Assignment 1

1. Linear Regression with Gradient Descent:

**Algorithm**

Initially I read the train and test data and stored them in a NumPy array, then I normalized the random variable X for train and test data to reflect mean = 0.0 and standard deviation to 1.0. Using the following code : **X=((X-X.mean())/X.std())**

Now I created a function named

**GradientDescent(X, Y, thetas=np.zeros(2), ita=0.001, JTheta=0.0):**

Here X and Y are the training data,

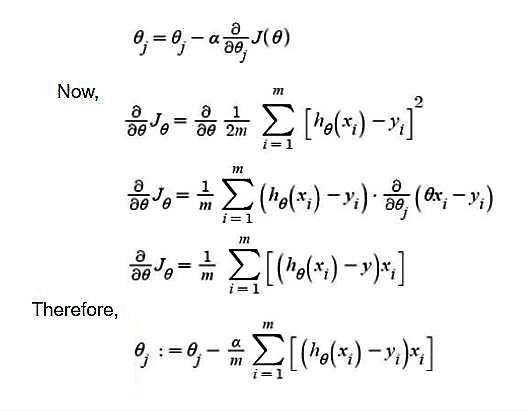
**thetas** are all initialized to 0.0

This function return the **thetas** after the model has been completely trained

Inside the function I started with an infinite loop. In each iteration, I calculate the hypothesis **h(θ) = np.matmul(thetas,X),** then cost using the following equation **J(θ) = (1/2\*m)(h(θ)-Y)²**

After this I check for the convergence criteria **|J(θ)i+1 - J(θ)i| < 0.00000001,** when it turns true, simply break from the infinite loop and return the thetas current value.

If the convergence criteria is not met then we update the thetas with the help of the following equation



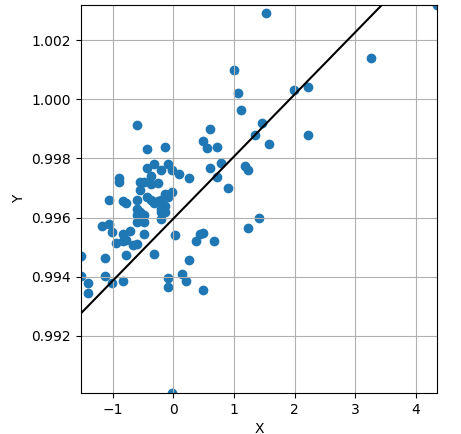
1.a) Learning rate which I chose for gradient descent: **0.1**

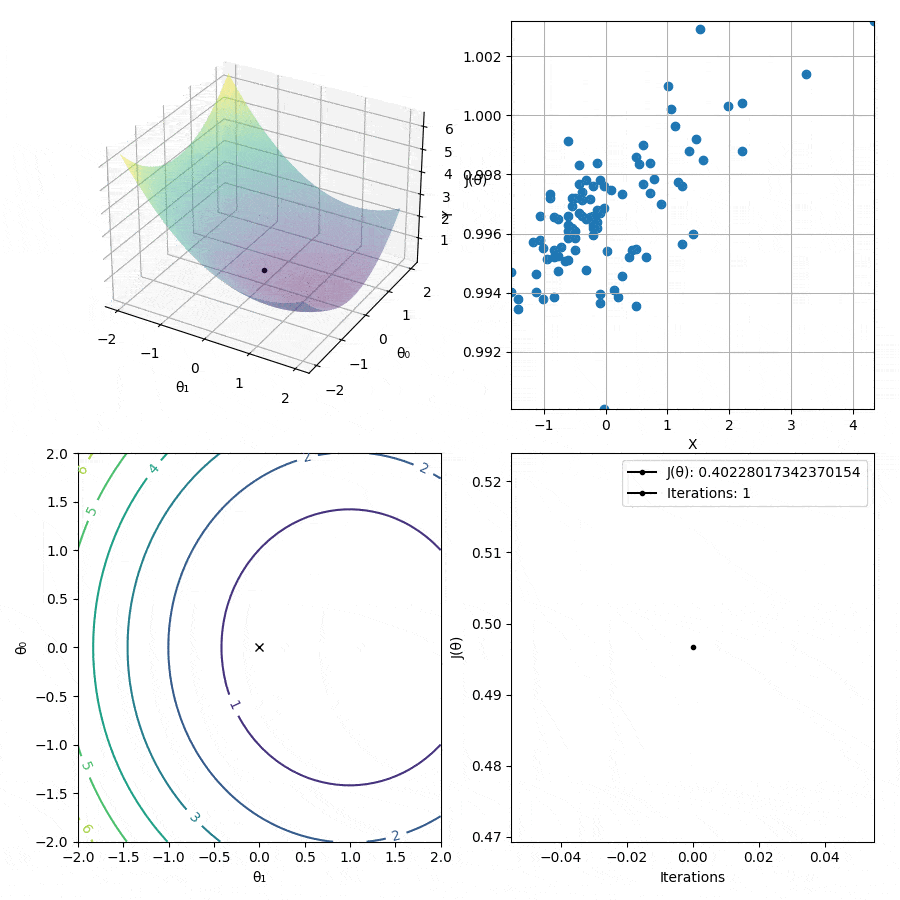
Convergence criteria : **|J(θ)i+1 - J(θ)i| < 0.00000001**

Final set of parameters : **θ₀ =** **0.99596584, θ₁ = 0.00210104**

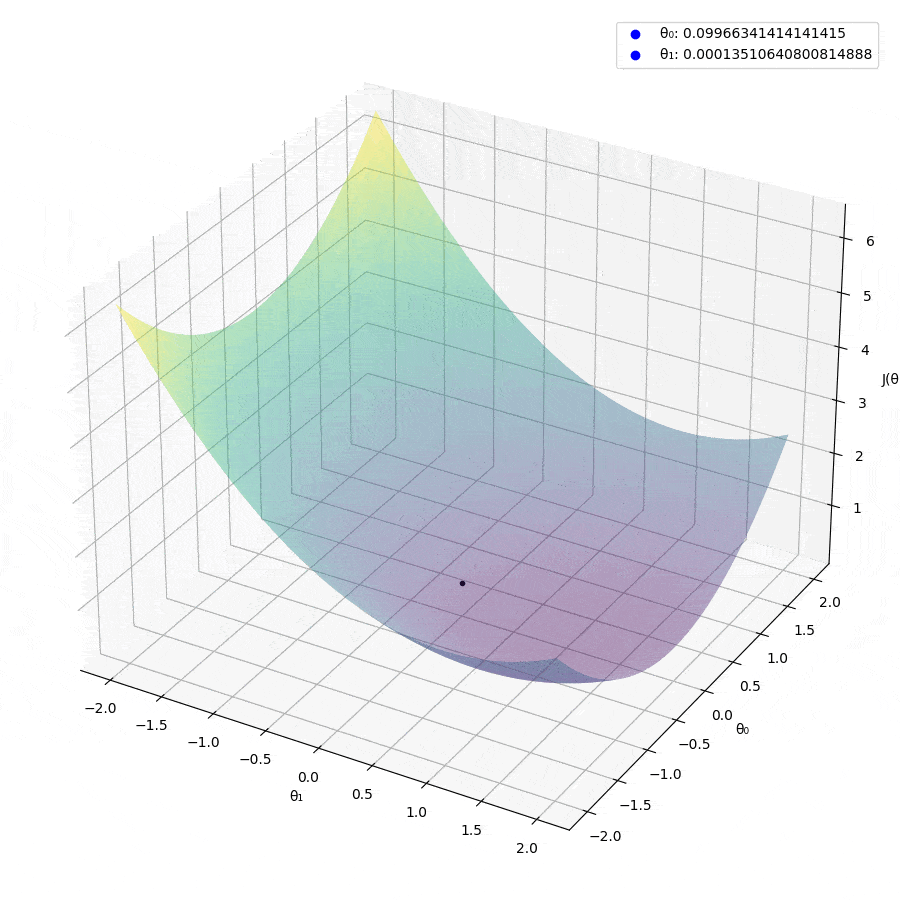
Final cost function value : **1.673696398896178e-06**

1.b)

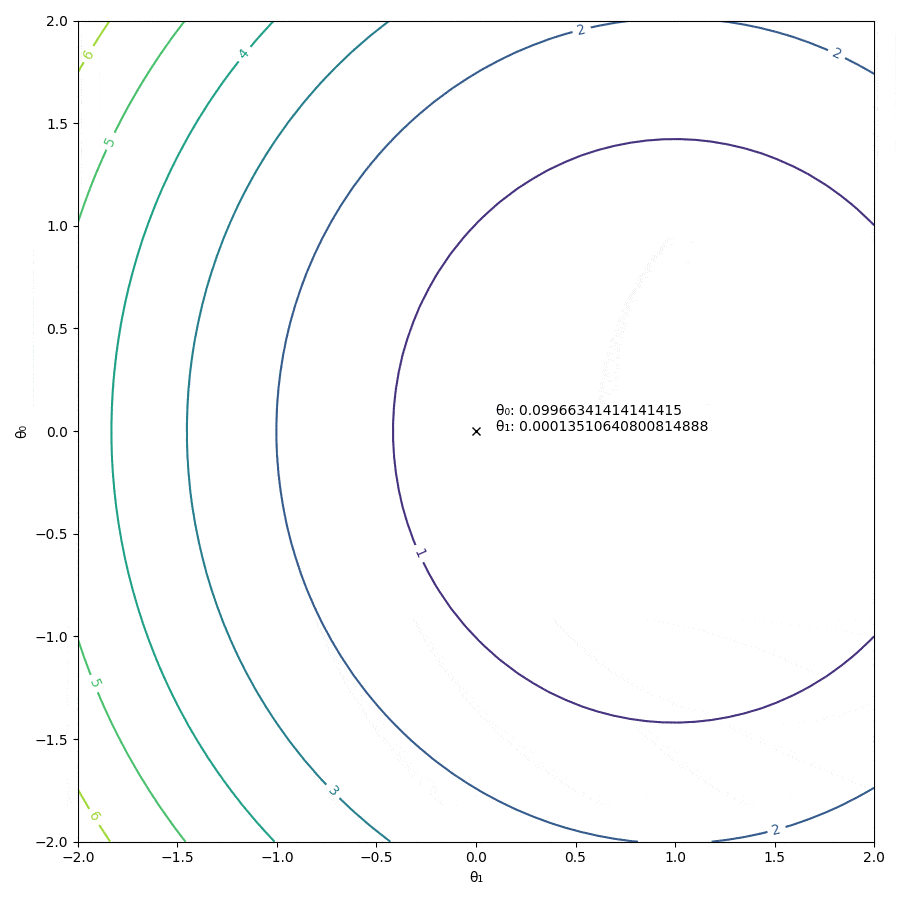




1.c)



1.d)



1.e) while learning data for different values for **η = [0.001, 0.025, 0.1],**  I observed that the they all

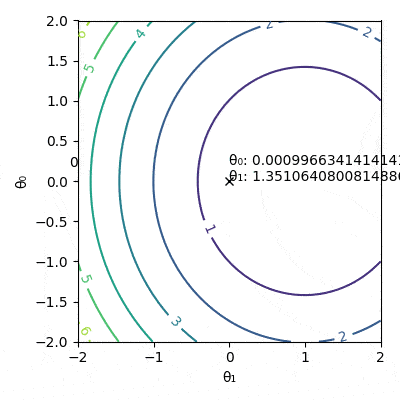
safely converge however as expected lower η value required much more time and iterations.

For **η=0.001**

**Executing time = 5.270089626312256 second**

**θ₀ =0.99346035, θ₁ = 0.00134242**

**Iterations = 5751**

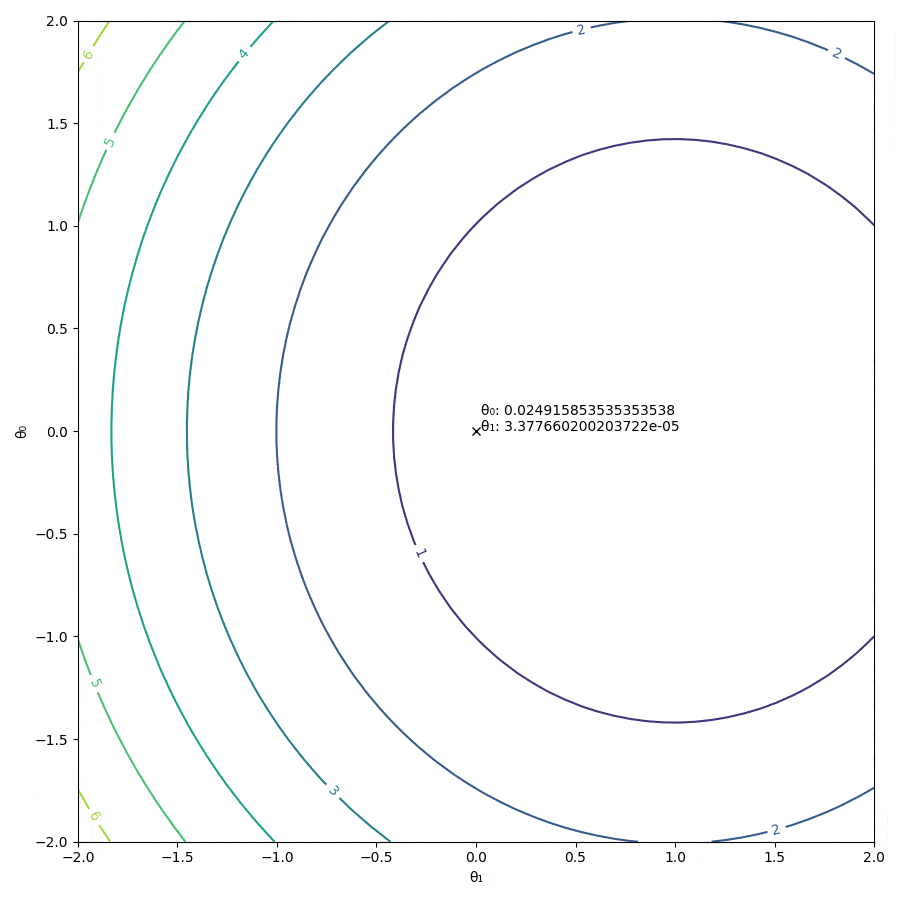
****

For **η=0.025**

**Executing time = 0.3139345645904541 second**

**θ₀ =0.99600651, θ₁ = 0.00134605**

**Iterations = 292**

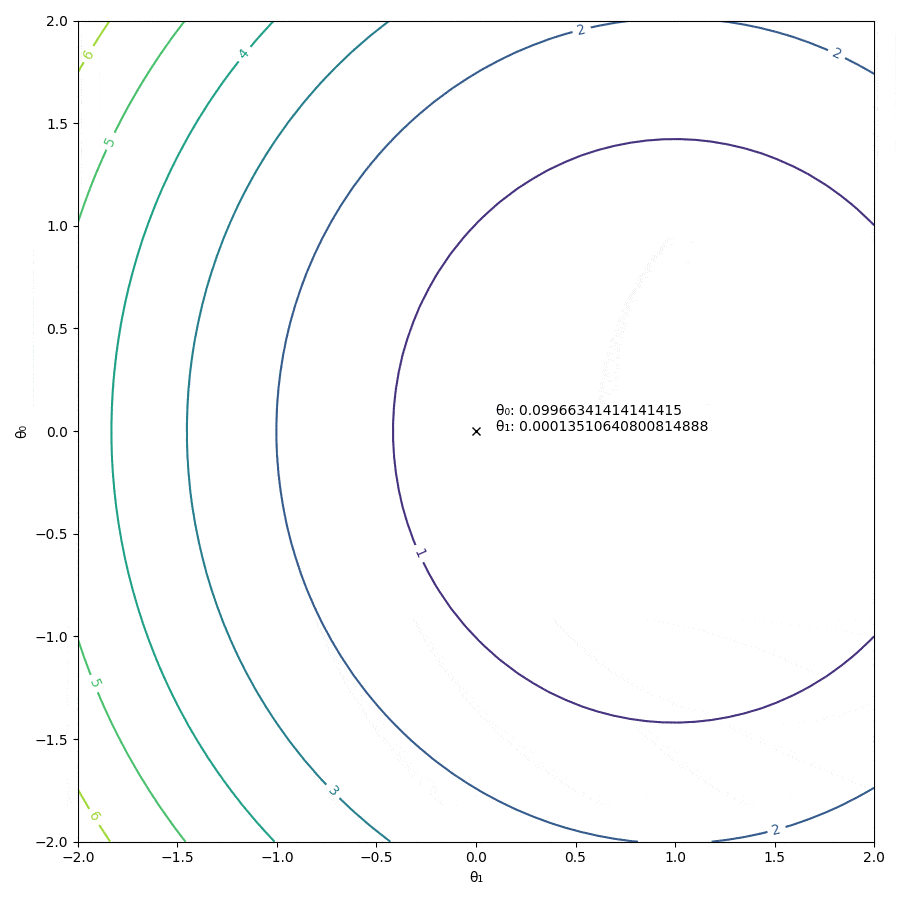
****

For **η=0.1**

**Executing time = 0.1031553745269775 second**

**θ₀ =0.99635129, θ₁ = 0.00134655**

**Iterations = 78**



1. Linear Regression with Stochastic Gradient Descent:

**Algorithm**

As required I randomly generated 1 million samples of training data using **np.random.normal(loc=3.0, scale=2.0, size=1000000)** here loc is the distribution mean and scale is SD, just like this I sample 3 random variables **X1, X2, epsilon(error)** and using these 3 Random variable Y is generated here is **Y=3+X1+2\*X2+epsilon**

**Stochastic Gradient Descent** is implemented exactly like normal gradient descent in terms of JTheta calculation and parameter updation via cost minimization. Their is variation only in two aspects firstly, the parameter updation and cost calculation is performaed after every ‘batch’ where **0 < batch size < 1000000** and secondly, the convergence criteria, For the given question the value of **η** has been fixed to **0.001** and hence no matter the convergence criteria the batch seize is directly proportional to the execution time until convergence.

I have experimented both ways with and without changing **η** and have publish the results in the document itself. For changing **η** following equations has been used:

**η= 1.0 / ( 0.5 \* iterations + 10)**

I have decided the following convergence criteria for each case:

**|J(θ)i+1 - J(θ)i| < 0.000000001 \* ( batchSize + iterations )**

Here JTheta is the cost calculated in the current iteration and PreJtheta is the cost calculated in the previous iteration just like in normal gradient descent.

**{ 0.000000001 \* ( batchSize + iterations ) }** this entire expression captures the intuition that initially for any given value of batchSize the convergence criteria is much stricter so it does not converge on accident.

Like for **batchSize=1**

when **iterations=1** convergence criteria will be **|J(θ)i+1 - J(θ)i| < 0.000000001** and as iterations increase the convergence criteria will become relaxed. This also guarantees the convergence of the algorithm.

When **iterations=100000** convergence criteria will become **|J(θ)i+1 - J(θ)i| < 0.0001.**

2.a) Implemented in the program

2.b) Implemented in the program

2.c) Setting the **η to 0.001,** my algorithm converge in each case: **r = {1, 100, 10000, 1000000}**

**For r = 1**

**θ₀: 2.988869576187893**

**θ₁: 0.9884431188592653**

**θ₂: 2.0305780152276656**

**iterations: 33304**

**Time taken(sec): 1.2686140537261963**

**For r = 100**

**θ₀: 3.0035321969151294**

**θ₁: 1.0015615552779418**

**θ₂: 1.9947172376572335**

**iterations: 29319**

**Time taken(sec): 5.481138467788696**

**For r = 10000**

**θ₀: 2.8498170478414764**

**θ₁: 1.0326582773734114**

**θ₂: 1.9888084690662902**

**iterations: 10697**

**Time taken(sec): 139.33497428894043**

**For r = 1000000**

**θ₀: 2.4658303876529392**

**θ₁: 1.2472837791713394**

**θ₂: 1.9167541561078172**

**iterations: 3295**

**Time taken(sec): 4937.229515552521**

**Observations**

We can see that as the value of r is increasing the number of iterations are decreasing

which is expected since given a fixed value of **η=0.001** each iteration will have r elements

to compute cost also the time is also increasing due to the same reason.

Testing the model I trained in question **2.c** for different values of **r** on the given testing dataset

**q2test.csv,** following results are obtained

Squared error for Batch of size **1** is **2.070294588876261**

Squared error for Batch of size **100** is **1.9702954712963525**

Squared error for Batch of size **10000** is **2.096354924143564**

Squared error for Batch of size **1000000** is **7.374213392531638**

2.d) Shape of moment for each case of r is relatively similar, we can observe that for smaller value

of r the moment is little jittery because we are updating thetas much frequently and it does

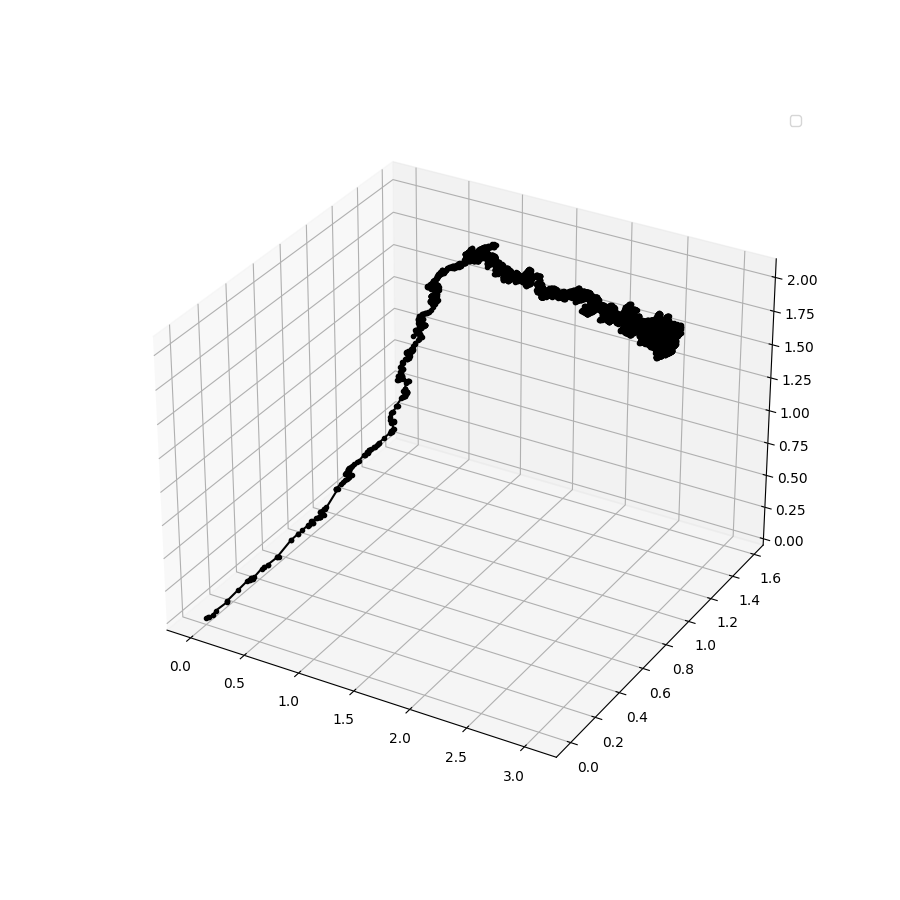
make **intuitive sense** because in any gradient descent algorithm the path followed will depend

on two factors initial position of the parameters and learning rate, in out case since both are

same for each case i;e, **η =** 0.001and **θs** are all initialized to 0.0

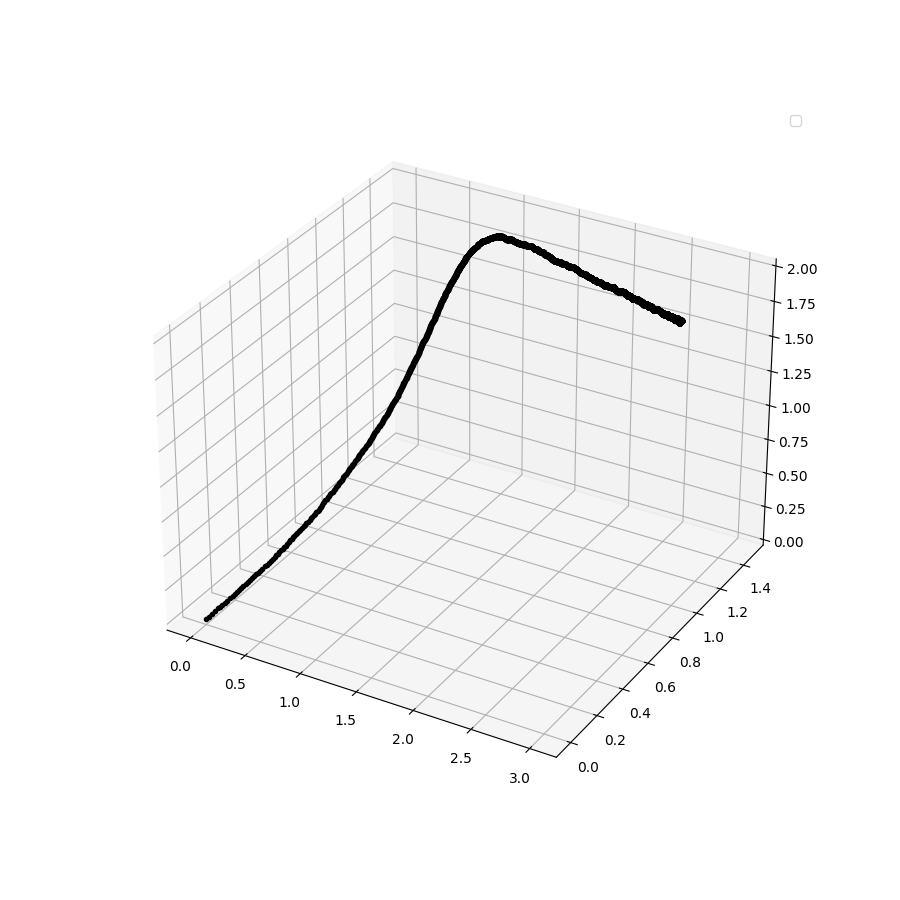
**For r = 1**

**iterations: 33304**



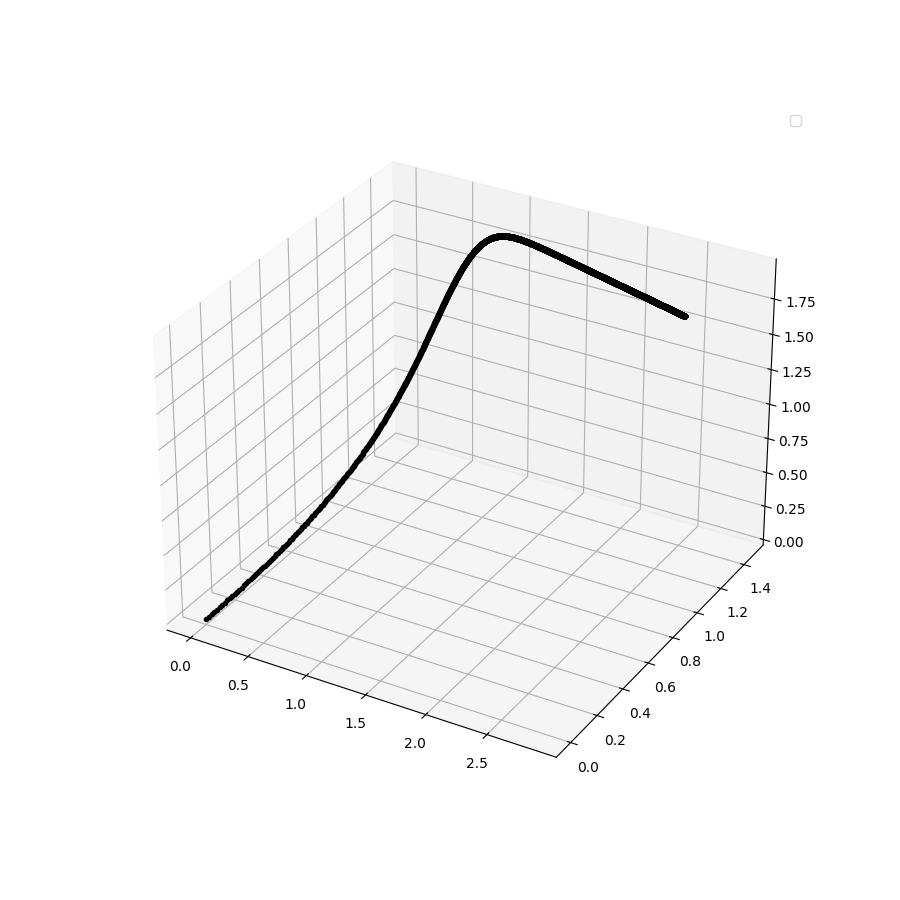
**For r = 100**

**iterations: 29319**



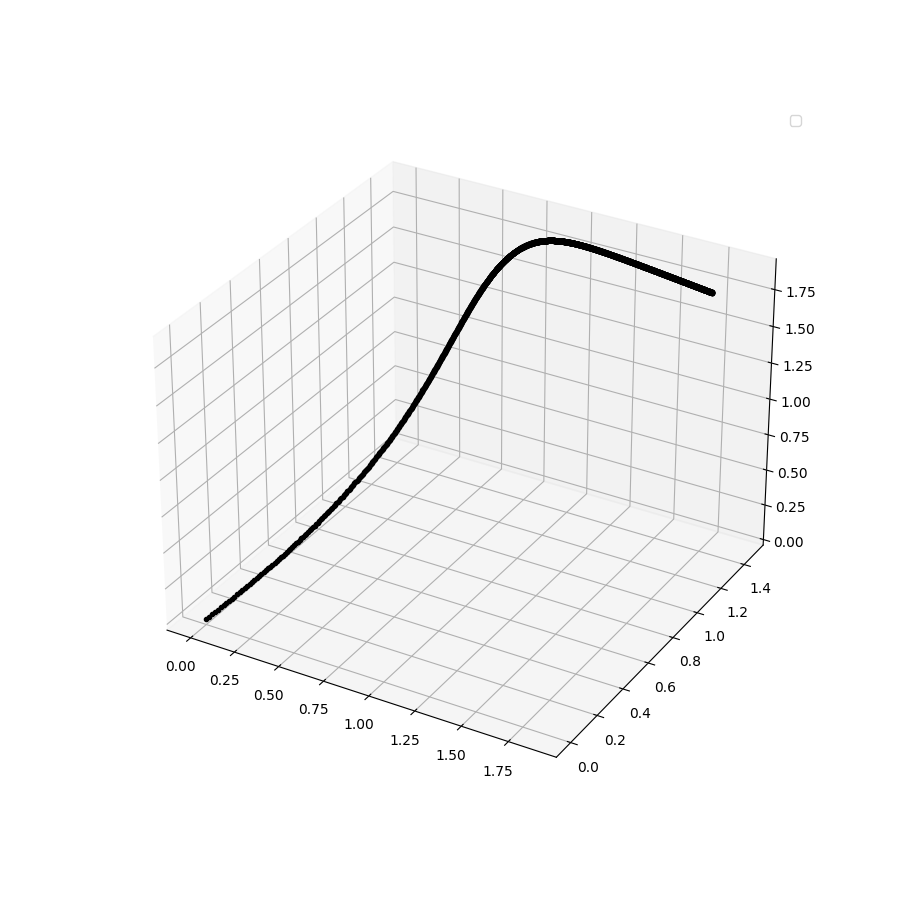
**For r = 10000**

**iterations: 10697**

****

**For r = 1000000**

**iterations: 3295**

****

I have also experimented with the changing **η**  as suggested by the Andrew NG which is

changing based on the following equations **η= 1.0 / ( 0.5 \* iterations + 10)**

**For r = 1**

**θ₀: 2.9990237995583433**

**θ₁: 0.9965381694913578**

**θ₂: 2.0080452480790187**

**iterations: 146273**

**Time taken: 7.635342836380005**

****

**For r = 100**

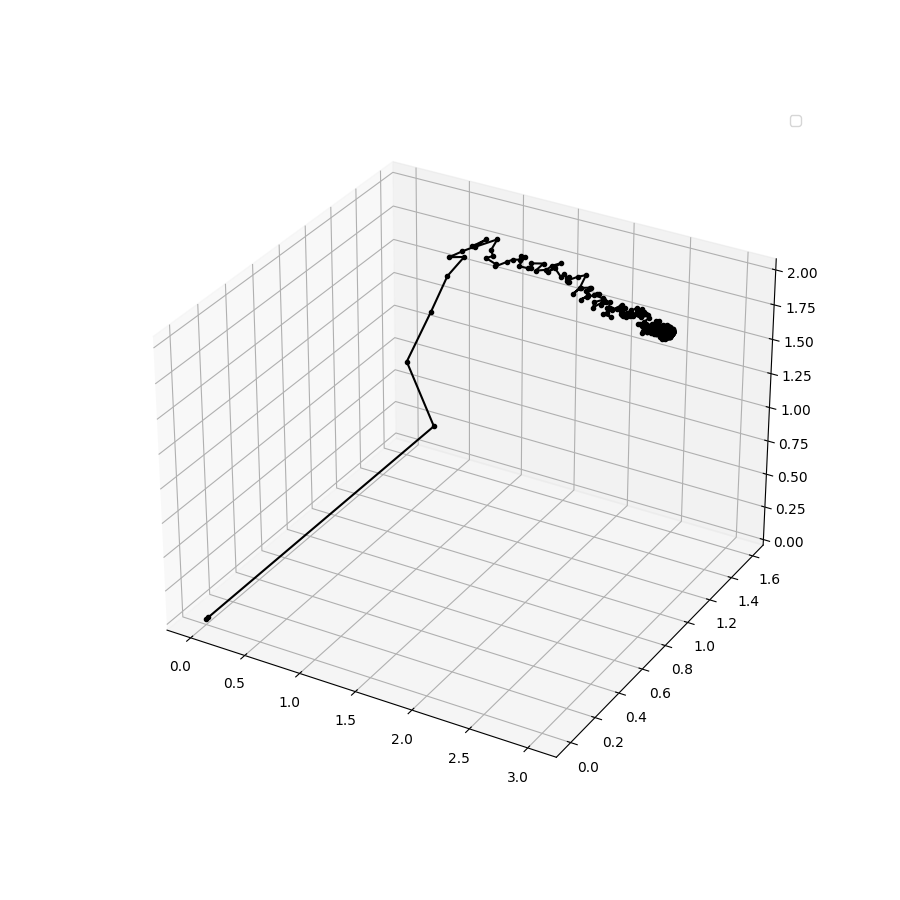
**θ₀: 3.0006887836180436**

**θ₁: 0.9979751351944088**

**θ₂: 2.0067364600408686**

**iterations: 11920**

**Time taken: 3.058948516845703**

****

**For r = 10000**

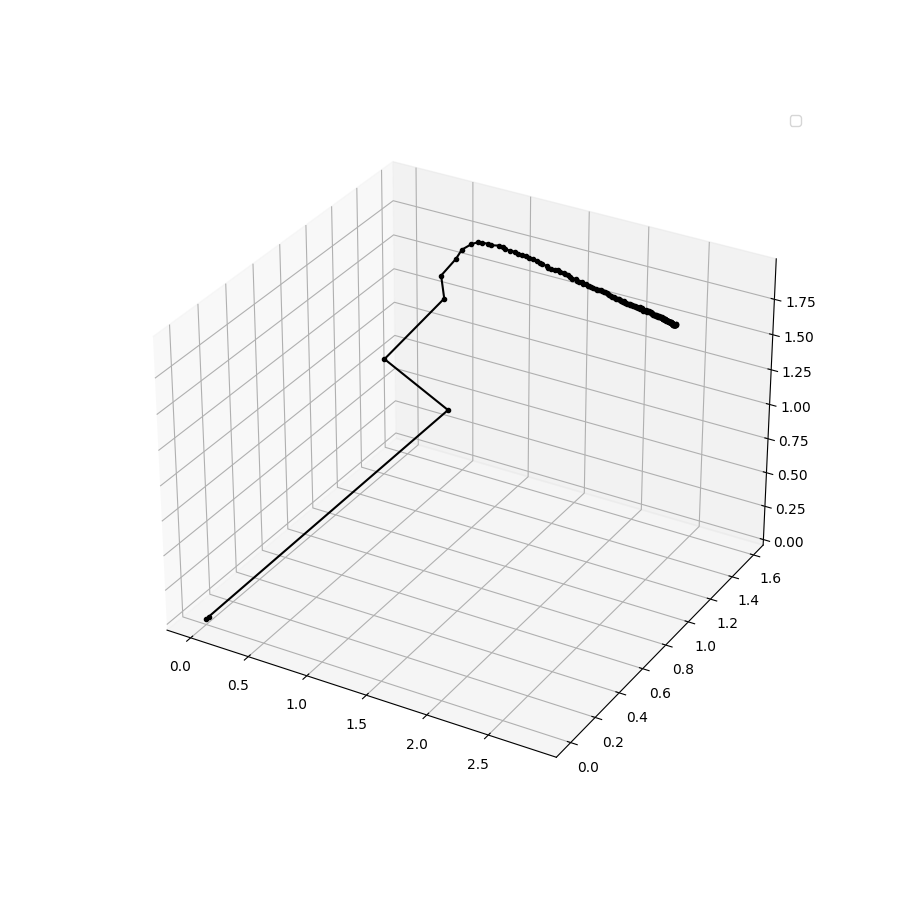
**θ₀: 2.8372994846309623**

**θ₁: 1.0359193574612098**

**θ₂: 1.9908785083499414**

**iterations: 185**

**Time taken: 3.4503045082092285**

****

**For r = 1000000**

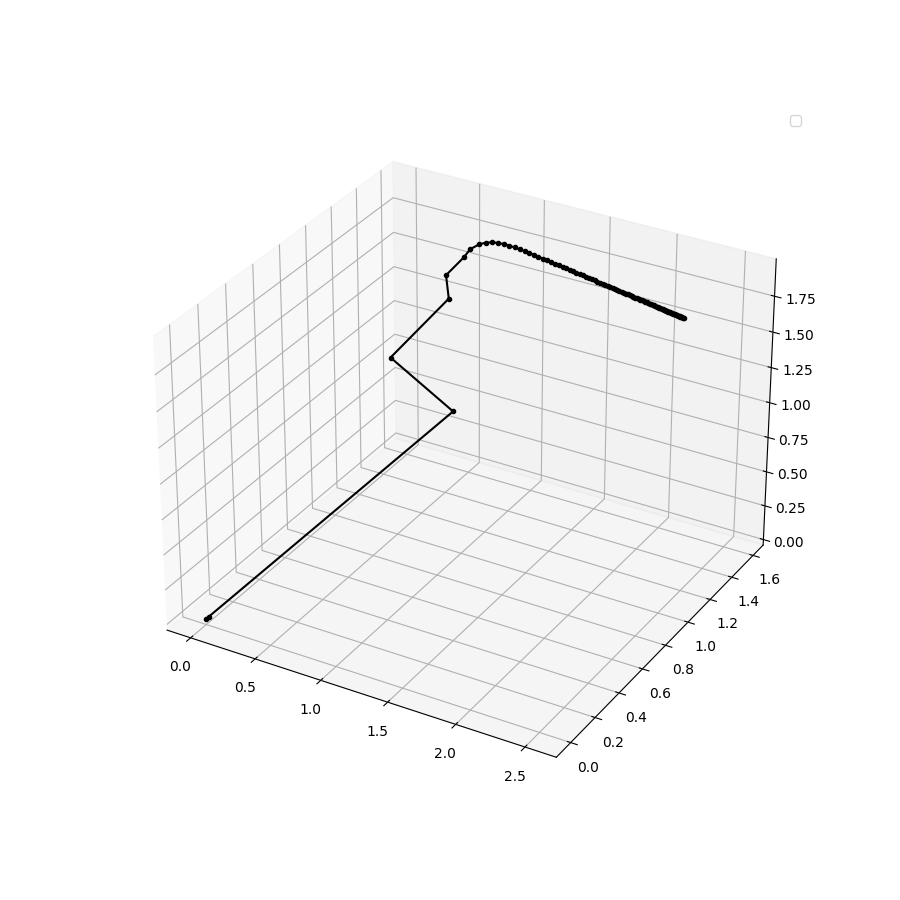
**θ₀: 2.538104086539994**

**θ₁: 1.1017599439980938**

**θ₂: 1.966478787492758**

**iterations: 94**

**Time taken: 200.86292171478271**

****

Squared error for Batch of size **1 i**s **1.9720586328921608**

Squared error for Batch of size **100** is **1.969359790473843**

Squared error for Batch of size **10000** is **2.1159349483211005**

Squared error for Batch of size **1000000** is **3.2140801969510844**

1. Logistic Regression

**Algorithm**

Initially I read the train and test data and stored them in a NumPy array, then I normalized the random variable X for train and test data to reflect mean = 0.0 and standard deviation to 1.0. Using the following code : **X=((X-X.mean())/X.std())**

Now I created a function named **logisticRegression()**

In the function I first calculated the hypothesis **h(θ) = 1.0/(1.0+np.exp(-Z))** by performing sigmoid squishification on Z where **Z=np.matmul(thetas,X)**

Now I calculated the log likelihood function **L(θ)** as mentioned in the question

**LTheta = np.multiply( Y, np.log( hTheta ) ) + np.multiply( ( 1 - Y ), np.log( 1 - hTheta ) )**

Now checked the convergence criteria **|L(θ)i+1 - L(θ)i| < 0.0000001**

When not converged, updated the **thetas** values, since It was asked to perform optimization

using Newton’s method, I calculated the Hessian matrix, We know that Hessian is a square

matrix of secord order partial derivatives of a multinomila function.

**L(θ)** = log( h(θ) ) + (1-) log( 1- h(θ) )

Partial differentiating **L(θ)** with respect to **θ** we get the following equation

= (- h(θ) )

Again differentiating **L’(θ)** with respect to **θ** we get

= - h(θ)( 1 - h(θ))

Finally **θs** are updating uing the following equation

**()i+1 = ()i +**

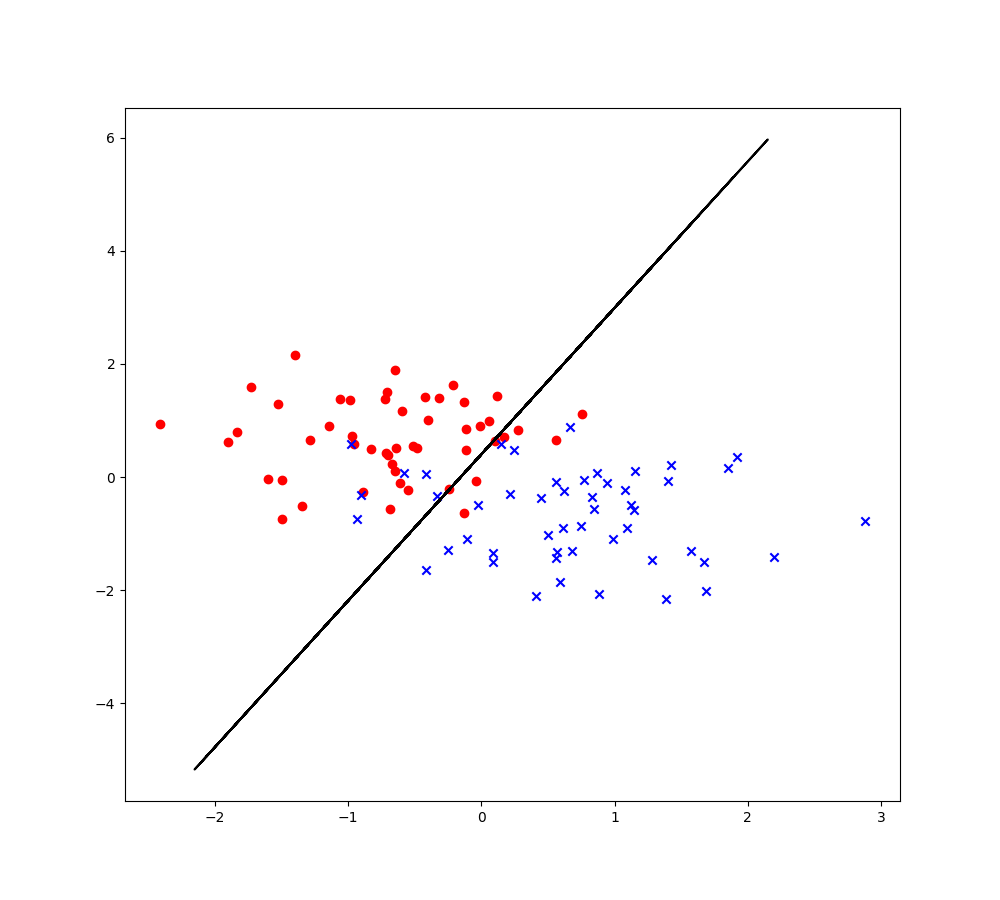
3.a) After running the algorithm it converges to the following **θ** values

**θ2** : -2.72558849

**θ₁** : 2.5885477

**θ₀** : 0.40125316

3.b) In the graph Red circle denotes category ‘0’ and Blue cross denotes category ‘1’



1. Gaussian Discriminant Analysis

**Algorithm**

Initially I read the train and test data and stored them in a NumPy array, then I normalized the random variable X for train and test data to reflect mean = 0.0 and standard deviation to 1.0. Using the following code : **X=((X-X.mean())/X.std()),** for training data I have read the categories values from **q4y.csv** and then implemented the indicator function for both when Y=**’Canada’** and when Y=**’Alaska’.**

SDA is a non iterative method for data classification and here we are trying to fit a best gaussian distribution to our data, so I just calculated gaussian parameters **()** for each category given in training data in the code. using the following equations.

k = =1}

k =

Σk(- k).(- k)T

4.a) In the closed form equation with the assumption that Σ0=Σ1=Σ the estimated values of 0, 1,

Σ0=Σ1=Σ I am getting after running the algorithm are

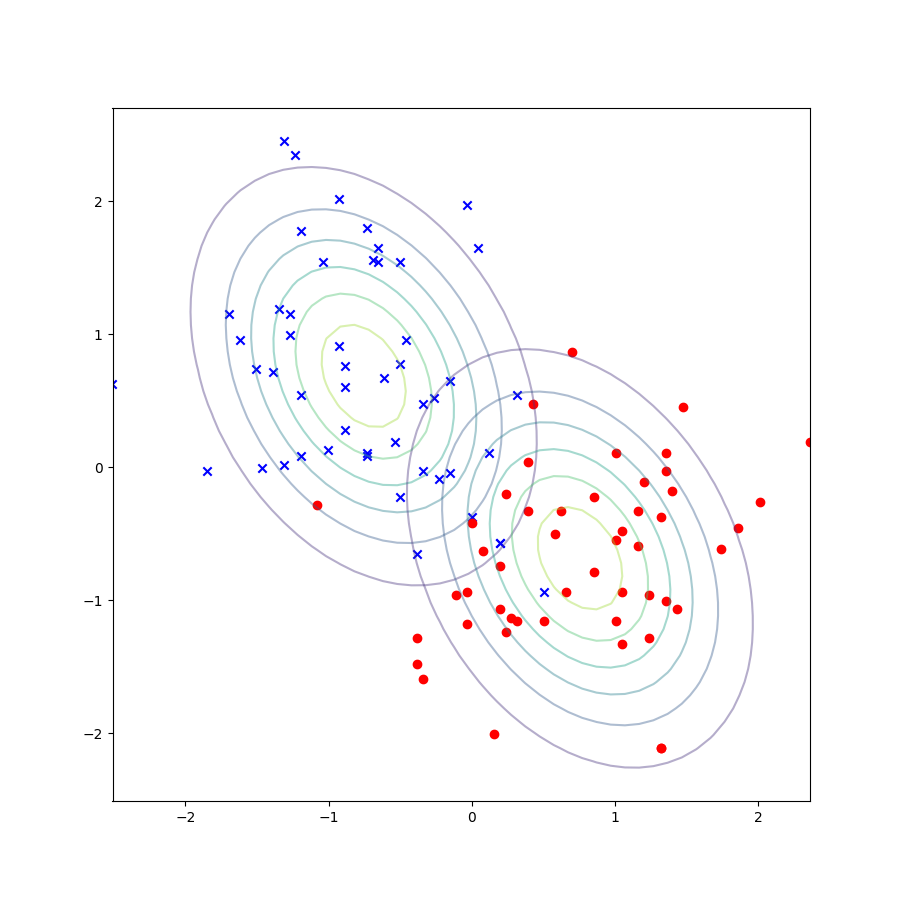
0 = [-0.75529433, 0.68509431]

1 = [ 0.75529433, -0.68509431]

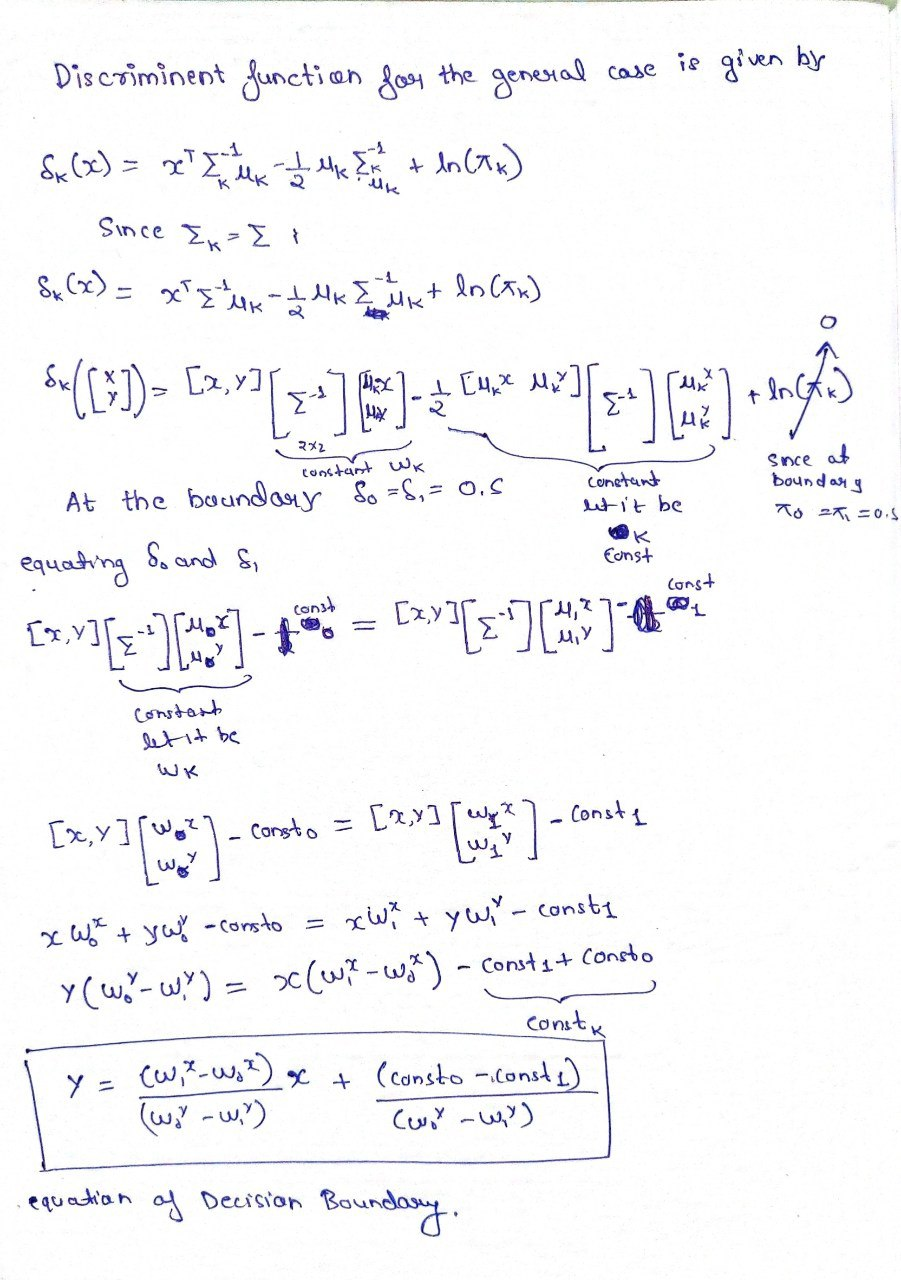
Σ0 = Σ = [[ 0.38158978 -0.15486516]

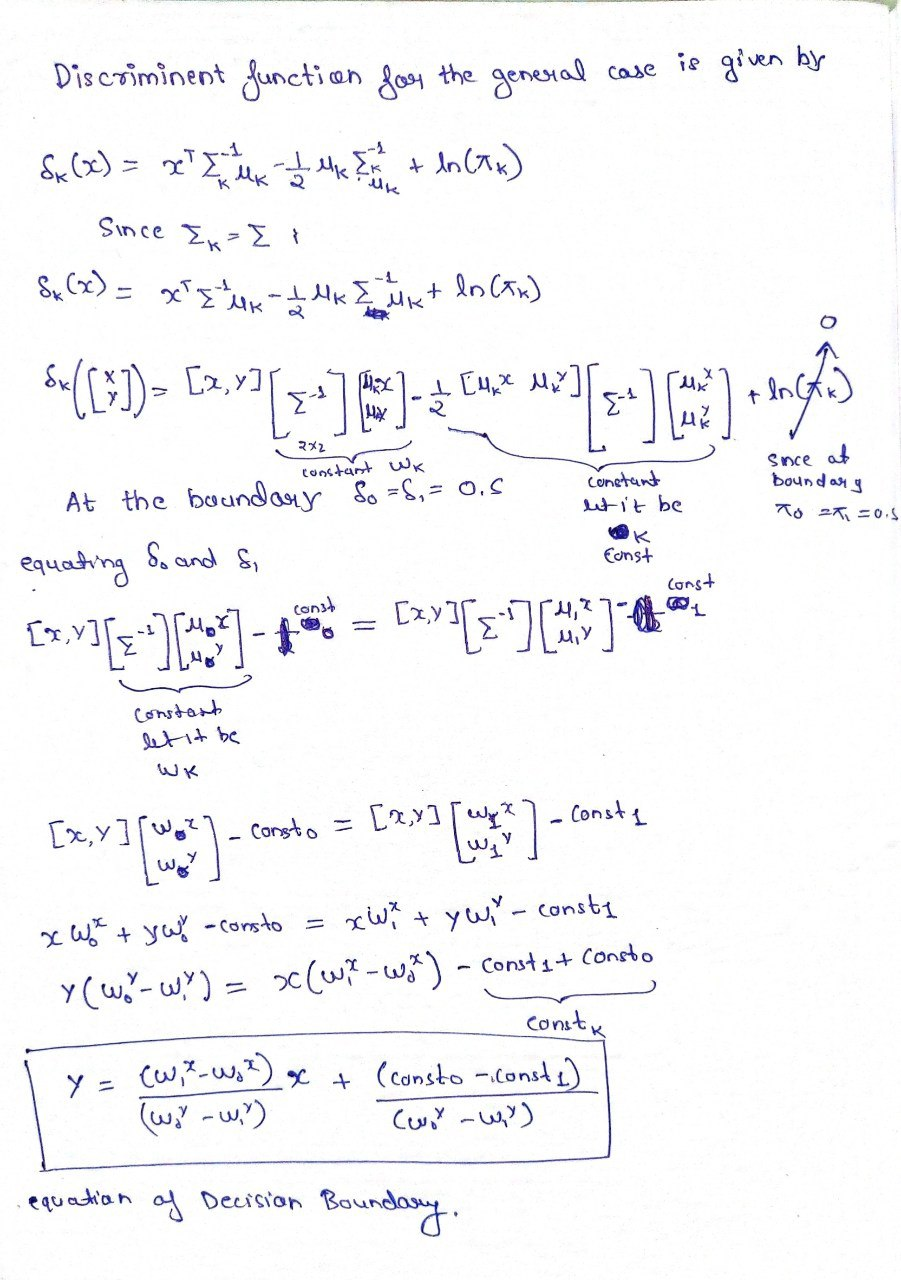
[-0.15486516 0.64773717]]

4.b) Plot of distributions along with contour lines



4.c) Derivation of the equation of decision boundary

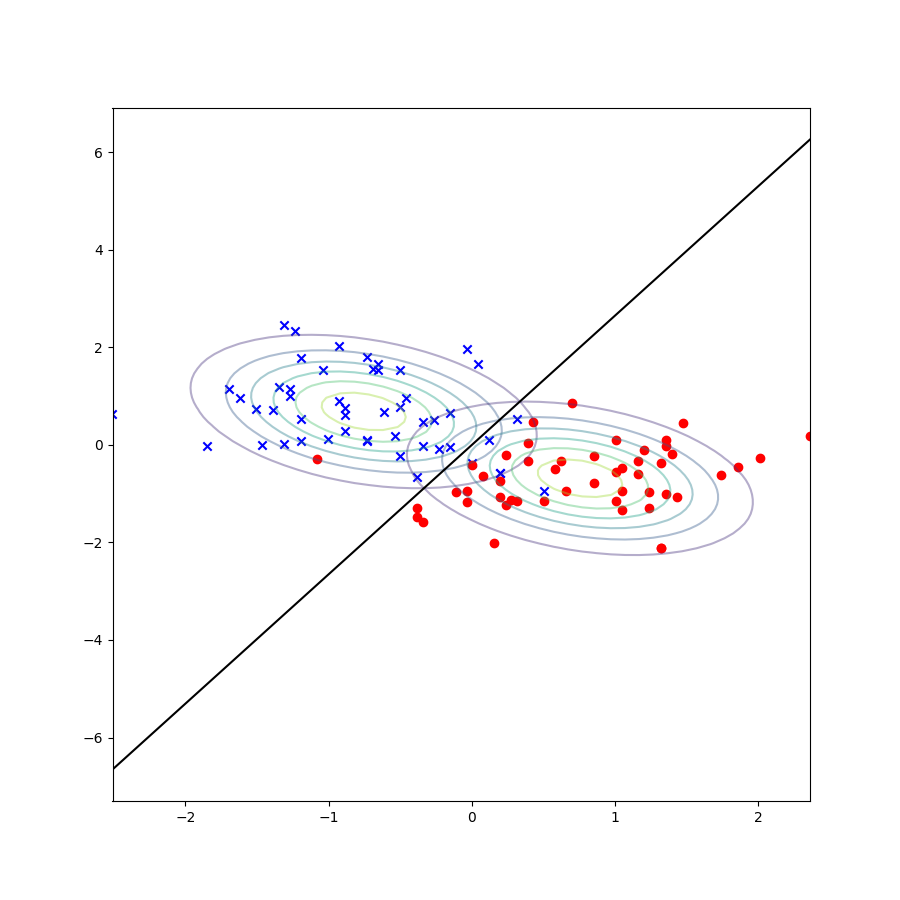




Plotting the linear decision boundary taking

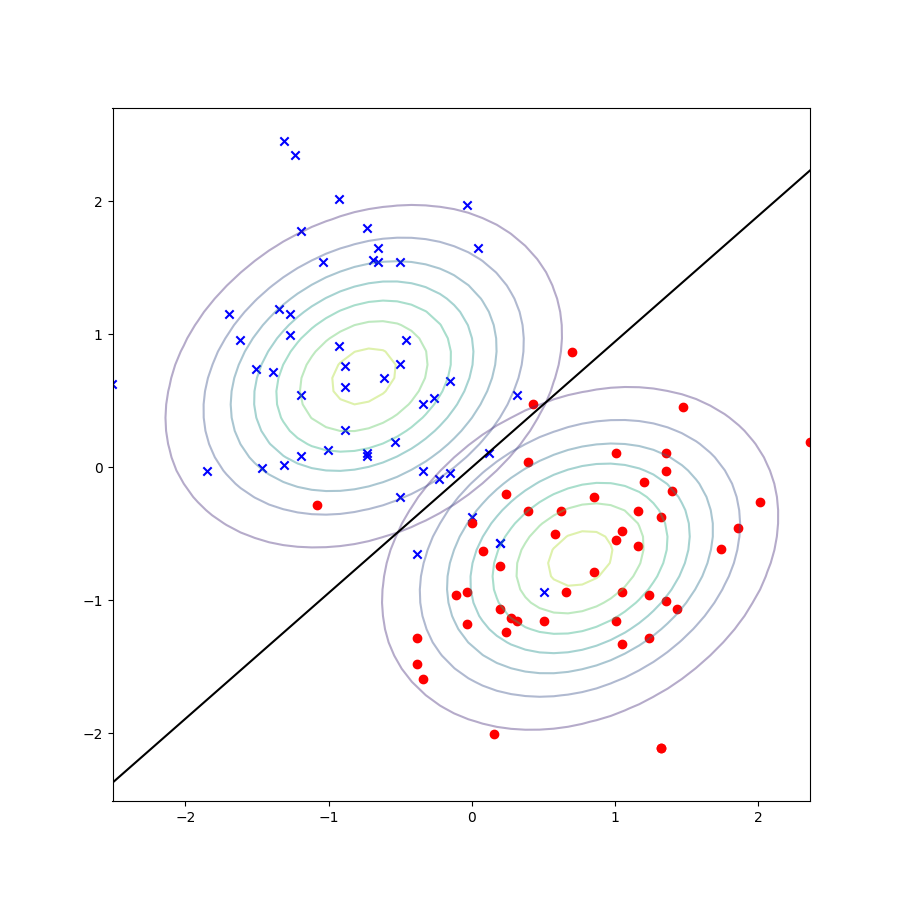
Σ0 = Σ = [[ 0.38158978 -0.15486516]

[-0.15486516 0.64773717]]



Σ0 = Σ1 = [[0.47747117 0.1099206 ]

[0.1099206 0.41355441]]



4.d) For Quadatic boundary separator for each category of the data we find gaussian parameters

**()** for each category

0 = [-0.75529433, 0.68509431]

1 = [ 0.75529433, -0.68509431]

Σ0 = [[ 0.38158978 -0.15486516]

[-0.15486516 0.64773717]]

Σ1 = [[0.47747117 0.1099206 ]

[0.1099206 0.41355441]]