x + (\mu_0^T\Sigma_0^{-1} - \mu_1^T\Sigma_1^{-1})x = 0$$

So this decision boundary will turn out to be of form

$$ax_1^2 + bx_2^2 + cx_1x_2 + dx_1 + ex_2 + f = 0$$

where,

$a = (\frac{1}{2}(\Sigma_1^{-1} - \Sigma_0^{-1}))[1][1]$

$b = (\frac{1}{2}(\Sigma_1^{-1} - \Sigma_0^{-1}))[2][2]$

$c = (\frac{1}{2}(\Sigma_1^{-1} - \Sigma_0^{-1}))[1][2] + (\frac{1}{2}(\Sigma_1^{-1} - \Sigma_0^{-1}))[2][1]$

$d = (\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1})[1]$

$e = (\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1})[2]$

$f = \log(\frac{(1-\phi)|\Sigma_1|^{1/2}}{\phi|\Sigma_0|^{1/2}}) + \frac{1}{2}(\mu_1^T\Sigma_1^{-1}\mu_1 - \mu_0^T\Sigma_0^{-1}\mu_0)$ * Here the decision surface we got is a hyperbola as sign of a and b is different.

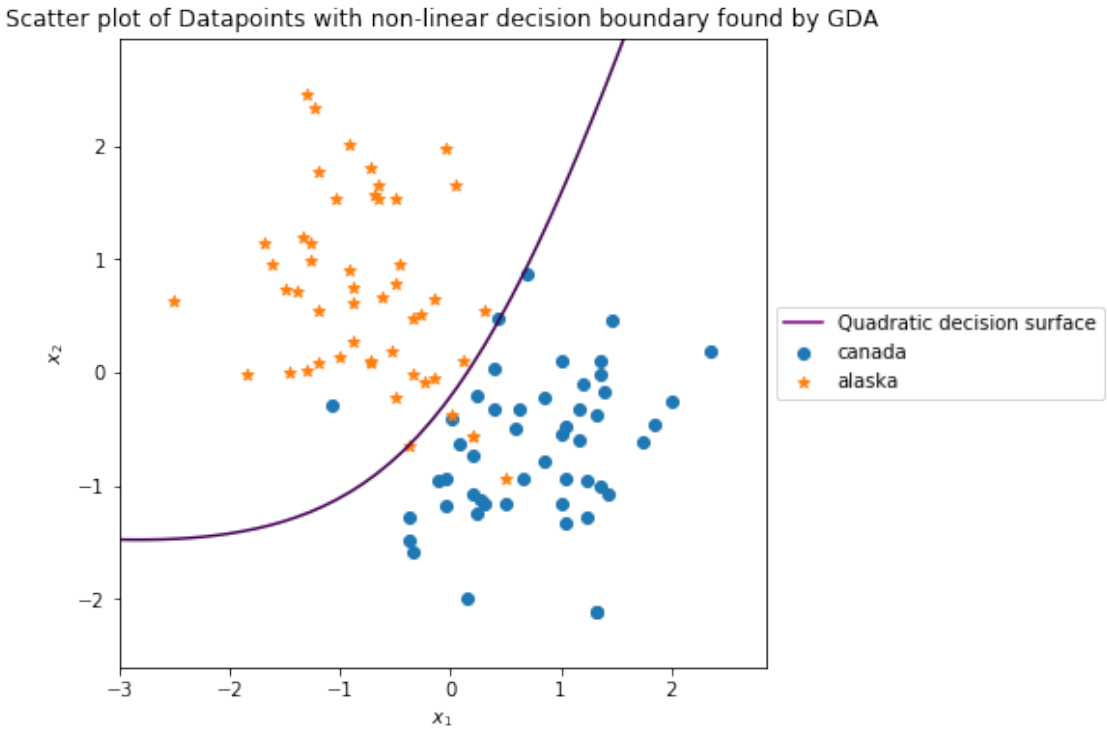
[13]:
```
def findCoeffs(phi, mu_0, mu_1, Sigma0, Sigma1):
    Sigma0_inv = np.linalg.inv(Sigma0)
    Sigma1_inv = np.linalg.inv(Sigma1)
    mu0_T = mu_0.T
    mu1_T = mu_1.T

    x_2_coeff = (1/2)*(Sigma1_inv - Sigma0_inv)
    a=x_2_coeff[0][0]
    b=x_2_coeff[1][1]
    c=x_2_coeff[0][1] + x_2_coeff[1][0]

    x_1_coeff = (mu0_T.dot(Sigma0_inv) - mu1_T.dot(Sigma1_inv))
    d=x_1_coeff[0]
    e=x_1_coeff[1]

    Sigma0_det = np.linalg.det(Sigma0)
    Sigma1_det = np.linalg.det(Sigma1)
    f = np.log(((1-phi)*math.sqrt(Sigma1_det))/(phi*math.sqrt(Sigma0_det))) +␣
↪(1/2)*(mu1_T.dot(Sigma1_inv).dot(mu_1) - mu0_T.dot(Sigma0_inv).dot(mu_0))
    return (a,b,c,d,e,f)
```

[14]:
```
coeffs = findCoeffs(phi, mu_0, mu_1, Sigma0, Sigma1)
```

[15]:
```
def f(x,y,coeffs):
    a=coeffs[0]
    b=coeffs[1]
    c=coeffs[2]
    d=coeffs[3]
    e=coeffs[4]
    f=coeffs[5]
    return a*(x**2) + b*(y**2) + c*x*y + d*x + e*y + f

def fcontour(f, xrange, yrange, coeffs=(0,0,0,0,0,0), **kwargs):
    """
    Draw the curve f(x,y) = 0 over the specified range.
    Arguments:
    f --- the function defining the curve
    coeffs --- co-efficients that we will pass to our function f defined in this␣
↪question
```

```
    """
    xs = np.linspace(xrange[0], xrange[1])
    ys = np.linspace(yrange[0], yrange[1])
    fs = [[f(x,y, coeffs) for x in xs] for y in ys]
    plt.contour(xs, ys, fs, [0], **kwargs)
    plt.axis('scaled')
```

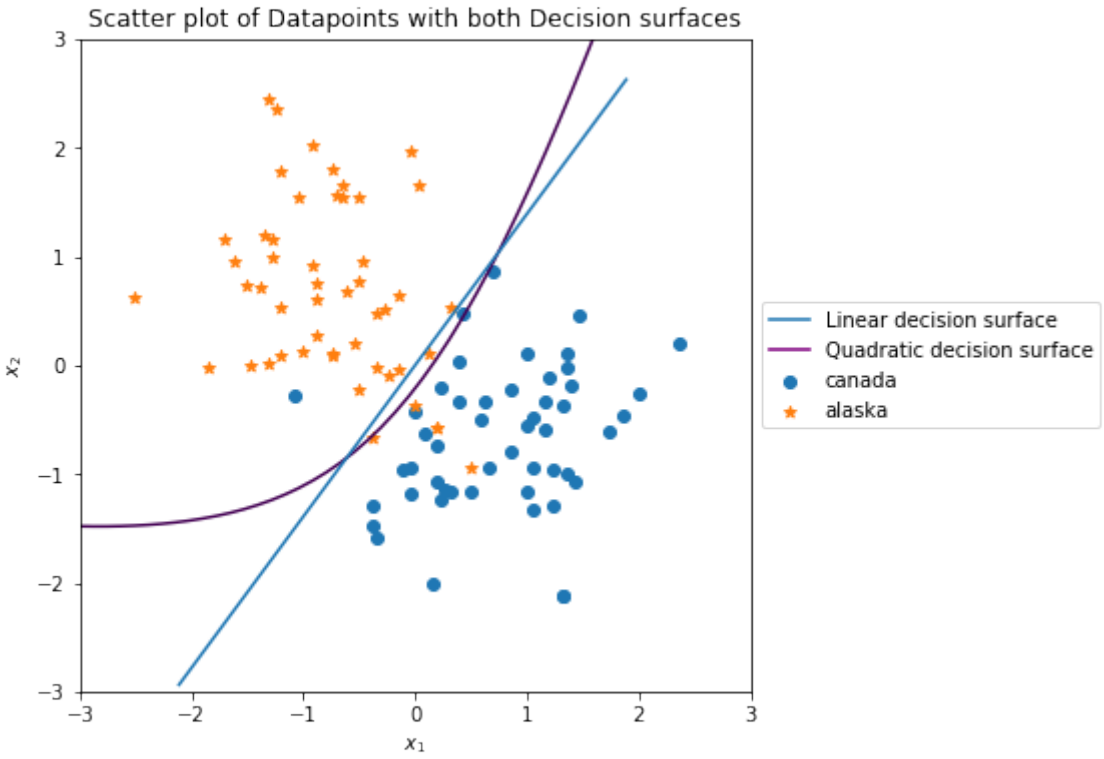
[16]:
```
fig = plt.figure(figsize=(6,6))
pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
plt.scatter(pos_X[:,0], pos_X[:,1], marker='o', label='canada')
plt.scatter(neg_X[:,0], neg_X[:,1], marker='*', label='alaska')

xs = np.linspace(-3, 3)
ys = np.linspace(-3, 3)
fs = [[f(x,y, coeffs) for x in xs] for y in ys]
cs = plt.contour(xs, ys, fs, [0])
plt.plot([],[],c='purple',label='Quadratic decision surface')

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.xlim((X[:,0].min()-0.5, X[:,0].max()+0.5))
plt.ylim((X[:,1].min()-0.5, X[:,1].max()+0.5))
plt.title('Scatter plot of Datapoints with non-linear decision boundary found␣
 ↪by GDA')
plt.show()
```



1.1.6 Part (f) : Comparing both decision boundaries

[17]:
```
fig = plt.figure(figsize=(6,6))

pos_X = X[np.where(Y==1)] #Getting datapoints with positive label
neg_X = X[np.where(Y==0)] #Getting datapoints with negative label
plt.scatter(pos_X[:,0], pos_X[:,1], marker='o', label='canada')
plt.scatter(neg_X[:,0], neg_X[:,1], marker='*', label='alaska')

X_line = np.arange(X[:,1].min(), X[:,1].max())
Y_line = -1*(1/theta[1])*((theta[0]*X_line)+theta_0)
plt.plot(X_line, Y_line, label='Linear decision surface')
plt.plot([],[],c='purple',label='Quadratic decision surface')
xs = np.linspace(-3, 3)
ys = np.linspace(-3, 3)
fs = [[f(x,y, coeffs) for x in xs] for y in ys]
plt.contour(xs, ys, fs, [0])


plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
```

```
plt.title('Scatter plot of Datapoints with both Decision surfaces')
plt.show()
```



Observations :

- Here we can see that non-linear decision surface classifies data better than linear boundary. We can see that for the points close to the decision surface.