COL774 – Machine Learning (Assignment-1)

Prakhar Aggarwal
2019CS50441

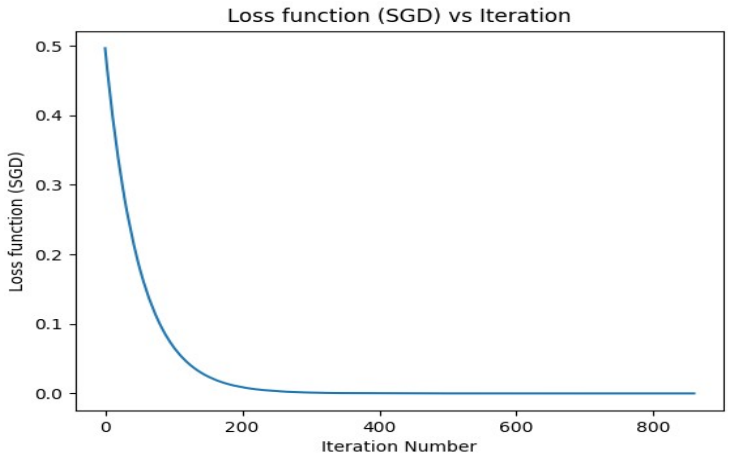
Part 1 (Linear Regression):

In this part, we implemented least square linear regression where x was a single feature list. So, our hypothesis function ($h_\theta(x) = \theta^T x$) is a straight line given by: $y = \theta_1 x + \theta_0$. We learn the parameters θ_1 and θ_0 by minimizing the least square error and updating the θ parameter using gradient descent. (Initially, $\theta = [0, 0]$)

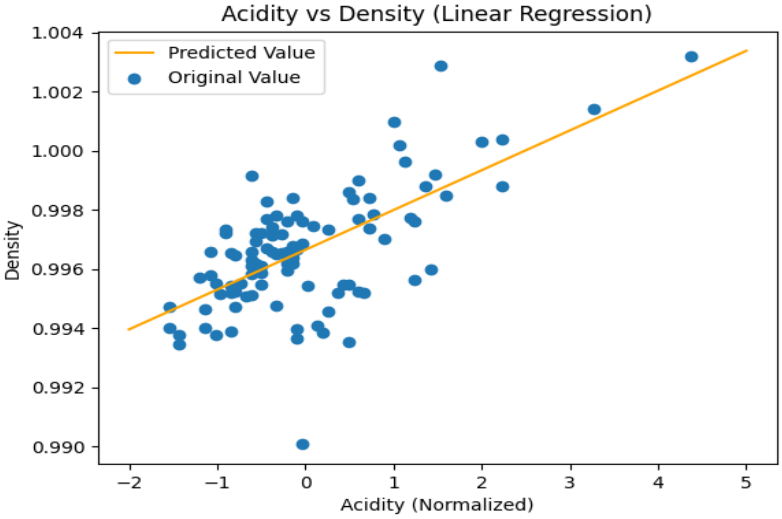
The stopping criteria is three-fold.

- There is a hard bound on the number of **iterations** on which our θ parameter will be updated
- We have a value **delta** which conditions over the change in cost function $J(\theta^t) - J(\theta^{t-1}) <= \textbf{delta}$. Delta is taken to be 1e-8 for good results.
- There is a value named **Min_Loss** which makes sure that the cost should be atmost **Min_Loss** i.e. loss1 <= Min_Loss and loss2 <= Min_Loss. This is done so as to avoid convergence due to small increment in J value even with a large cost.

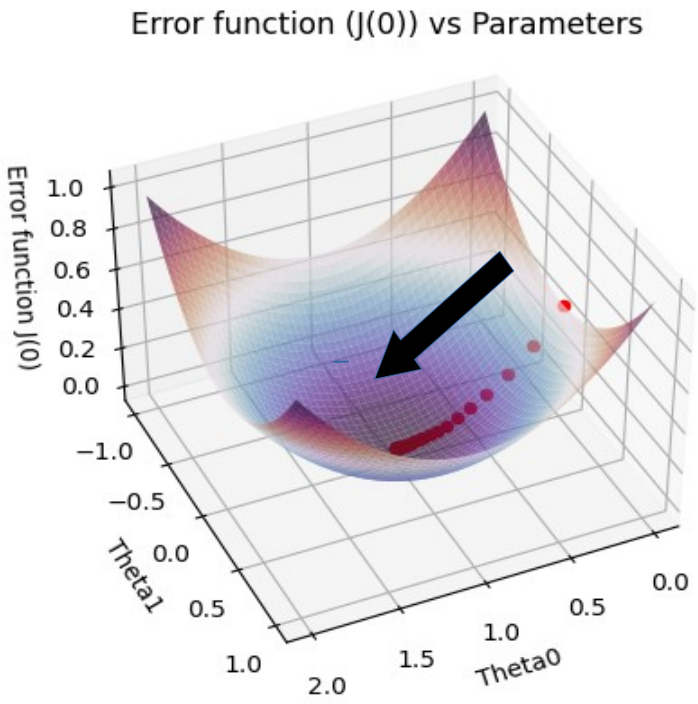
a) Learning Rate: 0.01
Stopping Criteria: $J(\theta^t) <= \text{Min\_Loss}$ and $J(\theta^{t-1}) <= \text{Min\_Loss}$ and $J(\theta^t) - J(\theta^{t-1}) <= \text{delta}$
Parameters: [0.9966491, 0.0013466]



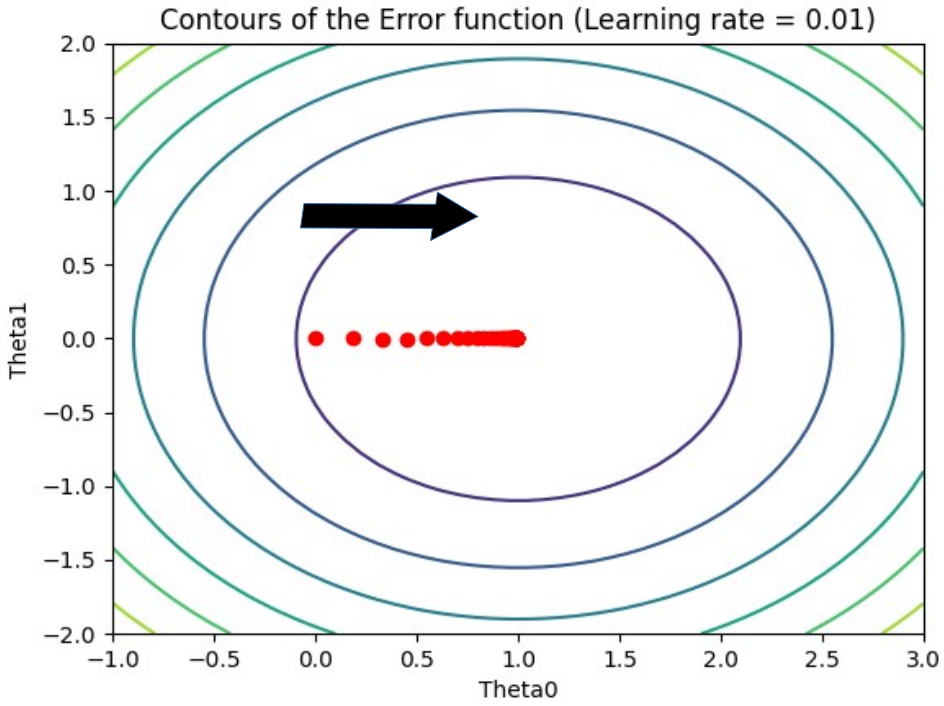
b)



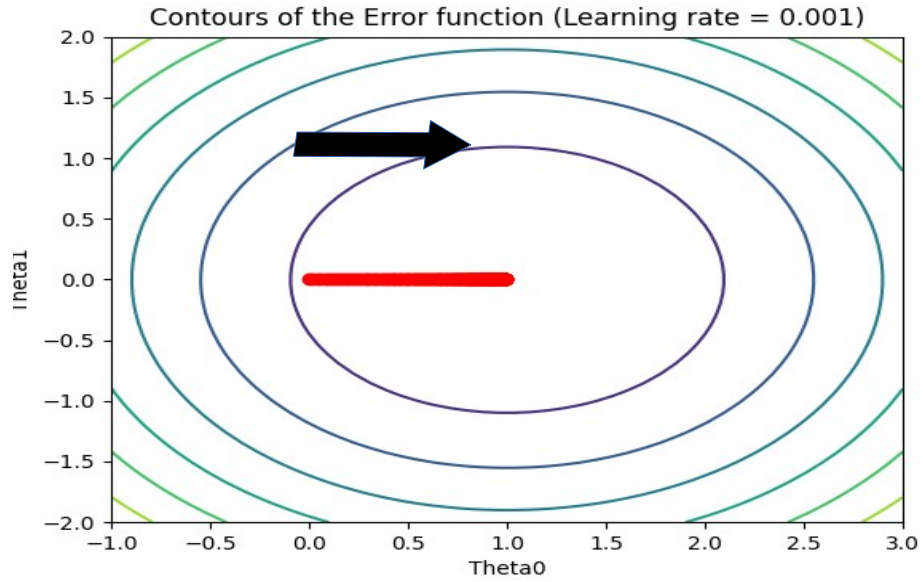
c) Points move towards the minima of the curve (direction of the arrow)

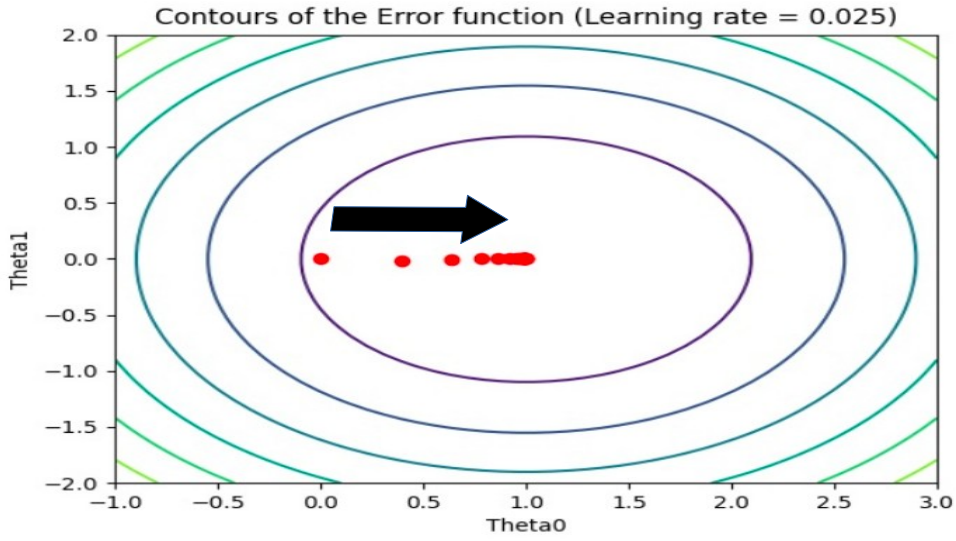


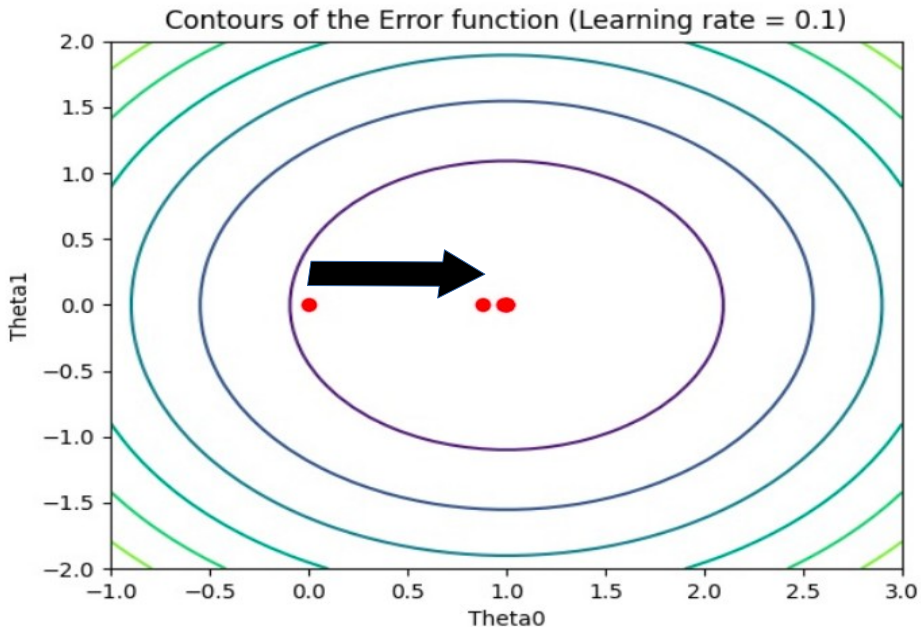
d) Points move towards the minima of the curve (direction of the arrow)



e) On increasing η from 0.001 to 0.1, theta parameters are learned at a faster rate. This is because, the jump in theta (parameter updation) is determined by the learning rate. We generally prefer small learning rate so as to avoid skipping the point of minima. If we would have used larger learning rate (like 2, 2.5, 10), then there is high possibilty that we will diverge from the local minima.







Part 2 (Sampling and Stochastic Gradient Descent):

In this part, we sample 1 million data points using $\theta = [\theta_0 \ \theta_1 \ \theta_2] = [3\ 1\ 2]$ and try to learn these θ parameters by Stochastic Gradient Descent (SGD) with different batch size. SGD is different from Gradient Descent in the sense that now we don't go through all m inputs to update our theta parameter but rather just go through r samples. This leads to faster convergence.

We will make a complete pass through the data in a round robin fashion (after shuffling the data)

a) Points are generated using inbuilt numpy function np.random.normal.

b) Convergence criteria (3 fold):

- There is a hard bound on the number of **iterations** on which our θ parameter will be updated (different for different batch size)
- We have a value **delta** which conditions over the change in cost function $J(\theta^t) - J(\theta^{t-1}) <= \mathbf{delta}$. Delta is different for different batch size.
- There is a value named **Min_Loss** which makes sure that the cost should be atmost **Min_Loss** i.e. loss1 <= Min_Loss and loss2 <= Min_Loss. This is done so as to avoid convergence due to small increment in J value even with a large cost.

  Original θ = [3, 1, 2]
  Batch Size: 1
  - Iterations: 100000
  - Delta: 1e-10
  - θ = [2.98796175 0.99441557 1.99418404] (terminated due to Iteration_Limit)
  - $Loss_1$: 1.0509, $Loss_2$: 1.0089

  Batch Size: 100
  - Iterations: 50000
  - Delta: 1e-9
  - θ = [3.00136287 0.99909424 1.99547033] (terminated due to Iteration_Limit)
  - $Loss_1$: 0.9976, $Loss_2$: 0.9975

  Batch Size: 10000
  - Iterations: 1000
  - Delta: 1e-3
  - θ = [0.87960143 1.45077856 1.8075334] (terminated due to Iteration_Limit)

- $Loss_1$: 1.6362, $Loss_2$: 1.6401

Batch Size: 1000000

- Iterations: 50
- Delta: 0.1
- θ = [0.15282246 0.59460321 0.22623141] (terminated due to Iteration_Limit)
- $Loss_1$: 10.4568, $Loss_2$: 10.3528

c) Algorithms with batch size 1 and 100 visibly converge to [3, 1, 2] (original parameters). For batch size, 10000 and 1000000, it will converge to the original parameters given the algorithm is run for sufficient time (more than 2-3 hrs)

For batch size 1, time to converge: 6.88 sec (100000 iterations)
For batch size 100, time to converge: 25.36 sec (50000 iterations)
For batch size 10000, time to converge: 39.35 sec (1000 iterations)
For batch size 1000000, time to converge: 206.10sec (50 iterations)

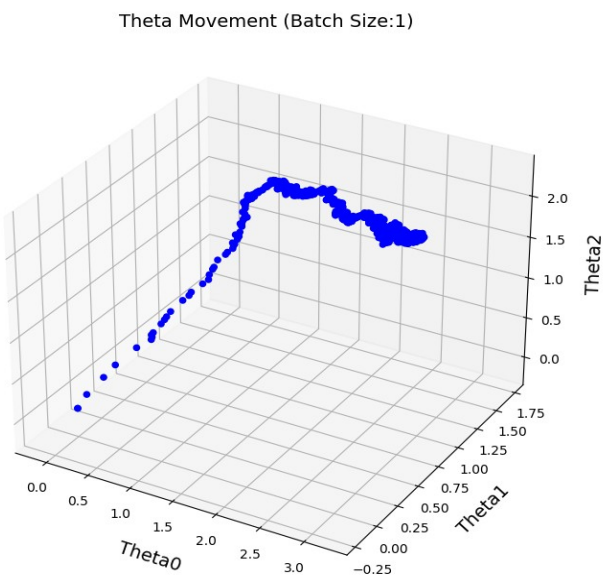
Error with original theta = [3, 1, 2] = 0.9825
Error with learned theta:

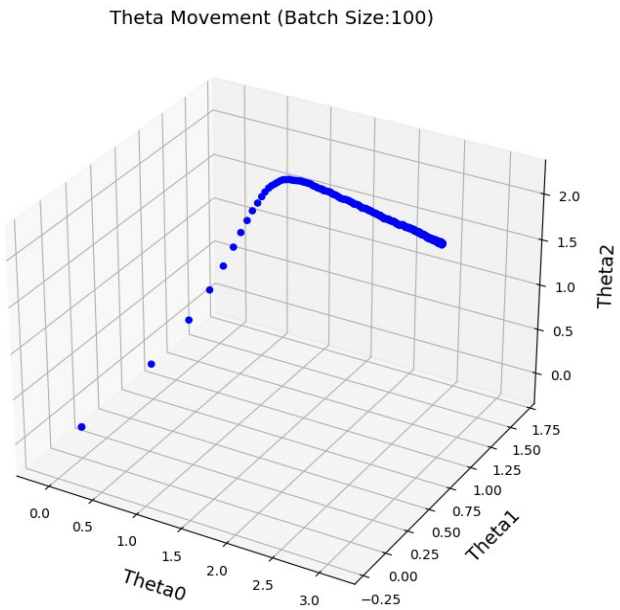
Batch Size: 1,	Error: 1.11007	Diff: 0.12757
Batch Size: 100,	Error: 0.9847	Diff: 0.0012
Batch Size: 10000,	Error: 14.00686	Diff: 13.02436
Batch Size: 1000000,	Error: 168.5889	Diff: 167.6064

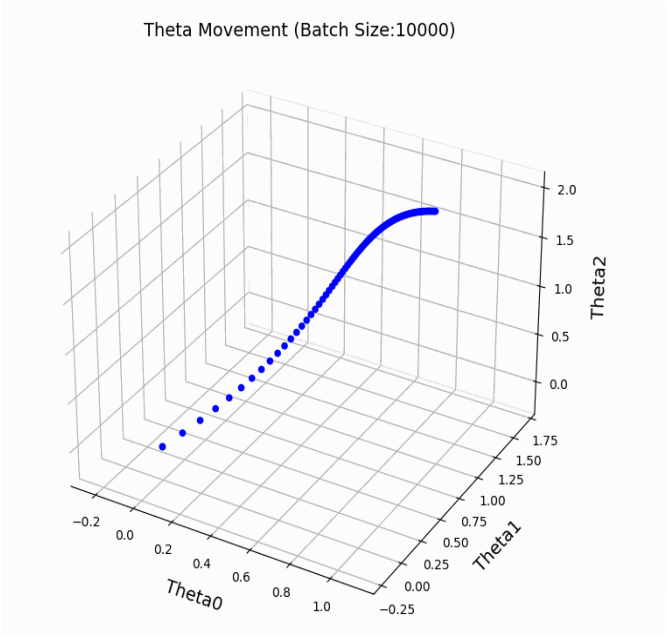
Error in case of batch size 1 and 100 is nearly equal to error with the original parameters. Whereas, in case of batch size 1e4 and 1e6, there is a significant error.

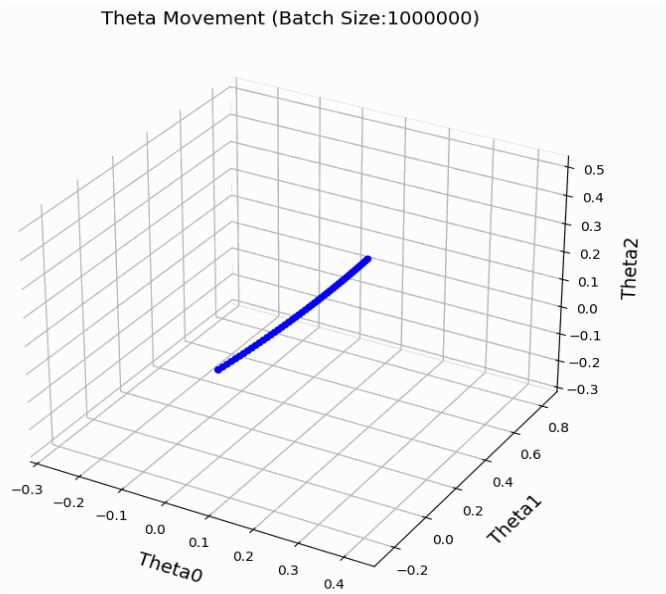
Also, model converges quite fast to the original parameters in case of batch size 1 and 100.

d)









Movement in each case:

For model with batch_size = 1, movement is zig zag because we are updating the theta parameters for each example (favours the particular example while updation) and not over a batch.

Batch Size: 100, movement is slightly zig zag because of the same reason.

For batch size 10000 and 1000000, theta movement is smooth because we update theta parameter over a reasonable set of examples (averaged change)

We can see from this figure too, that model with batch size 10000 and 1000000 will also converge to the original theta parameters given they are trained for several hours.

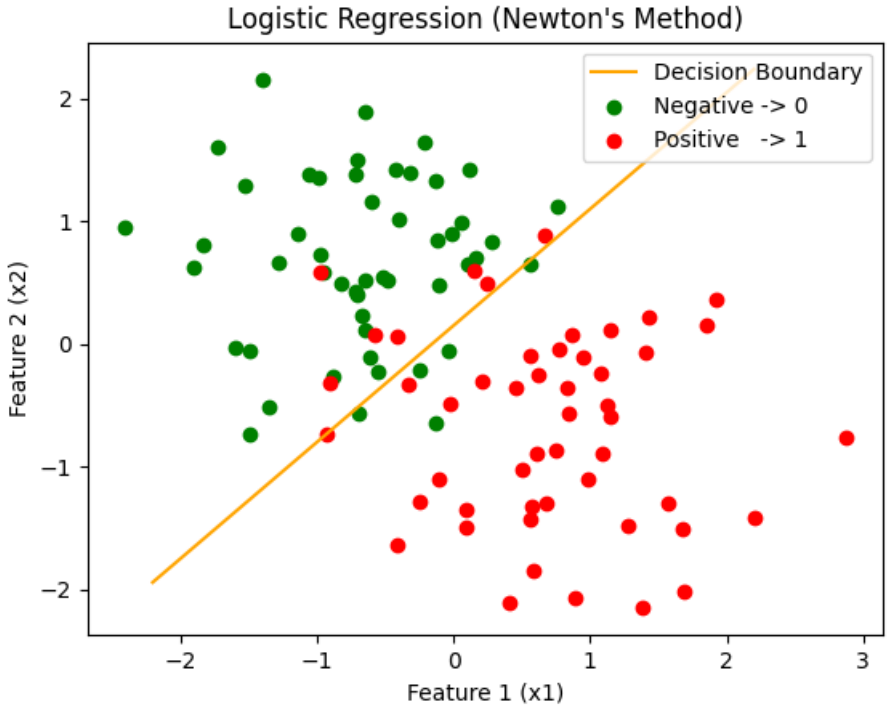
Part 3 (Logistic Regression)

In this question, we implement the Newton's method for computing the θ parameters. For updating the theta parameters, we compute gradient and Hessian matrix of L(θ). Newton's method is much faster than gradient descent. One caveat in using Newton's method is, when the derivate is zero or the second derivate of the funtion is not defined, then we can use the gradient descent approach but not Newton's method.

a) $\mathbf{x} \leftarrow \mathbf{x} - \left(\nabla^2 f(\mathbf{x})\right)^{-1} \nabla f(\mathbf{x})$ Paramter Updation (in terms of θ)

$\theta = [\ \theta_0\ \theta_1\ \theta_2\ ] = [0.4013, 2.5885, -2.7256]$

b) Training Data with the Decision Boundary



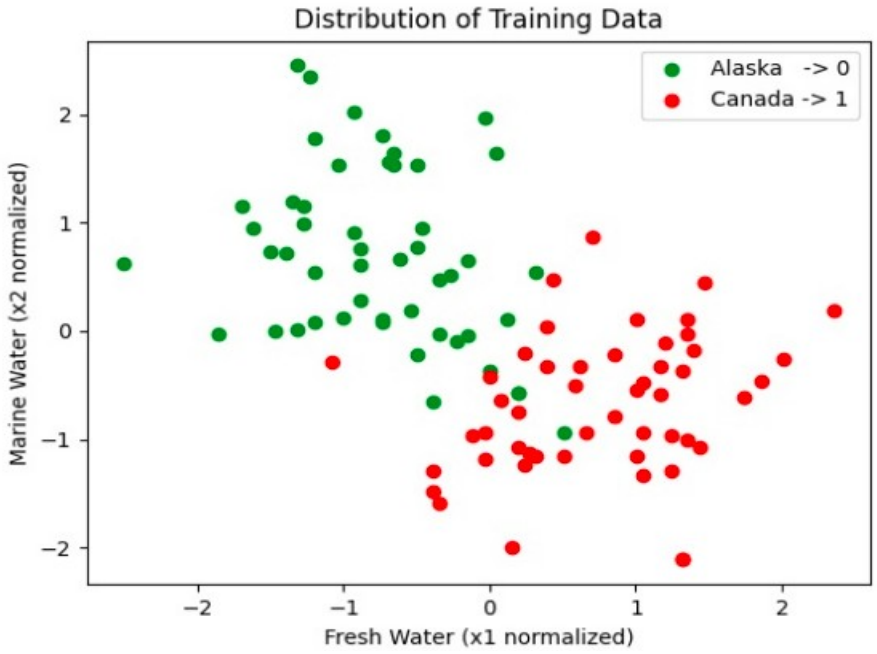
Part 4 (Gaussian Discriminant Analysis)

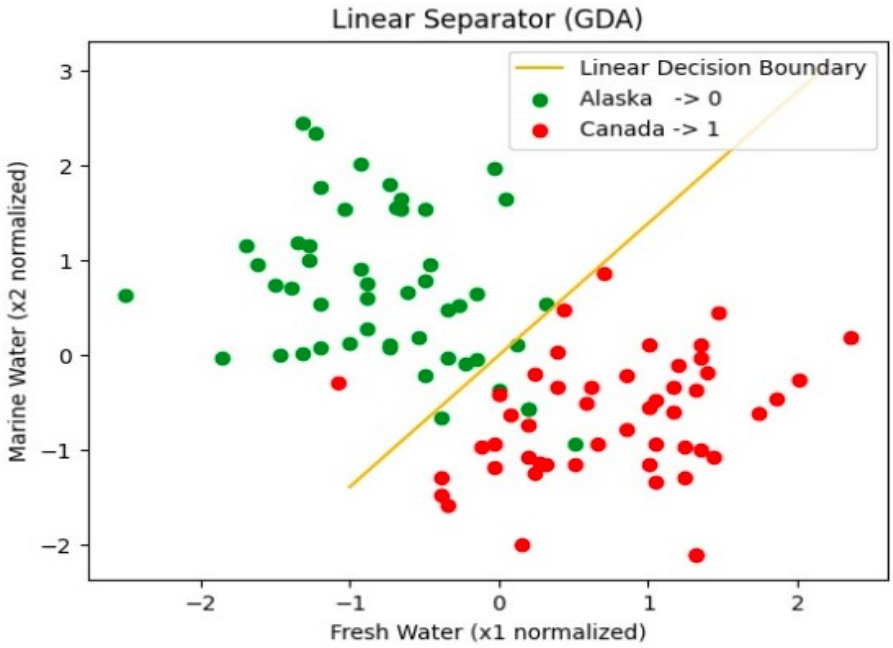
In this part, we implement Gaussian Discriminant Analysis using both the Linear Separator and the Quadratic Separator. We compute the mean and the covariance matrix for each class, we also compute the sigma matrix for linear separator. Using μ_0, μ_1, Σ, we plot the linear separator, and, using μ_0, μ_1, Σ_0, Σ_1, we plot the quadratic separator.

a) μ_0 = [-0.75529433, 0.68509431]
μ_1 = [0.75529433, -0.68509431]

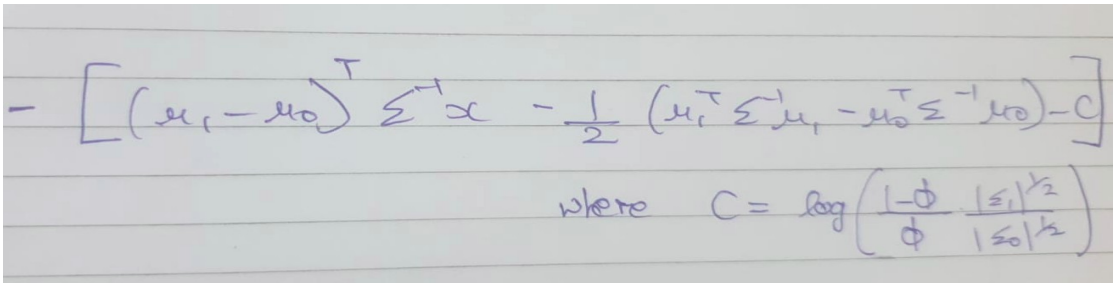
Σ = [[0.42953048, -0.02247228]
[-0.02247228, 0.53064579]]

b)

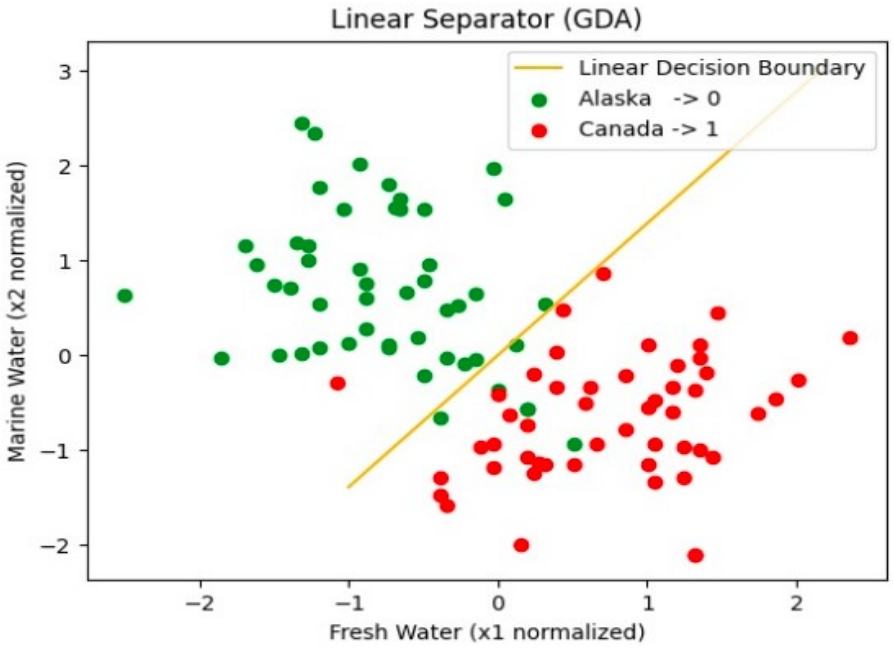




c) Equation:



This is the value of log(A). At decision boundary, log(A) = 0 (done in class)

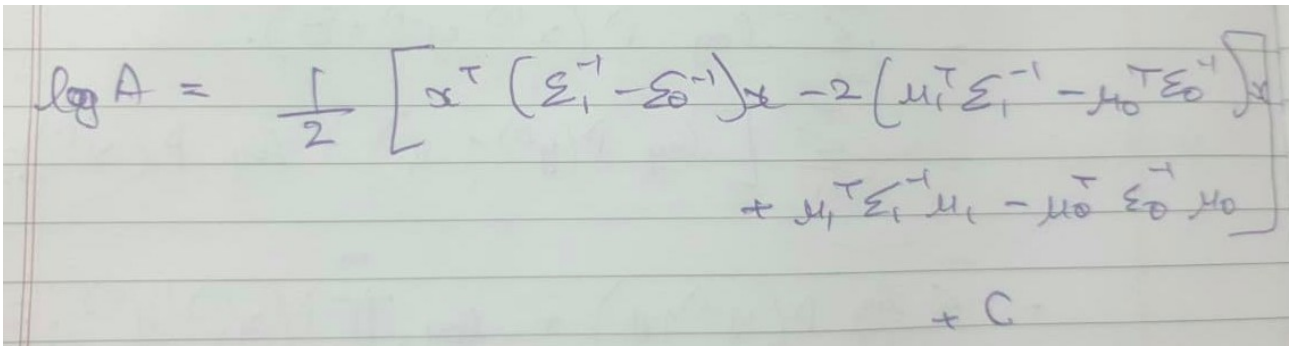


d) μ_0 = [-0.75529433, 0.68509431]
μ_1 = [0.75529433, -0.68509431]

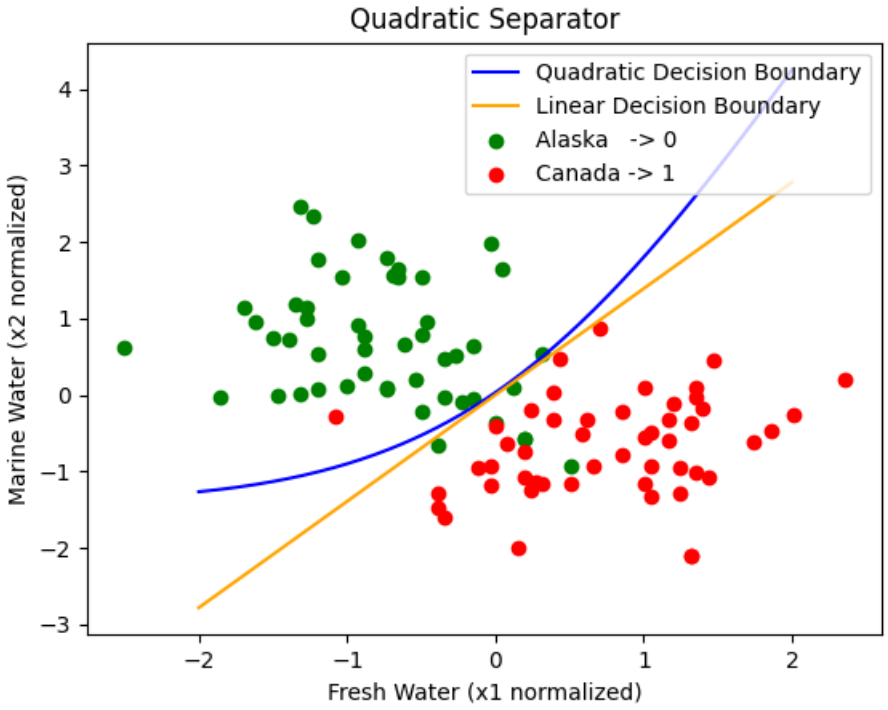
Σ_0 = [[0.38158978, -0.15486516]
[-0.15486516, 0.64773717]]

Σ_1 = [[0.47747117, 0.1099206]
[0.1099206, 0.41355441]]

e) Equation:



And for the decision boundary, log(A) = 0 (done in class)



f) On analyzing both the separators, we can say that the Quadratic Separator is slightly better than the Linear Separator.

This is because the Quadratic Separator is a more general separator i.e. it is more flexible and allows different classes to have different co-variance matrices whereas linear separator assumes variance to be consistent across all classes.

Hence, Quadratic Separator makes better predictions in general.