Fundamentals of Machine Learning

COL 7074

Assignment 1

Name: Trishanku Sarma
Entry ID: 2025AIY7584

September 7, 2025

Contents

1 Linear Regression

In this problem, we implement least squares linear regression to predict the density of wine based on its acidity. The error metric for least squares is defined as:

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

where

$$h_\theta(x) = \theta^T x,$$

1.1 Batch Gradient Descent

We implemented batch gradient descent with initialization $\theta = \vec{0}$. **The stopping criteria used was:**

$$\text{Stop if } (\text{epoch} + 1 \geq \text{max\_epochs}) \quad \text{or} \quad (\text{epoch} > 1 \ \wedge \ (J(\theta)^{prev} - J(\theta)^{curr}) < \epsilon),$$

where ϵ is the convergence threshold.

The input $X \in R^{D*N}$ where D = number of features and N = number of input data points and $Y \in R^{1*N}$

The gradient used in batch gradient descent is given by:

$$\nabla_\theta J(\theta) = \frac{1}{m} \mathbf{X} \left(\theta^T \mathbf{X} - \mathbf{Y} \right)^T,$$

where the transpose ensures correct matrix dimensions for the update. And the update rule is:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta),$$

The parameters used were:

- Maximum epochs: 100,000
- Convergence threshold: 1×10^{-6}
- Learning rate: 0.01

Final set of parameters learned by the algorithm: With these parameters, my algorithm converges to global minima after Epoch 798 with Loss: 0.00492338432 and $\theta_0 : 6.21665396, \theta_1 : 29.05521858$

The following figure shows the descent of loss v/s epochs

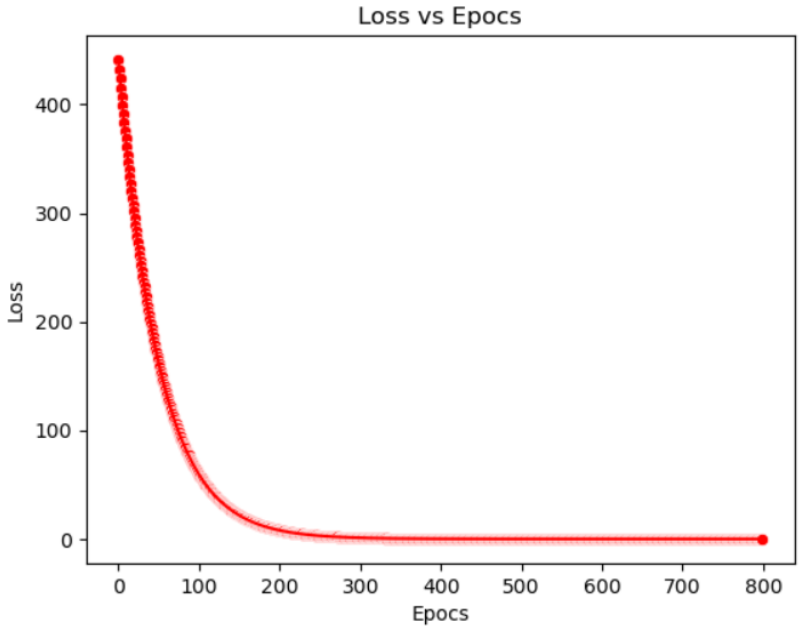


Figure 1: Descent of loss v/s epochs.

1.2 Plotting Hypothesis in 2-D

The following figure shows the inputs against the ground truth levels.

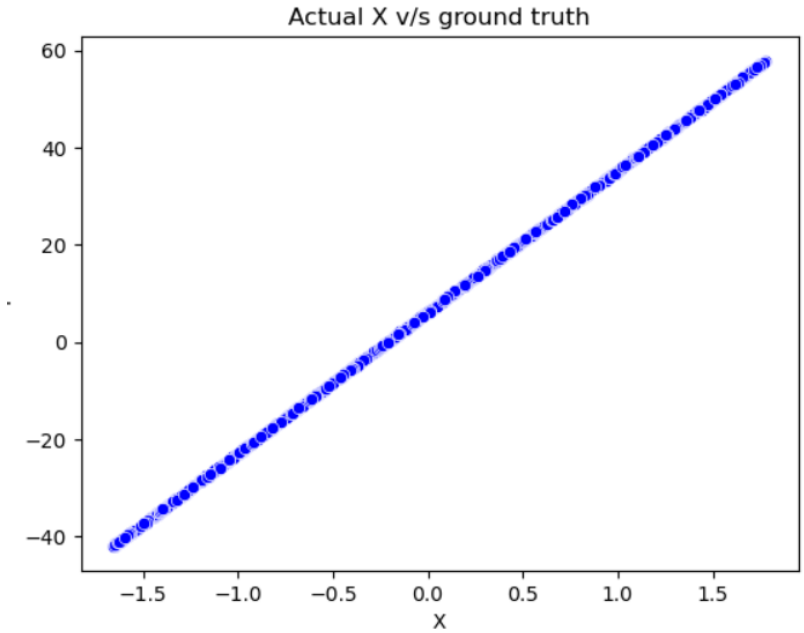


Figure 2: Training data with actual ground truth.

The following figure shows the training data points along with the hypothesis line learned by batch gradient descent.

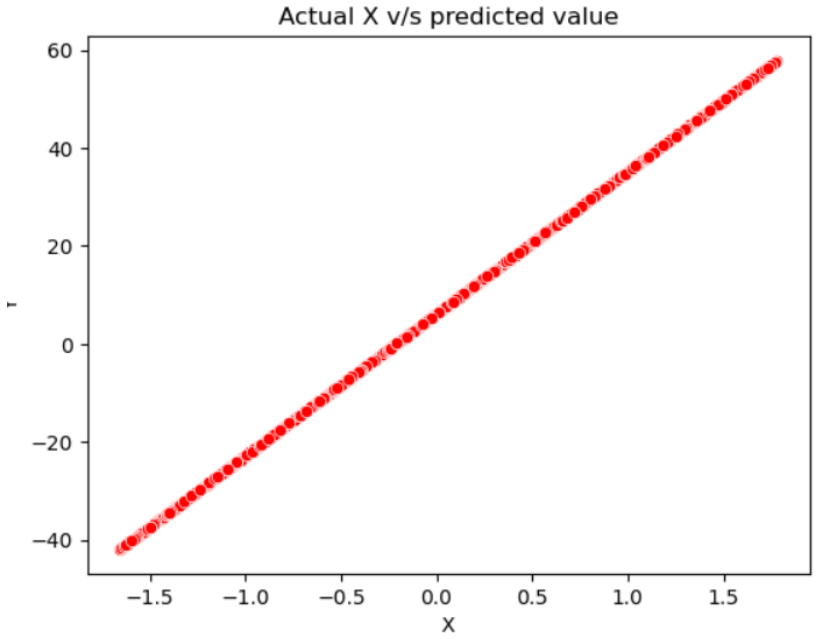


Figure 3: Training data learned by hypothesis function.

1.3 3-D Mesh Grid of Error Function

To analyze the gradient descent trajectory on the cost surface $J(\theta_0, \theta_1)$:

1. Generated a mesh grid of θ_0 and θ_1 values by taking the optimal solution and adding/subtracting a margin of 30 in each direction.

2. Selected an extreme point and took gradient descent on this grid and updated the parameters iteratively, observing how the algorithm moves the point towards the optimum.

3. To visualize the movement clearly, a small delay of 0.02 seconds was added between iterations so that the descent path can be perceived on the 3D surface.

Notes: This visualization helps in understanding how gradient descent navigates the cost surface and converges towards the global minimum.

The error function $J(\theta)$ is visualized as a 3D surface plot with θ_0 and θ_1 on the axes.

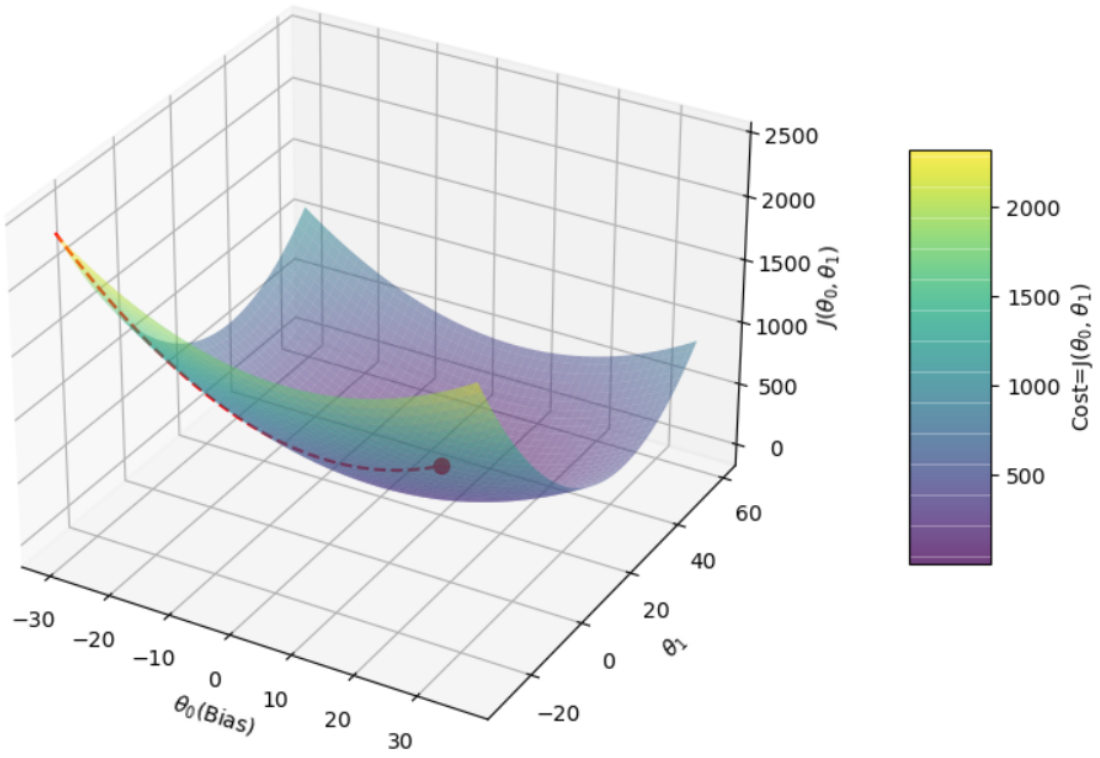


Figure 4: 3D mesh of error function $J(\theta)$ with gradient descent trajectory.

1.4 Contour Plots of Error Function

Similarly, we generated contours of the error function with gradient descent parameter updates at each iteration for the same learning rate.

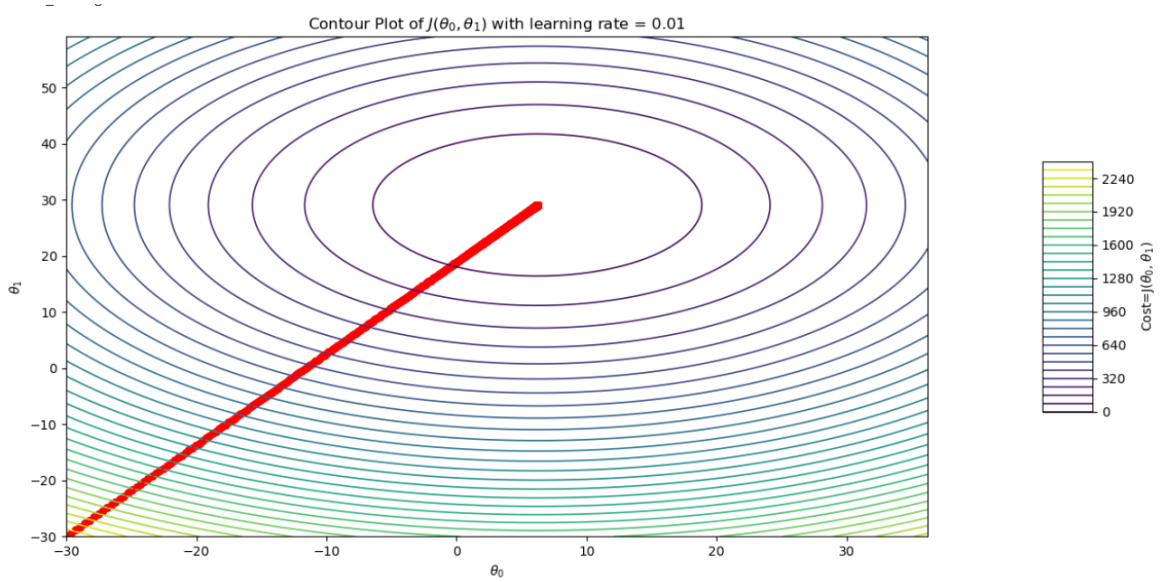
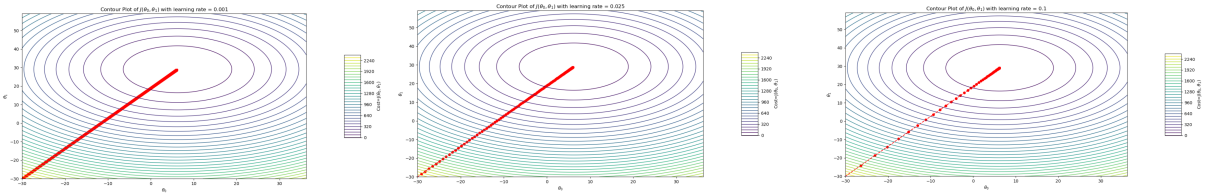


Figure 5: 2D contour plot of error function with gradient descent steps.

1.5 Effect of Learning Rates

Finally, We compare gradient descent with different learning rates $\eta \in \{0.001, 0.025, 0.1\}$.



(a) $\eta = 0.001$ (b) $\eta = 0.025$ (c) $\eta = 0.1$

Figure 6: Contour plots of error function for different learning rates.

Under the same convergence criteria, the results for different learning rates are summarized below:

Learning Rate	**Epoch of Convergence**	**Final Loss $J(\theta)$**	θ_0	θ_1
0.001	5390	0.0548140	6.05405	28.79484
0.025	277	0.0067413	6.18689	29.01255
0.1	73	0.0052851	6.20378	29.04027

Table 1: Results of batch gradient descent for different learning rates

Observation: `As it can be seen the algorithm took a lot of time to converge under low learning rate still not close to convergence which implies that a large step helps Gradient descent to converge faster`

2 Sampling, Closed Form, and Stochastic Gradient Descent

2.1 Sampling Data from Normal Distribution and Adding Noise

Given the true parameters:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix},$$

the noiseless hypothesis is:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 3 + x_1 + 2x_2$$

where the input features are sampled as:

$$x_1 \sim \mathcal{N}(\mu_1 = 3, \sigma_1^2 = 4), \quad x_2 \sim \mathcal{N}(\mu_2 = -1, \sigma_2^2 = 4)$$

Adding Gaussian noise with variance $\sigma^2 = 2$, the observed output becomes:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \epsilon = 3 + x_1 + 2x_2 + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 2)$$

```
# Intercept term (x0 = 1)
X0 = np.ones(total_data_points)

X1 = np.random.normal(
    loc = self.gaussianDetails["x1"]["mean"],
    scale = np.sqrt( self.gaussianDetails["x1"]["variance"] ),
    size = self.total_data_points
    ) # N(3, 4)

X2 = np.random.normal(
    loc = self.gaussianDetails["x2"]["mean"],
    scale = np.sqrt( self.gaussianDetails["x2"]["variance"] ),
    size = self.total_data_points
    ) # N(-1, 4)

# Stack features into design matrix X
self.X = np.vstack((X0, X1, X2))
print("Shape of X :",self.X.shape)

Y_Noise = np.random.normal(
    loc = self.gaussianDetails["y_noice"]["mean"],
    scale = np.sqrt( self.gaussianDetails["y_noice"]["variance"] ),
    size = self.total_data_points
    ) # N(0, 2)

theta = np.array([self.params["theta0"], self.params["theta1"], self.params["theta2"]])

# Y = theta0 + theta1*x1 + theta2*x2 + N(0,2)
self.Y = (np.dot(self.X.T, theta) + Y_Noise).reshape((1, total_data_points))
print("Shape of Y :",self.Y.shape)
```

Finally, the dataset of 1,000,000 points is split as:

$$\text{Training set: } 80\% = 800{,}000, \quad \text{Test set: } 20\% = 200{,}000$$

2.2 Implementing Stochastic Gradient Descent

To learn the original hypothesis from this data using SGD:

- Shuffle the examples initially and divide them into batches of size r. The b-th batch contains:
$$\{x^{(i_1)}, x^{(i_2)}, \ldots, x^{(i_r)}\}, \quad i_k = (b-1)r + k$$
- Perform gradient updates in a round-robin fashion over all batches for each epoch.
- The loss function is evaluated after each epoch with the newly learned parameters and the convergence criteria is checked

Convergence Criteria is determined by Moving-Average Loss

To determine convergence in Stochastic Gradient Descent, a moving-average loss was used instead of relying on the instantaneous loss value. At each epoch:

1. Compute the average of the previous *window size* loss values.
2. Compare this moving-average loss with the current epoch loss.
3. If the difference is less than a predefined threshold, the algorithm is considered to have converged:

$$\left| L_{\text{current}} - \frac{1}{w} \sum_{i=1}^{w} L_{\text{prev},i} \right| < \epsilon$$

where w is the window size and ϵ is the convergence threshold.

Rationale:

- Small batch sizes produce noisy updates, causing loss to fluctuate around the minimum. Using a larger window smooths out the noise and avoids premature convergence detection.
- Medium batch sizes require a moderate window, while large batch sizes have stable updates and can use a window size of 1.

The parameters used were:

- Maximum epochs: 100000
- Convergence threshold: 1×10^{-4}
- Learning rate: 0.001

Batch Size	**Window Size**	**Epochs Iterated**	**Final Loss** $J(\theta)$	θ_0	θ_1	θ_2
1	5	54	1.000588	3.0074	1.0148	1.9973
80	3	9	0.998605	3.0006	1.0019	2.0024
8000	1	118	1.000296	2.8912	1.0236	1.9922
800000	1	3300	1.182602	1.8664	1.2474	1.9175

Table 2: SGD results for different batch sizes, including moving-average smoothing window, convergence threshold, epochs to converge, final loss, and final parameters.

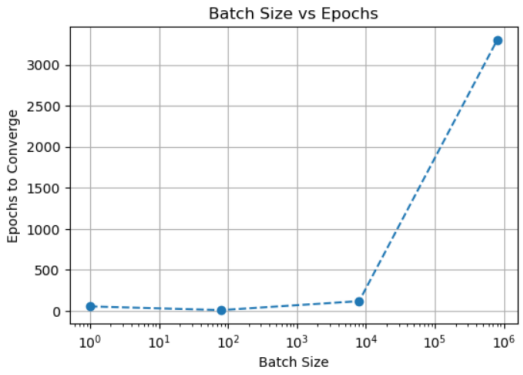
2.3 Implementing the Closed Form Solution and Comparison

2.3.1 Analysis on Varying Batch Sizes

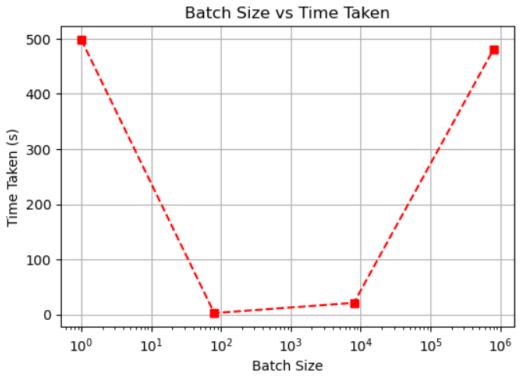
In the above section, the behavior of gradient descent strongly depends on the chosen batch size. Observations from the experiments are summarized below:

- **Batch size = 1 (Stochastic GD):** Converged in 76 epochs, but the wall-clock time was very high (677s) since each epoch computes gradients for all m samples individually. The final parameters θ were close to the true values, and the loss ~ 1.0. Fast in terms of epochs, but inefficient in real time.

- **Batch size = 80 (Mini-batch GD):** Converged in only 7 epochs and 2.38s total. This provides the best trade-off between convergence speed and computational efficiency. The final θ values were extremely close to the true parameters, showing stable convergence.
- **Batch size = 8000 (Large batch GD):** Required 118 epochs and 20.9s total. Converged reasonably well, but slower in epochs compared to batch size 80. Faster than full SGD, but less efficient than mini-batch GD.
- **Batch size = 800,000 (Full batch GD):** Needed 3296 epochs and 446s total. The final parameters deviated significantly from the true values, with a loss > 1.18. Full batch GD moves too cautiously and struggles to escape plateaus, leading to poorer convergence.

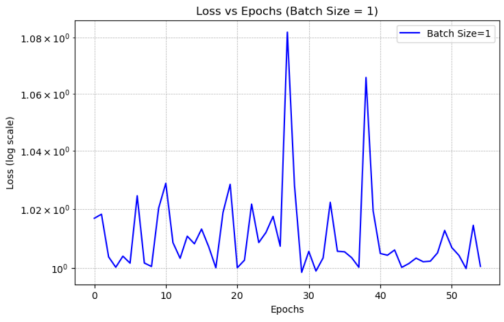


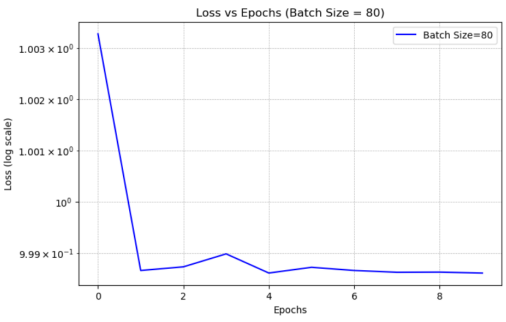
(a) Batch sizes vs Epochs to Converge



(b) Batch sizes vs Time to Converge

Figure 7: Comparison of different batch sizes on convergence metrics





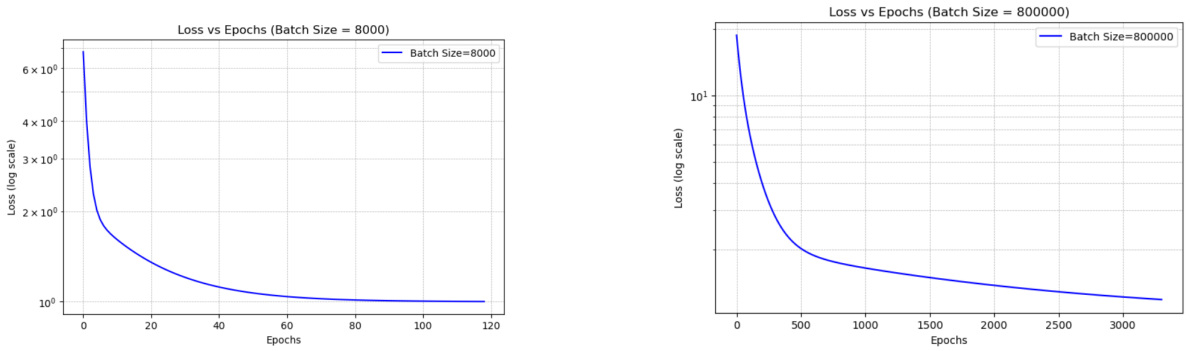


Figure 9: Comparison of different batch sizes on epochs iterated v/s loss at that epoch

Conclusion: These observations demonstrate why in practice mini-batches (e.g., sizes 32, 64, 128) are commonly used — they strike a balance between stable convergence and computational efficiency.

2.3.2 Evaluating parameters obtained from closed form solution

Here, the closed form is given by

$$\theta = (X^T X)^{-1} X^T Y$$

On evaluating it on the train data :

```
XTX = np.dot( train_x.T, train_x)
XTX_INV = np.linalg.inv(XTX)
XTX_INV_XT = np.dot(XTX_INV, train_x.T)
theta_pred_closed_form = np.dot(XTX_INV_XT, train_y)

print("Theta in closed form : ", theta_pred_closed_form)
```

```
train_x shape (3, 800000)
train_y shape (1, 800000)
After taking transpose
train_x shape (800000, 3)
train_y shape (800000, 1)
Theta in closed form :  [[3.00089878]
 [0.9994806 ]
 [2.00023715]]
```

We found the parameters are $\theta_0 = 3.00089878, \theta_1 = 0.9994806, \theta_2 = 2.00023715$. They are too close to the true parameters but not exactly same. Reasonably because we evaluated with 80% shuffled data with added noise to output but far better than that we learned by SGD

Method / Batch Size	θ_0	θ_1	θ_2	Epochs	Final Loss	Time Taken (s)
True Parameters	3.0000	1.0000	2.0000	–	–	–
Closed Form	3.0009	0.9995	2.0002	–	–	–
SGD (Batch = 1)	3.0074	1.0148	1.9973	54	1.0006	497.94
Mini-batch (80)	3.0006	1.0019	2.0024	9	0.9986	2.74
Mini-batch (8000)	2.8912	1.0236	1.9922	118	1.0003	21.24
Full Batch (800000)	1.8664	1.2474	1.9175	3300	1.1826	480.88

Table 3: Comparison of different methods and batch sizes for convergence performance

2.4 Evaluating and Comparing the Mean Squared Error on Train and Test Sets

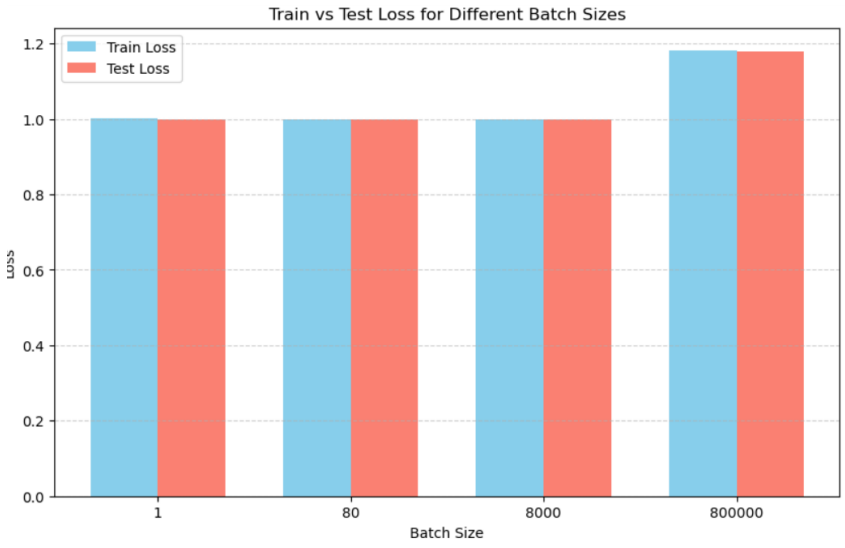
```
for idx, batchSize in enumerate(batchSizes):
    # y_pred = (1/test_size)*(<theta^T, train_x> - train_y)
    theta = params[batchSize]['finalTheta']

    y_pred_train = np.dot(theta.T, x_train)
    loss_train = np.sum(np.square(y_pred_train - y_train))/(2*model.train_size)

    y_pred_test = np.dot(theta.T, x_test)
    loss_test = np.sum(np.square(y_pred_test - y_test))/(2*test_data_size)
```

Batch Size	Train Loss	Test Loss
1	1.000588	1.000310
80	0.998605	0.998370
8000	1.000296	0.999856
800000	1.182602	1.180433

Table 4: Train vs Test Loss for different batch sizes



2.5 Plotting the Movement of Parameters in 3-D Plane for Different Batch Sizes and Analysis

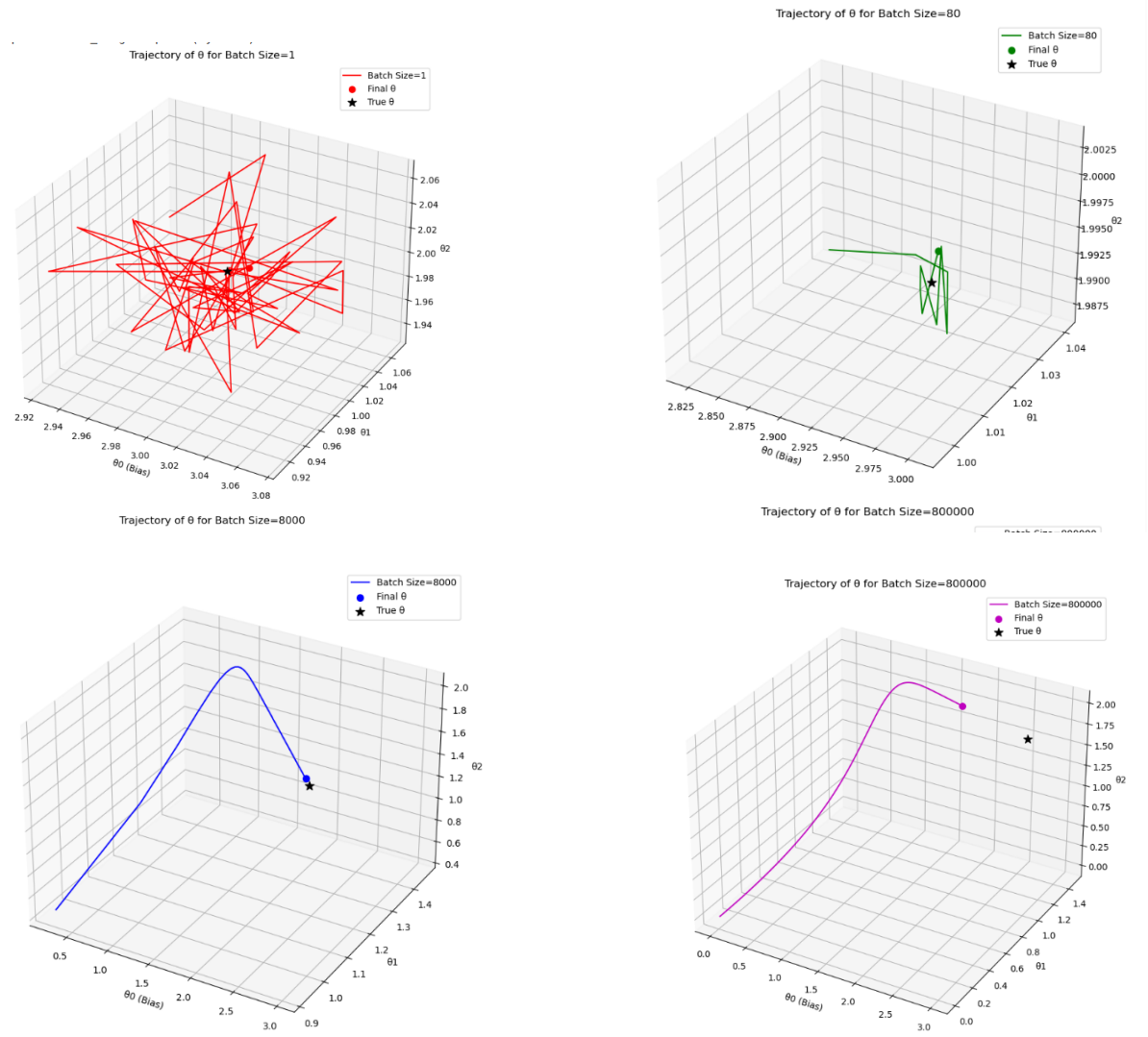


Figure 10: Comparison of parameters movement in 3-D on batch sizes

Observations and Intuitions that we can deduce from these:

- **Batch size 1 (SGD):** Very jagged trajectory, lots of bouncing, highly stochastic.
- **Batch size 80 (Mini-batch):** Smoother path, but still with some oscillations.
- **Batch size 8000:** Almost a straight line toward the optimum.
- **Batch size 800000 (Full-batch):** Very smooth and direct path, but takes many epochs to move.

3 Logistic Regression

3.1 Implementing Newton's Method for Optimizing Log Likelihood $L(\theta)$

The log-likelihood function for logistic regression is given by:

$$L(\theta) = \sum_{i=1}^{m} \Big[y^{(i)} \log \big(h_\theta(x^{(i)})\big) + \big(1 - y^{(i)}\big) \log \big(1 - h_\theta(x^{(i)})\big) \Big],$$

where

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}.$$

To optimize this function, we apply Newton's Method. And we are optimizing the negative log likelihood and will try to find its optimal minimum. The gradient and Hessian of negative log likelihood are:

$$\nabla L(\theta) = -X^T \big(y - h_\theta(X)\big),$$

$$H = X^T R X,$$

where R is a diagonal matrix with entries $h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))$. We initialize with $\theta = \vec{0}$ and iterate until convergence.

```
def calculateGrad(self, theta):
    # Gradient of negative log-likelihood
    hTheta = self.sigmoid(np.dot(self.X, theta))  # (m,1)
    error = self.Y - hTheta
    grad = self.X.T @ error    # (n,1)
    return -grad

def calculateHessian(self, theta):
    hTheta = self.sigmoid(np.dot(self.X, theta)).flatten()
    r = hTheta * (1 - hTheta)
    # Efficient diagonal multiplication
    XtRX = self.X.T @ (r[:, np.newaxis] * self.X)
    return XtRX
```

The Newton update rule is:

$$\theta^{(t+1)} = \theta^{(t)} - H^{-1} \nabla L(\theta^{(t)}).$$

```
for epoc in range(1, maxEpocs + 1):
    grad = self.calculateGrad(theta)
    H = self.calculateHessian(theta)

    delta = np.linalg.pinv(H) @ grad
    theta_new = theta - alpha * delta

    newLoss = self.getLoss(theta_new)
```

The parameters used were:

- Maximum epochs: 100000
- Convergence threshold: 1×10^{-9}
- Learning rate(alpha): 0.001

3.1.1 Results

```
# Question 1
#  maxEpocs=100000, tolerance=1e-9, alpha=0.001
optimalTheta = model.optimizeLikelihoodFunction(theta)
# So the coefficients that I got are :
# Epoch 17313, Loss: 22.834145, Theta: [[ 0.40125296  2.58854703 -2.72558777]]
```

```
Starting loss: 69.31471805599453
Epoch 1000, Loss: 37.447526, Theta: [[ 0.0229086   0.71990046 -0.72732714]]
Epoch 2000, Loss: 27.215714, Theta: [[ 0.1013104   1.35451977 -1.38182374]]
Epoch 3000, Loss: 23.890227, Theta: [[ 0.21068115  1.89297098 -1.96502183]]
Epoch 4000, Loss: 23.034385, Theta: [[ 0.30515276  2.25903034 -2.36641457]]
Epoch 5000, Loss: 22.865945, Theta: [[ 0.36024556  2.45170238 -2.57685672]]
Epoch 6000, Loss: 22.838741, Theta: [[ 0.38522181  2.53561855 -2.66814351]]
Epoch 7000, Loss: 22.834782, Theta: [[ 0.3952178   2.56870152 -2.70406154]]
Epoch 8000, Loss: 22.834232, Theta: [[ 0.39901431  2.58119667 -2.71761666]]
Epoch 9000, Loss: 22.834157, Theta: [[ 0.40042725  2.58583741 -2.72264956]]
Epoch 10000, Loss: 22.834147, Theta: [[ 0.40094911  2.58755014 -2.72450681]]
Epoch 11000, Loss: 22.834145, Theta: [[ 0.40114131  2.58818076 -2.72519062]]
Epoch 12000, Loss: 22.834145, Theta: [[ 0.40121203  2.58841276 -2.72544217]]
Epoch 13000, Loss: 22.834145, Theta: [[ 0.40123804  2.58849808 -2.72553469]]
Epoch 14000, Loss: 22.834145, Theta: [[ 0.4012476   2.58852945 -2.7255687 ]]
Epoch 15000, Loss: 22.834145, Theta: [[ 0.40125112  2.58854099 -2.72558121]]
Epoch 16000, Loss: 22.834145, Theta: [[ 0.40125241  2.58854523 -2.72558581]]
Epoch 17000, Loss: 22.834145, Theta: [[ 0.40125288  2.58854679 -2.7255875 ]]
Epoch 17313, Loss: 22.834145, Theta: [[ 0.40125296  2.58854703 -2.72558777]]
Converged :: Optimal Values of theta that we got are :: [[ 0.40125296]
 [ 2.58854703]
 [-2.72558777]]!
```

With the current setting, after running Newton's Method, the algorithm converged at epoch 17313 with a loss of

$$\mathcal{L} = 22.834145,$$

and the learned parameters were:

$$\theta_0 = 0.4013, \quad \theta_1 = 2.5885, \quad \theta_2 = -2.7256.$$

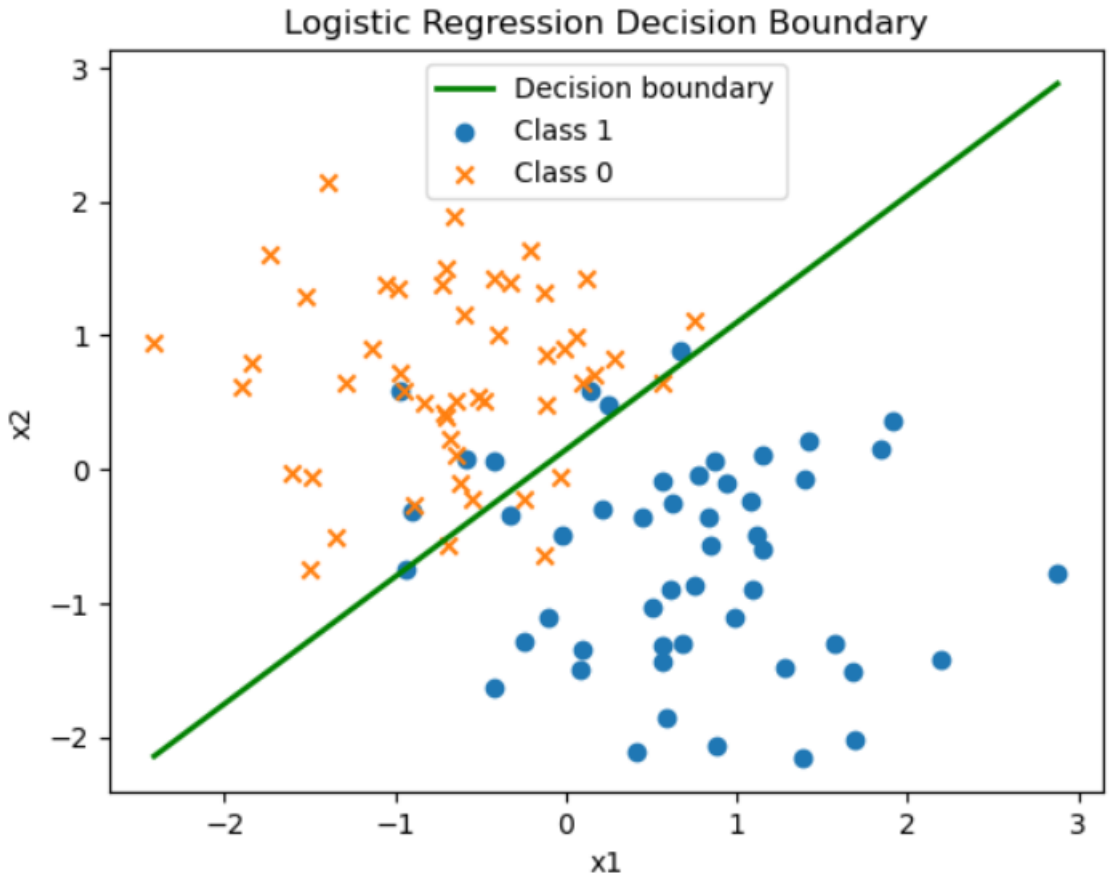
3.2 Plotting the Decision Boundary and Analysis

We visualize the dataset with x_1 on the horizontal axis and x_2 on the vertical axis. Points with label $y = 0$ are marked using one symbol (e.g., crosses), while points with label $y = 1$ are marked with another (e.g., circles).

The decision boundary is given by solving:

$$h_\theta(x) = 0.5 \quad \Rightarrow \quad \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0.$$

This results in a straight line separating the two classes. The plot clearly shows how the logistic regression classifier distinguishes between the two regions.



4 Gaussian Discriminant Analysis (GDA)

4.1 Evaluating Parameters for Linear GDA

Before evaluating, since GDA assumes the features to be independent and univariate gaussian distribution. We have to normalize the feature columns before computing the estimates. Let the training set be $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ with $x^{(i)} \in \mathbb{R}^2$ and $y^{(i)} \in \{0, 1\}$. The

maximum-likelihood estimates for the class priors and means are:

$$\phi = \frac{1}{m}\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\}, \qquad \mu_0 = \frac{1}{m_0}\sum_{i:y^{(i)}=0} x^{(i)}, \qquad \mu_1 = \frac{1}{m_1}\sum_{i:y^{(i)}=1} x^{(i)}$$

where $m_0 = \sum_i \mathbf{1}\{y^{(i)} = 0\}$ and $m_1 = \sum_i \mathbf{1}\{y^{(i)} = 1\}$.

Assuming identical covariance $\Sigma_0 = \Sigma_1 = \Sigma$, the pooled covariance estimate is:

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

On evaluating on the data for growth ring diameters in 2 regions, we got the following results:

Learned Parameters for Linear GDA

The prior probabilities are:

$$\phi = P(y = \text{Canada}) = 0.5, \qquad 1 - \phi = (y = \text{Alaska}) = 0.5$$

The class means are:

$$\mu_1 = \begin{bmatrix} 0.75529433 \\ -0.68509431 \end{bmatrix}, \qquad \mu_0 = \begin{bmatrix} -0.75529433 \\ 0.68509431 \end{bmatrix}$$

The shared covariance matrix is:

$$\Sigma = \begin{bmatrix} 0.42953048 & -0.02247228 \\ -0.02247228 & 0.53064579 \end{bmatrix}$$

4.2 Plotting the Training Data

Plot of the normalized training points with x_1 on the horizontal axis and x_2 on the vertical axis.

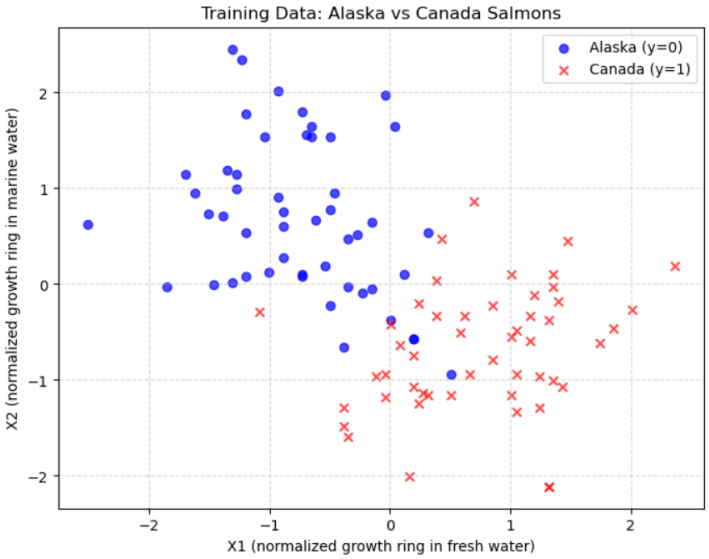


Figure 11: Training data: Canada vs Alaska (normalized).

4.3 Plotting the Decision Boundary from Linear GDA

For equal covariances, the decision boundary is linear. The discriminant (log-odds) reduces to:

$$(\mu_1 - \mu_0)^T \Sigma^{-1} x + \underbrace{\left(-\tfrac{1}{2}\mu_1^T \Sigma^{-1} \mu_1 + \tfrac{1}{2}\mu_0^T \Sigma^{-1} \mu_0 + \log \frac{\phi}{1-\phi} \right)}_{c} = 0.$$

Thus the linear boundary is:

$$(\mu_1 - \mu_0)^T \Sigma^{-1} x + c = 0.$$

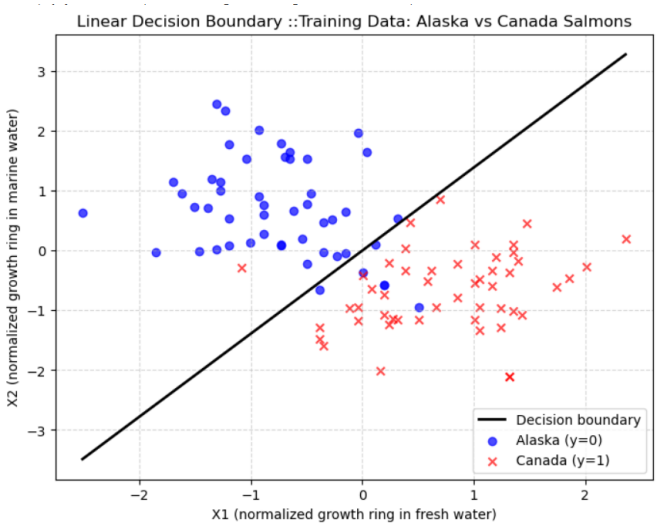


Figure 12: Decision Boundary learned by Linear GDA

4.4 Evaluating Parameters for Quadratic GDA

When classes have different covariances,we use class-specific MLEs:

$$\Sigma_0 = \frac{1}{m_0} \sum_{i:y^{(i)}=0} \big(x^{(i)} - \mu_0\big)\big(x^{(i)} - \mu_0\big)^T, \qquad \Sigma_1 = \frac{1}{m_1} \sum_{i:y^{(i)}=1} \big(x^{(i)} - \mu_1\big)\big(x^{(i)} - \mu_1\big)^T.$$

Learned Parameters for Quadratic GDA

The class means are(same as what we had for Linear GDA):

$$\mu_1 = \begin{bmatrix} 0.75529433 \\ -0.68509431 \end{bmatrix}, \qquad \mu_0 = \begin{bmatrix} -0.75529433 \\ 0.68509431 \end{bmatrix}$$

The covariance matrices are:

$$\Sigma_0 = \begin{bmatrix} 0.38158978 & -0.15486516 \\ -0.15486516 & 0.64773717 \end{bmatrix}, \qquad \Sigma_1 = \begin{bmatrix} 0.47747117 & 0.10992060 \\ 0.10992060 & 0.41355441 \end{bmatrix}$$

.

4.5 Plotting the Decision Boundary from Quadratic GDA

The quadratic boundary is obtained from equating the class discriminants:

$$\delta_1(x) - \delta_0(x) = 0,$$

where

$$\delta_k(x) = -\tfrac{1}{2}\log|\Sigma_k| - \tfrac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log\phi_k.$$

Expanding and rearranging gives a quadratic form in x:

$$x^T\big(\tfrac{1}{2}\big(\Sigma_0^{-1} - \Sigma_1^{-1}\big)\big)\,x + \big(\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1}\big)x + C \;=\; 0,$$

where

$$C \;=\; -\tfrac{1}{2}\mu_1^T\Sigma_1^{-1}\mu_1 + \tfrac{1}{2}\mu_0^T\Sigma_0^{-1}\mu_0 + \tfrac{1}{2}\log\frac{|\Sigma_0|}{|\Sigma_1|} + \log\frac{\phi_1}{\phi_0}.$$

The quadratic decision boundary is given by:

$$x^TPx + Q^Tx + R = 0$$

Substituting the learned parameters:

$$x^T\begin{bmatrix} 0.3356739 & 0.64341817 \\ 0.64341817 & -0.43296599 \end{bmatrix}x + \begin{bmatrix} 3.80785319 \\ -2.85967306 \end{bmatrix}^T x - 0.5847869867727312 = 0$$

where $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.

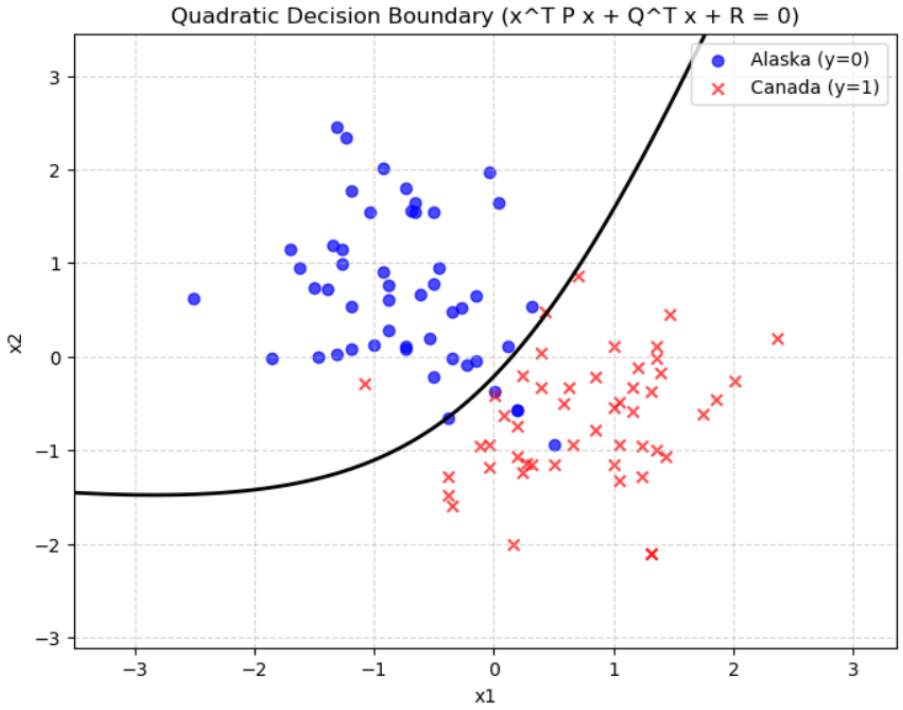


Figure 13: Data with quadratic (class-specific covariance) decision boundaries.

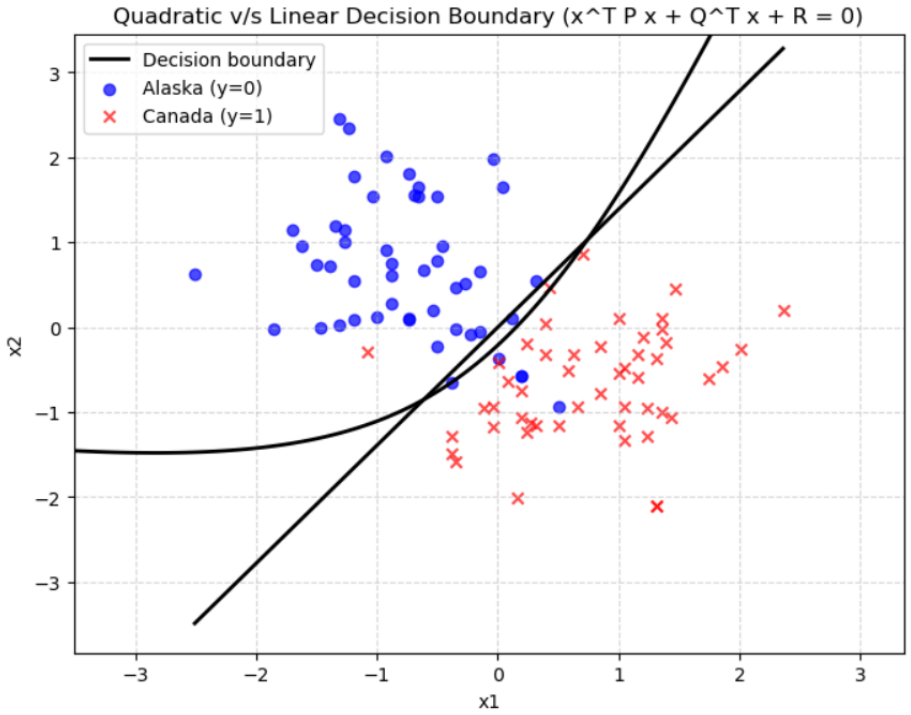


Figure 14: Data with linear (GDA equal-covariance) and quadratic (class-specific covariance) decision boundaries.

4.6 Analysis: Linear vs Quadratic GDA

From the experiments and visualizations, we can conclude the following:

- **Quadratic GDA Decision Boundary:** The boundary is curved because each class has its own covariance matrix. Hence, the decision function becomes quadratic in x. We observe the curve bending towards the class distribution of Alaska ($Y = 0$), meaning it adapts more flexibly to the class distributions.

  The determinants of the covariance matrices are:

  $$\det(\Sigma_0) = 0.2232 \quad \text{(Alaska)}, \qquad \det(\Sigma_1) = 0.1854 \quad \text{(Canada)}$$

  Since the covariance of Alaska ($Y = 0$) is larger, the quadratic curve bends more towards that class.

- **Linear GDA Decision Boundary:** Here we assume a shared covariance, i.e. $\Sigma_0 = \Sigma_1$. This explains why the separating boundary is linear.

- **Conclusions:**

  - Linear GDA assumes both classes share the same covariance matrix — this leads to a straight decision boundary. It is simpler and less flexible, but works well when the assumption holds.
  - Quadratic GDA allows different covariance matrices for each class — this results in a quadratic (curved) decision boundary. It is more flexible and fits the data better, but may risk overfitting when data is limited.

In this dataset, Quadratic GDA clearly adapts better to the real separation between Canada vs Alaska salmons.