ASSIGNMENT 1

COL 774: Machine Learning

1 Linear Regression

For Linear Regression, we will perform batch gradient descent, trying to minimize the following loss function:

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

We optimize J(θ) by using the argmin operation, since the loss function is convex, and we'll get a minima. To perform linear regression using gradient descent, we have two hyperparameters:

(a) Learning Rate (eta)
(b) Epsilon (convergence parameter)

After a few experiments, I have decided the **learning rate to be 0.01**. I have carried out experimentation by splitting the training (80%) and test data (20%) and calculating error on some values:

Learning Rate	Error
0.005	0.021
0.01	0.018
0.02	0.020
0.04	0.022

Choosing a good learning rate is essential because higher learning rate will cause jumps and will diverge (instead of converging to the minima)

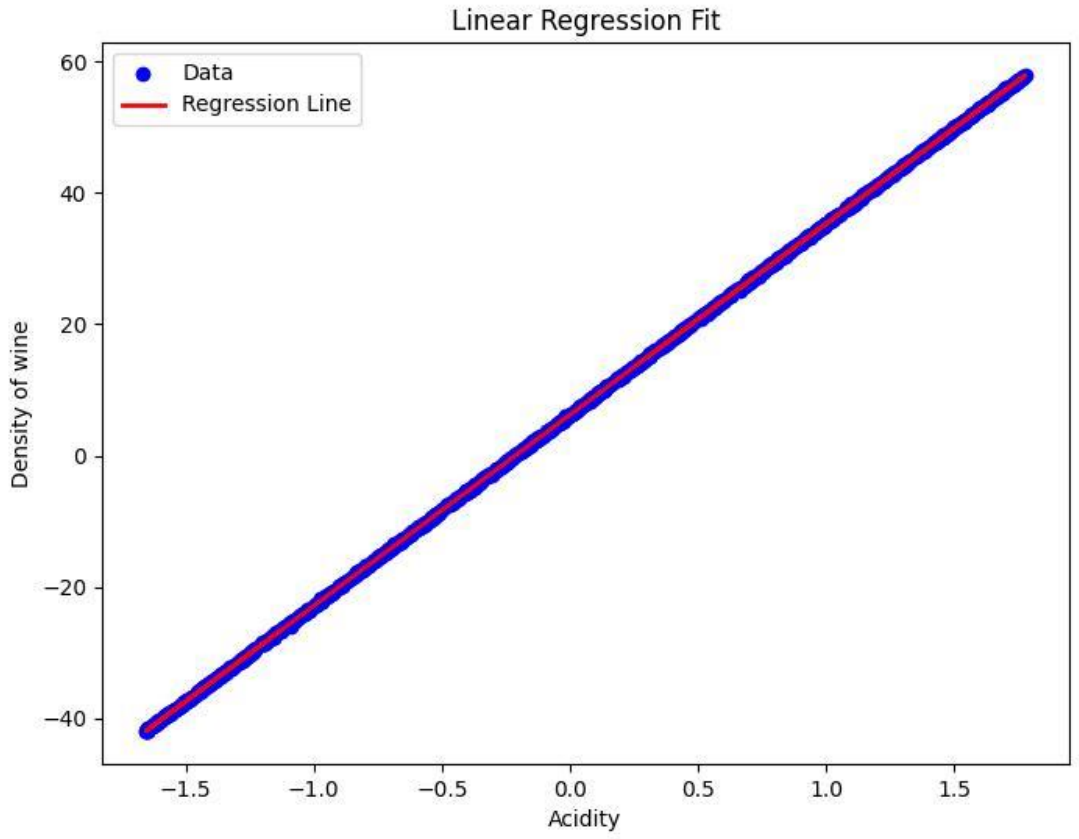
Similarly, for the convergence parameter, **Epsilon = 10^-6** (I have also set max iterations to be 100000, just in case). Choosing an epsilon was also done on the same basis as above. Choosing an epsilon is also necessary as we need to prevent both overfitting (i.e. make the curve more inclined towards the dataset) and underfitting (i.e. make the curve general).

Epsilon	Error (on validation dataset)

1e-4	0.026
1e-5	0.020
1e-6	0.018
1e-7	0.023

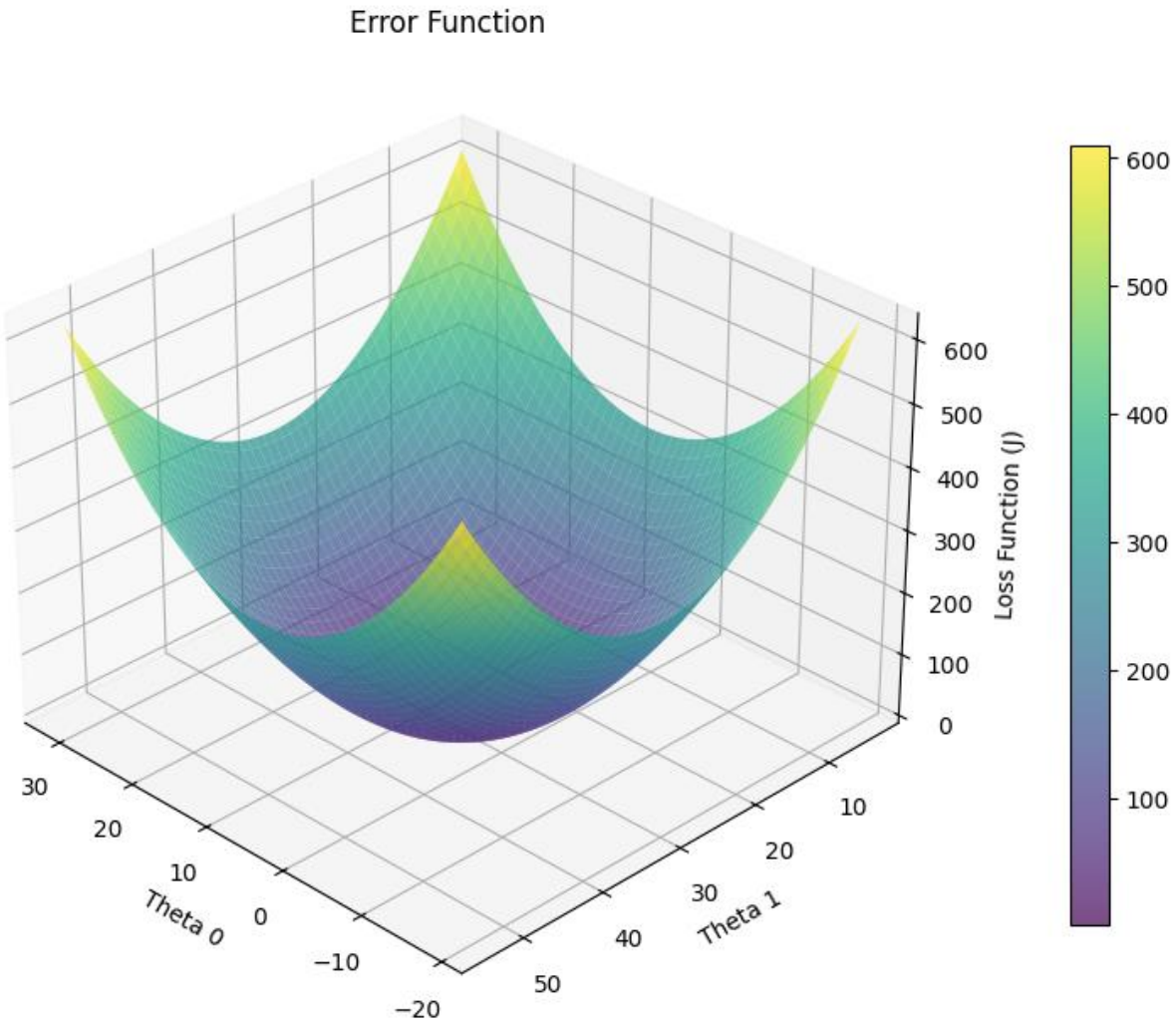
Hence,

Learning Rate = 0.01
Convergence condition: | (J(θ) @ t) - (J(θ) @ t-1) | <= Epsilon,
where Epsilon = 10^-6
Learned parameters: [6.22, 29.06]

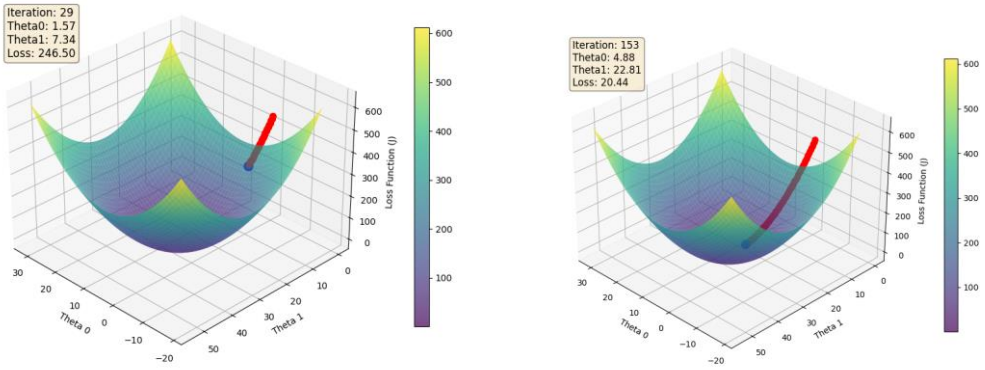


y = 6.22 + 29.06x is the learned equation

The error function is in the shape of a convex bowl
(as is expected from the equation given)

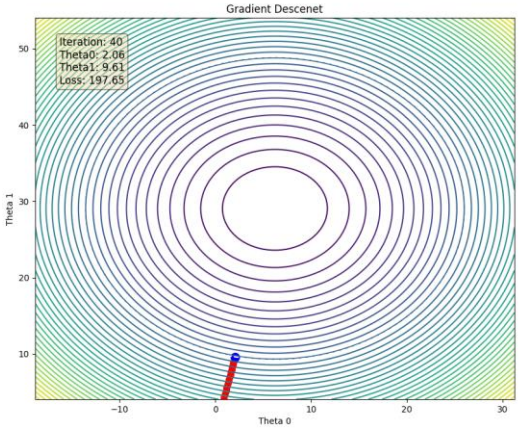


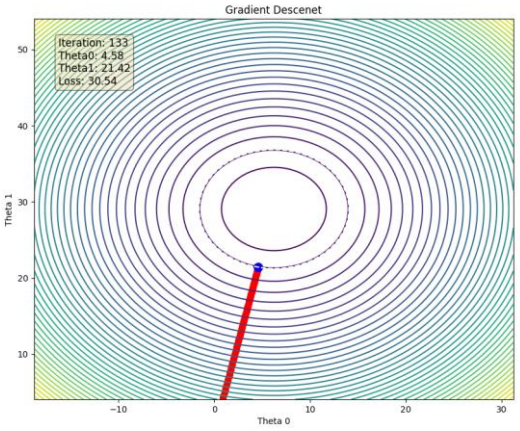
The gradient descent can be shown in the plots below: (iteration 29 and 152)



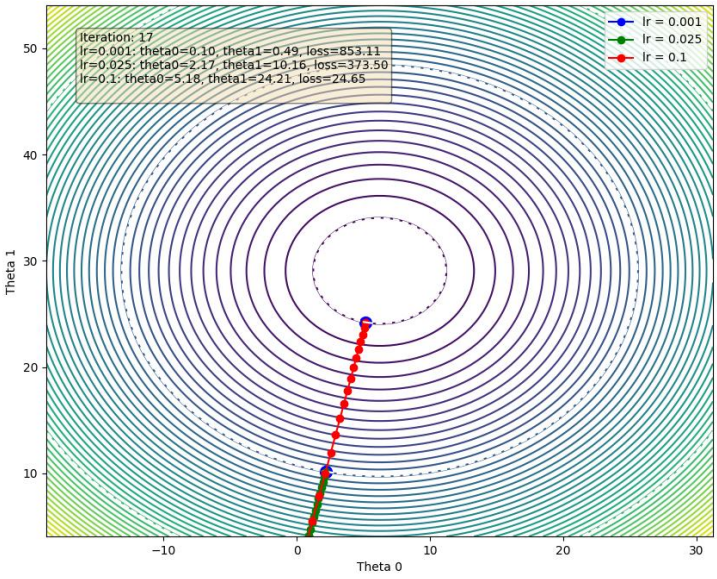
As you can see from the above graphs, using gradient descent we can reach the minima of the convex shape bowl by J(θ)

We can visualize the above 3D plot as a contour plot as well, where the inner circles represent lower values of J(θ) and the gradient descent line goes inwards, perpendicular to the contours. (iterations 40 and 133)

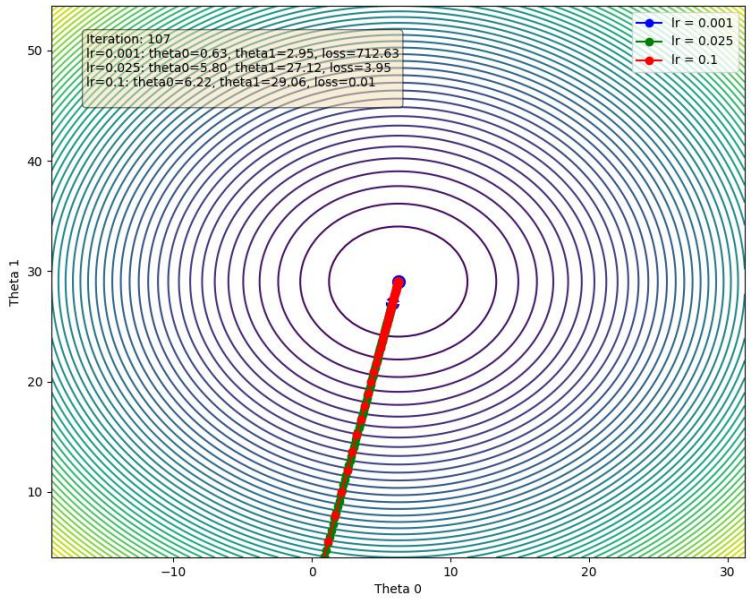




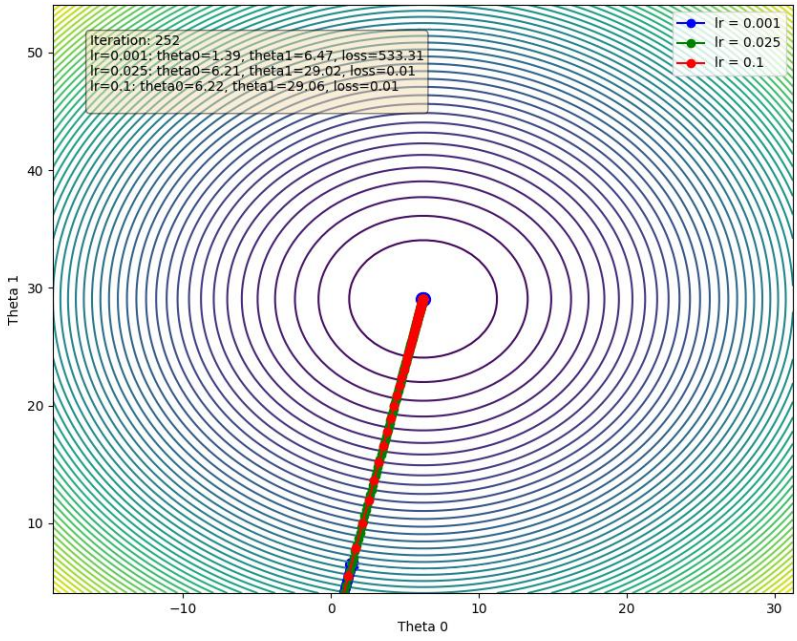
We can also finally see the contour plot, with three different learning rates:



In the first one, we can see learning rate = 0.1 running fast ahead, and then learning rate = 0.025, while 0.001 is not yet in picture



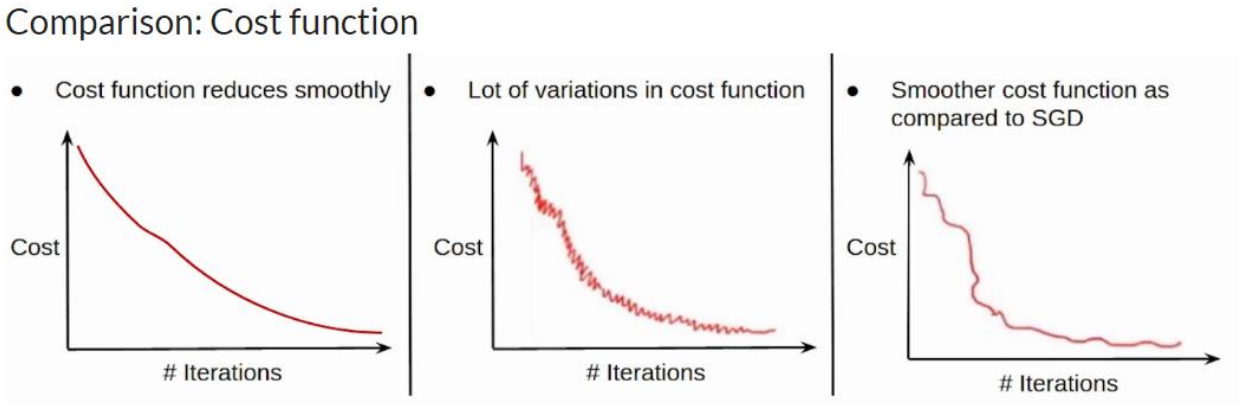
Learning rate = 0.1 and learning rate = 0.025 are in the inner circle, while learning rate = 0.001 has just entered the figure



Learning rate = 0.1 and learning rate = 0.025 are almost converged, while learning rate = 0.001 is still in outer circle

2 Linear Regression (with Stochastic Gradient Descent)

To improve upon the time complexity, stochastic gradient descent or mini-batch gradient descent considers mini-batches of data (consisting of k training examples), over which the gradient descent is applied. Since stochastic gradient descent only considers a small amount of data at a time, there may be a distribution in variation, and hence selecting the convergence criteria properly is critical.



To prevent such extreme variations, we would need to smoothen the curve in a way that a convergence criterion applies. There are two such ways to smoothen the curve discussed in class:

(1) Take k-moving average
(2) Take a moving average over k points, and use the difference

Method (1) seemed a bit strange here, since it would mean sampling over separate points at a gap of k. Instead, it seemed more intuitive to use (2), which would sample a 'representative' sample over k samples to compute convergence.
(as can be seen in the figure above)

We now introduce the following hyperparameters:
eps : for convergence of the k-sampled average (k batches)
k : how many batches of data to be sampled over
max_iter : again, kept to be 10000
learning rate is fixed at 0.001

$$J_b(\theta) = \frac{1}{2k} \sum_{k=1}^{r} \left(y^{(i_k)} - h_\theta(x^{(i_k)})\right)^2$$

Convergence equation:

| avg over k-batches (i = m to m+k) (J) - avg over k-batches (i = m-k to m) | <= eps

Results (model-fitting):

Theta for batch size 1: [3.01203443 0.97448168 2.00946894]
Theta for batch size 80: [2.79672854 1.01820435 2.02968263]
Theta for batch size 8000: [2.89208077 1.00986746 2.01517952]
Theta for batch size 800000: [1.87888 1.09706777 2.15979476]

Theta computed using closed form: [2.99973428 1.00019572 1.99995455]

Results (Error on test data and train data):

Test Error for batch size 1: 0.2508087184745415
Test Error for batch size 80: 0.2510855897657011
Test Error for batch size 8000: 0.2501963601838278 5
Test Error for batch size 800000: 0.2859936624625987

Train Error for batch size 1: 0.25096639961605344
Train Error for batch size 80: 0.2510967452347833
Train Error for batch size 8000: 0.25022220566069664
Train Error for batch size 800000: 0.2857947750080463

Results (time):

Time taken for batch size 1: 4.4374098777771
Time taken for batch size 80: 0.6512830257415771
Time taken for batch size 8000: 3.5791282653808594
Time taken for batch size 800000: 218.04223370552063

Batch size	Number of iterations
1	1
80	1000
8000	880
800000	4150

On the number of iterations, time and batch size:

SGD with 1 took only 1 iteration, but the results overshooted, this is because of a high variance in SGD in this case (it took a long time due to convergence checks)

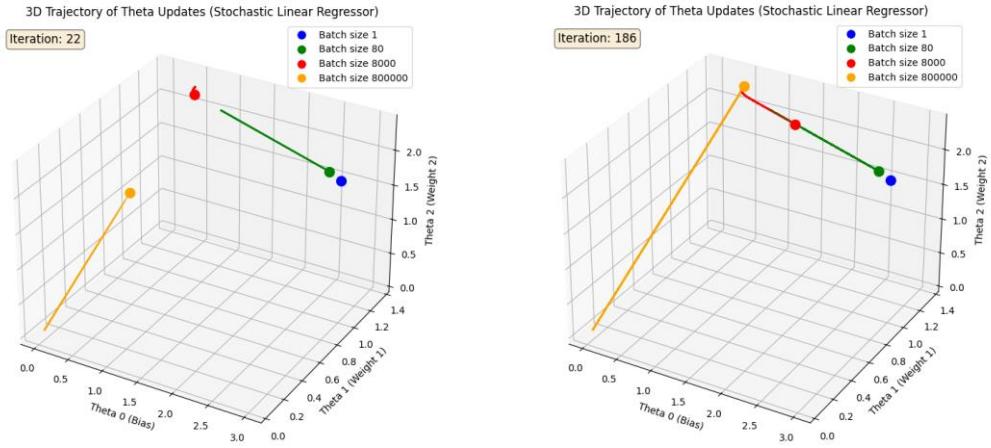
SGD with batch = dataset gave bad results, possible due to the unstochastic nature of the descent, and took a long time to converge (which pretty much shows why stochastic gradient descent is much better than batch gradient descent)

The remaining two cases were much better

On the estimation:

The estimation with batch descent was bad, whereas it was pretty close for the others. There was scope for trying more, but with CPU power, this is what was achievable (even after vectorization and other efficiency boosters)

Movement of parameters:



Due to this being a 3D graph (perspective) and a small scale, the variations are not much visible, however you can see that **lower batch size has higher aberrations**

3 Logistic Regression (using Newton's method)

Theta_learned = [0.04703699 2.68344818 -2.21461449]
(in the order, theta0, theta1 and theta2)

To ensure that the Hessian matrix is +ve semi definite, we have added a regularization of (a)*I, where a = 10^-12

Also, note that the loss function has an almost quadratic shape (though it is a concave curve i.e. an upside bowl), Newton's method is helpful here for optimization over nearly second-degree curves (near the maxima)

For **maximizing** a function $f(x)$, Newton's method iterates using the update rule:

$$x^{(k+1)} = x^{(k)} - \left[\nabla^2 f(x^{(k)})\right]^{-1} \nabla f(x^{(k)})$$

where:

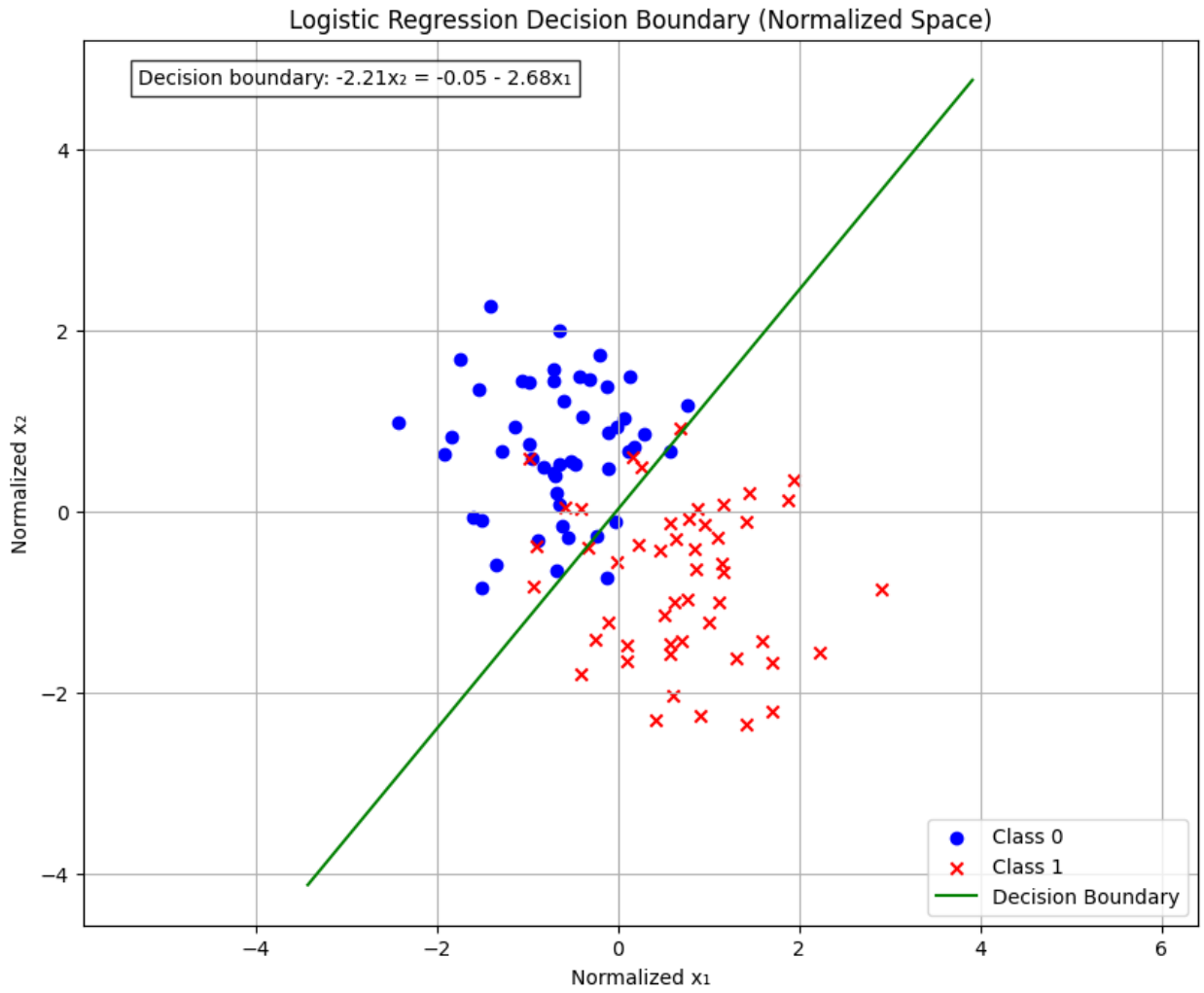
- $\nabla f(x)$ is the gradient (first derivative) of $f(x)$,
- $\nabla^2 f(x)$ is the Hessian (second derivative) of $f(x)$.

Convergence Condition:

Newton's method converges **quadratically** near a maximum if:

1. The Hessian $\nabla^2 f(x)$ is **negative definite** at the maximum.
2. The initial point $x^{(0)}$ is sufficiently close to the maximum.

Newton's method is particularly as it decreases the number of iterations almost by an exponential factor (i.e. if gradient descent took around 10000, Newton's method would get enough close by 5-10 steps)



4 Gaussian Discriminative Analysis (GDA)

In my analysis, I have mapped Alaska to 0 and Canada to 1

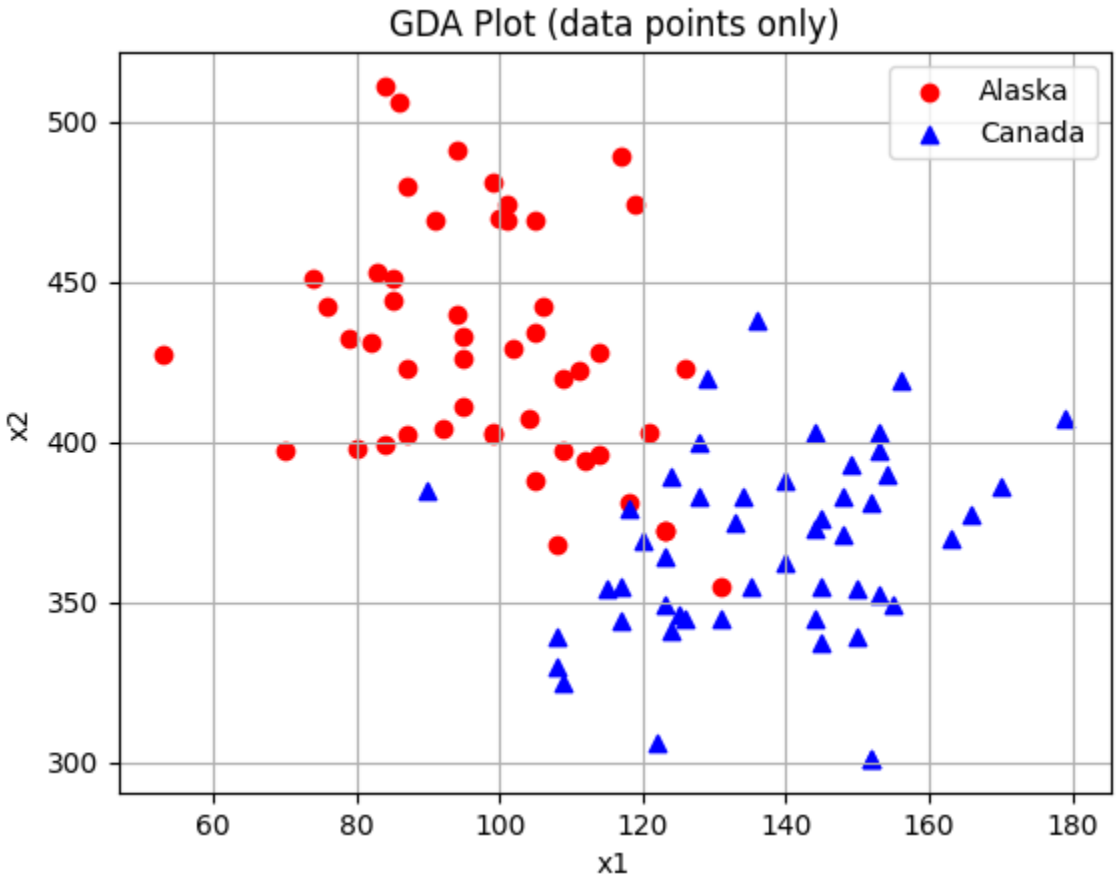
For the shared covariance model:
mu_0: [-0.75529433 0.68509431]
mu_1: [0.75529433 -0.68509431]
sigma: [[0.42953048 -0.02247228] [-0.02247228 0.53064579]]

For the different covariance model:

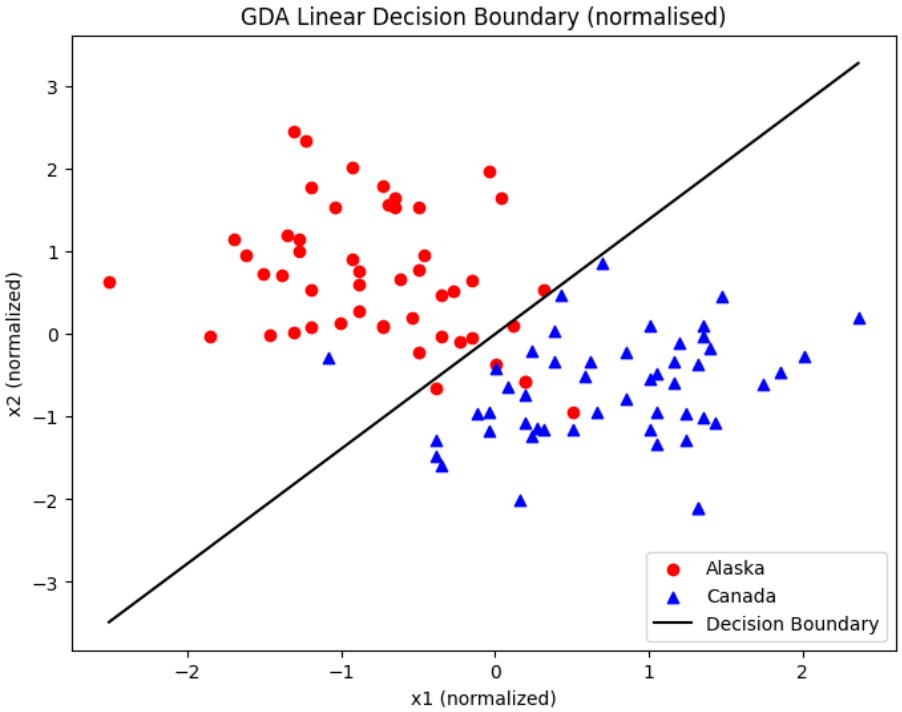
mu_0: [-0.75529433 0.68509431]
mu_1: [0.75529433 -0.68509431]

sigma_0: [[0.38158978 -0.15486516] [-0.15486516 0.64773717]]
sigma_1: [[0.47747117 0.1099206] [0.1099206 0.41355441]]

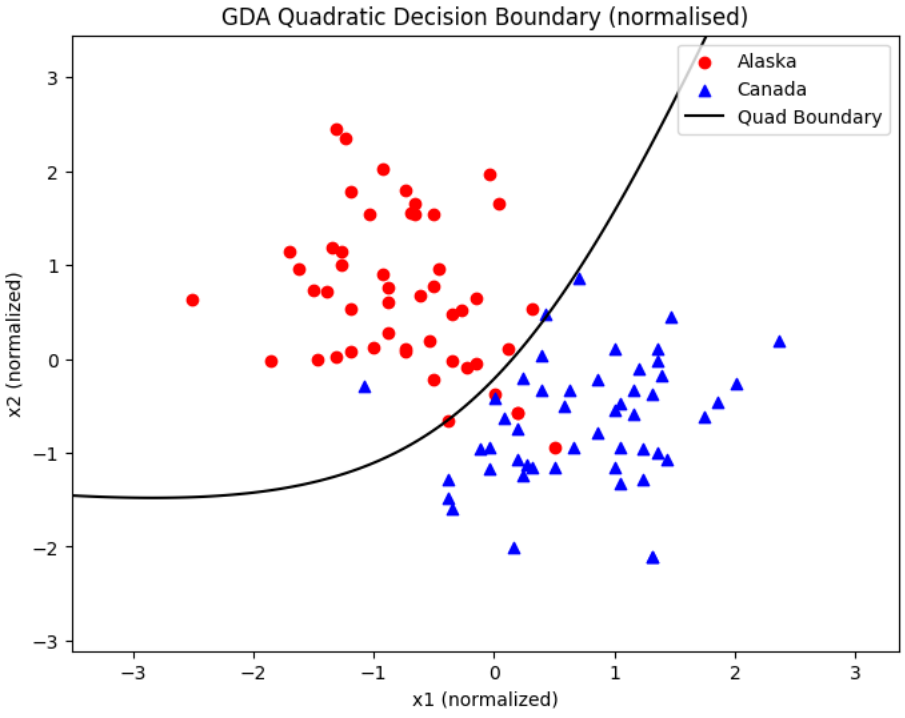
Data Plot:



Linear Boundary Plot:



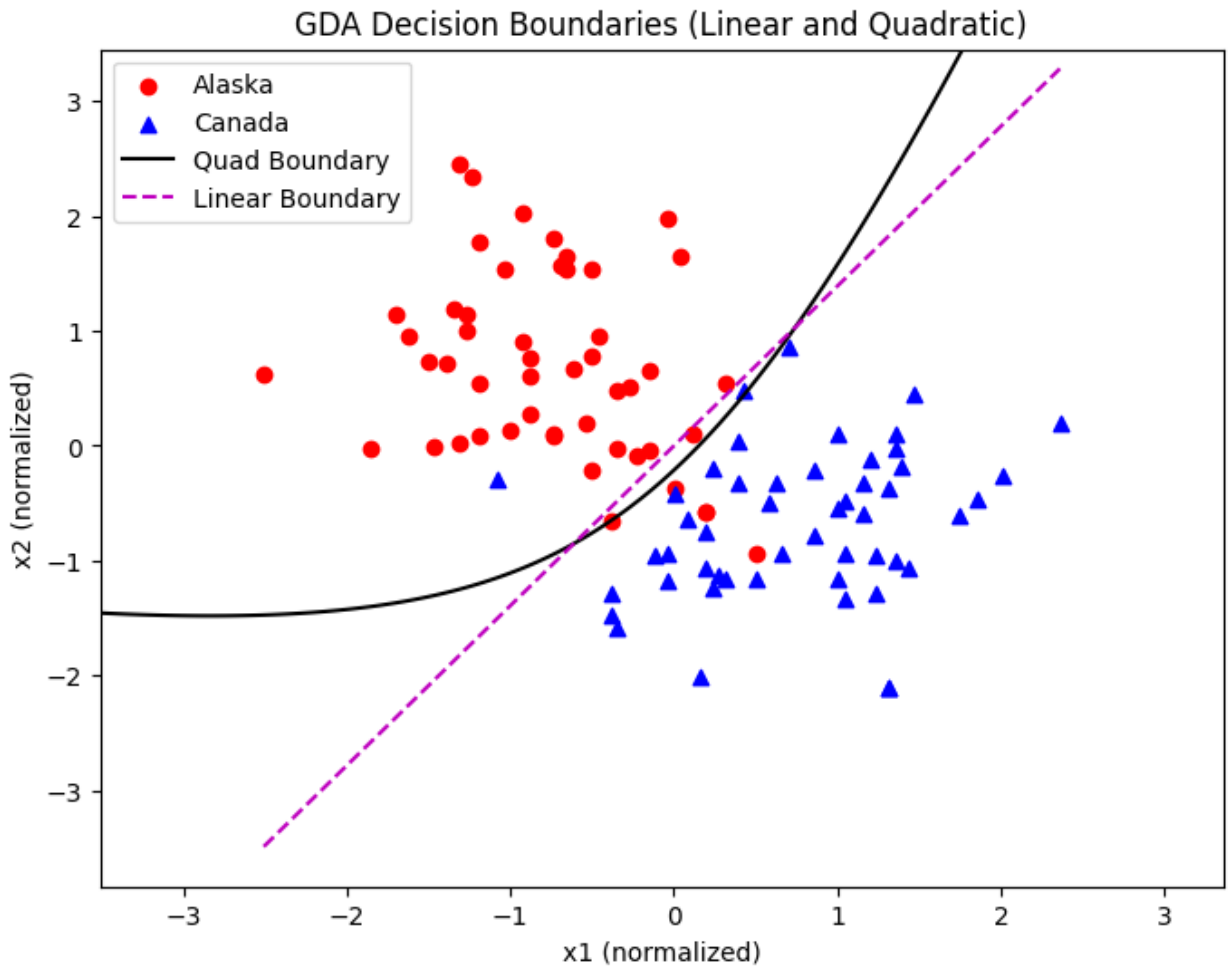
Quadratic Boundary Plot:



\

Linear Decision Boundary Equation (for normalized features):
$3.389x_1 + -2.439x_2 + 0.000 = 0$

Quadratic Decision Boundary Equation (for normalized features):
$0.336\ x_1^2 + -0.433\ x_2^2 + 1.287\ x_1x_2 + 3.808\ x_1 + -2.860\ x_2 + -0.585 = 0$



LDA is efficient for high-dimensional, linearly separable problems, offering simplicity and robustness by assuming equal class covariances.
QDA provides greater flexibility by modeling different covariances, making it suitable for complex class distributions but requiring more data to avoid overfitting.
Choosing between LDA and QDA depends on the bias-variance tradeoff, where LDA reduces variance for stability, while QDA lowers bias for better class separation in non-linear cases