COL774 Assignment 1

Rishabh Kumar (2021CS10103)

February 2025

1 Linear Regression(Gradient Descent)

Update Rule:

$$\nabla J(\theta) = -\frac{1}{m} X^T (Y - X\theta) \tag{1}$$

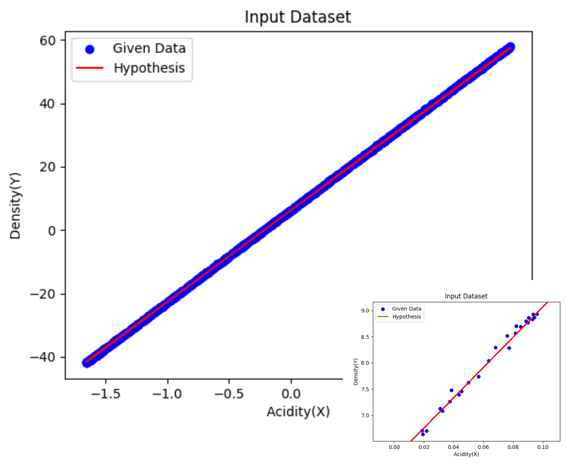
$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta^{(t)}) \tag{2}$$

1.1 Gradient Descent

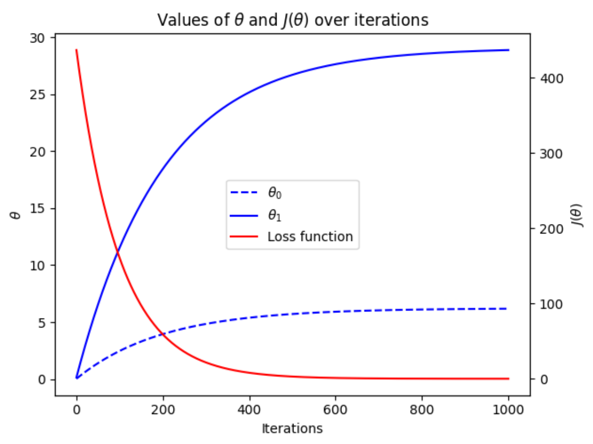
- **Learning Rate :** $\alpha = 0.1$
- **Stopping Criteria :**
  - $iterations > 1000$
  - $|J(\theta_i) - J(\theta_{i-1})| < 10^{-9}$
- **Learned Parameters :** $\theta_0 = 6.21865, \theta_1 = 29.06462256$
- **Observations :** Smaller learning rate fits data more efficiently i.e. up to 10^{-10} but require more iterations but in our case when $\alpha = 0.1$, it takes 240 iterations. Also, 10^{-9} is still good considering as it converges in much less iterations and cost decrease is far less.

1.2 Data and Learned Hypothesis function

Plotting the given data with normalised X on a two-dimensional graph and the hypothesis function learned by the GD algorithm with conditions in the previous part.



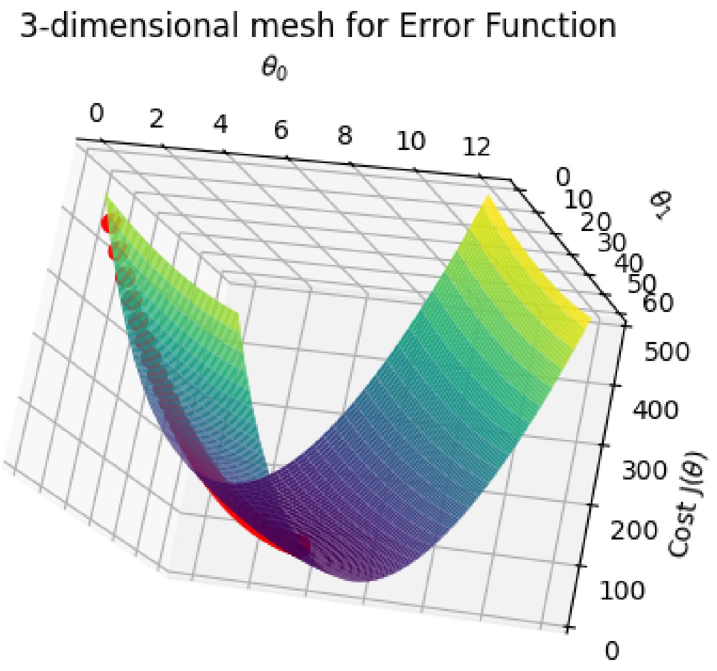
(a) Data and hypothesis function for linear regression



(b) Learning Path vs Iterations

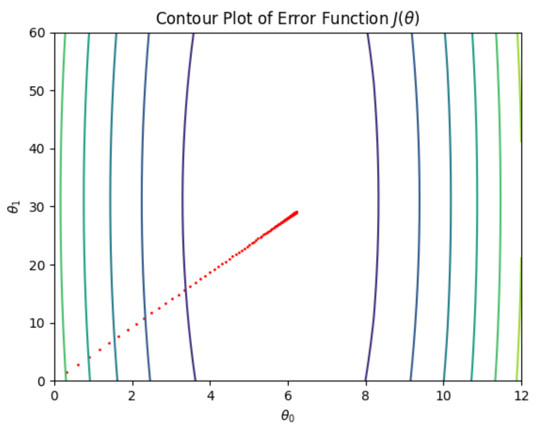
1.3 Loss vs Parameters 3D plot

The figure illustrates a 3-dimensional mesh showing the error function ($J(\theta)$) on the z-axis and the parameters in the x-y plane. Also, the red colored path displays the error value using the current set of parameters at each iteration of the gradient descent. The movement of this path can also be observed by running the code.



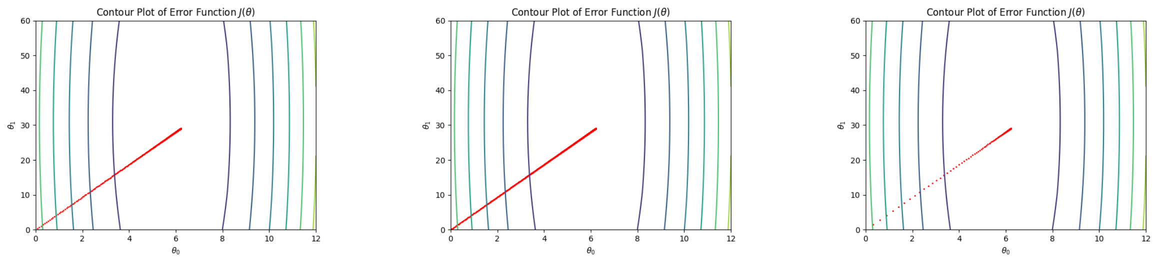
1.4 Contours of the error function

The figure is the contours of the error function at each iteration of the gradient descent. The movement of the red path can also be observed by running the code.



1.5 Contours for different step sizes

α	Number of Iterations	Final Cost	Final Theta
0.025	921	$4.874984338640748 \times 10^{-3}$	$\theta_0 = 6.21862015, \theta_1 = 29.06448156$
0.001	19920	$4.875944581913783 \times 10^{-3}$	$\theta_0 = 6.21838489, \theta_1 = 29.06337146$
0.1	240	$4.874954449907472 \times 10^{-3}$	$\theta_0 = 6.21865, \theta_1 = 29.06462256$



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

Figure 2: Contour plots for different α values.

As can be seen in Figure 2, the different step sizes converge in a very similar way. The final parameters learned after the entire training process were very similar for all three step sizes of 0.001, 0.025, and 0.1. From the table, with a learning rate of 0.1, the loss converged in 240 iterations, while with a step size of 0.025, it took 921 iterations. With a step size of 0.001, it took 19,920 iterations, specifically because smaller updates lead to more gradual progress towards the optimum, hence requiring a greater number of iterations. Allowing them to converge completely, the three experiments required 240, 921, and 19,920 iterations, respectively, implying that a larger learning rate allows the Gradient Descent algorithm to converge faster.

2 Sampling, Closed Form and Stochastic Gradient Descent

2.1 Sampling of Data

The following code snippet demonstrates generating sample data with the given parameters and given samplesize N. The data includes normally distributed variables x_1 and x_2 as two features of X, as well as noise (ϵ) added to the dependent variable y. The relationship between the variables is defined as:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \epsilon$$

```python
def generate(N, theta, input_mean, input_sigma, noise_sigma):
    """
    Generate normally distributed input data and target values
    Note that we have 2 input features
    """

    # Generate N samples each with 2 features from a normal distribution with mean=input_mean and std=input_sigma
    X = np.random.normal(loc=input_mean, scale=input_sigma, size=(N, 2))

    # Generate Gaussian noise with mean 0 and standard deviation noise_sigma for each sample
    noise = np.random.normal(0, noise_sigma, size=N)

    # Compute the target values as the linear combination of a bias (theta[0]) and weighted inputs (theta[1]*X[:,0] and theta[2]*X[:,1]) plus noise
    y = theta[0] + theta[1] * X[:, 0] + theta[2] * X[:, 1] + noise

    return X, y
```

2.2 Stochastic Gradient Descent

To implement SGD, after each epoch, the data is randomly shuffled and divided into batches of size r. The training update step is applied to each batch separately. For convergence, k determines the number of iterations over which the cost is averaged before checking the convergence criteria. Since smaller batch sizes introduce more noise in updates, a larger k is used to smooth fluctuations, ensuring stable convergence, whereas larger batch sizes require a smaller k due to inherently lower variance.

Let us define $AD(i,k)$ as:

$$AD(i,k) = \frac{1}{k}\sum_{j=i-k}^{i} |J(\theta_j) - J(\theta_{j-k})| \tag{3}$$

Batch Size(r)	Stopping Criteria	Learned Theta	Loss
1	$AD(i,1000) < 10^{-9}$	$\theta = [[2.97033309][1.03012371][2.24197106]]$	1.1568470424571805315
80	$AD(i,1000) < 10^{-9}$	$\theta = [[3.00395839][0.99655032][1.99370101]]$	1.0873634605028773
8000	$AD(i,100) < 10^{-9}$	$\theta = [[3.00023121][0.99949083][2.00005328]]$	1.0249893772267469
800000	$AD(i,1) < 10^{-9}$	$\theta = [[3.00210095][0.99922903][2.00108595]]$	1.0023109554536165

2.3 Part 3: Observations

Higher batch sizes lead to more stable updates, but since the learning rate is small, this effect is counterbalanced. Additionally, the cost follows, generally, a decreasing trend with increasing batch size due to improved convergence.

2.3.1 (a) Convergence Behavior

Across varying batch sizes, all implementations of SGD converge to nearly the same parameter values. However, smaller batch sizes take more iterations due to noisier updates, while larger batch sizes require fewer iterations but more epochs, leading to a trade-off between stability and efficiency.

2.3.2 (b) Closed-Form Solution

The closed-form solution for θ is:

$$\theta = [2.99793349, 1.0000041, 1.99974486]$$

The closed-form θ values closely match the true parameters used for generating the data.

2.3.3 (c) Comparison of Methods

Both the closed-form solution and SGD converge to true parameter values $\theta = [3, 1, 2]$, with larger batch sizes leading to smoother convergence. The closed-form solution provides the most precise estimate and is computationally efficient for our dataset, keeping in mind that we have only three parameters. However, for larger datasets, SGD remains practical, especially with appropriate batch size selection. While smaller batch sizes introduce noise and require more iterations, they allow SGD to approximate the optimal solution efficiently. Larger batch sizes stabilize updates but require more epochs to converge. Thus, while the closed-form solution was the most accurate in our case, SGD remains a viable alternative, particularly for large datasets, can handle high-dimensional data, suitable for online learning (updating model with new data as it arrives).

2.4 Error Analysis

The errors on the test data (20,000 samples from the original 1-million-sample dataset, with 20% used for testing) are summarized below:

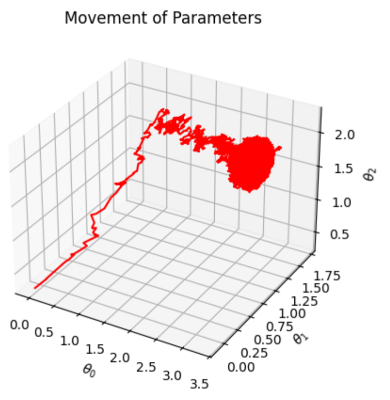
Batch Size (r)	**Loss (Trained θ)**	**Loss (Original θ)**
1	1.1377838300077106	1.001670488973027
80	1.0017862079503108	1.001670488973027
8000	1.0012981452475707	1.001295353967849
800000	0.9991471786544557	0.9991477666529902

Table 1: Test set loss comparison across different batch sizes.

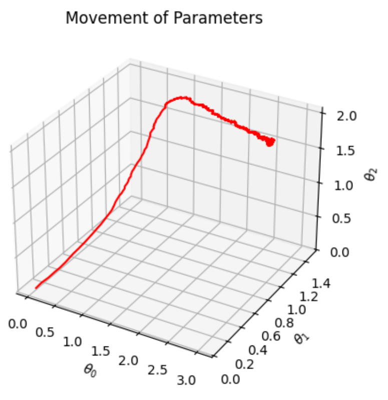
As batch size increases, both training and test errors decrease, indicating better convergence and generalization. The test error closely follows the training error, with smaller batch sizes showing higher training error due to noisier updates in SGD.

2.5 Movement of theta for different batch sizes

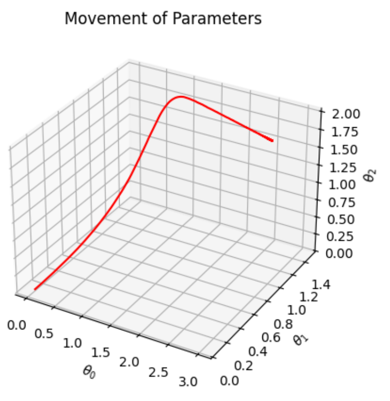
Observing the figures below, the movement of the parameter θ for a batch size of 1 is more noisy in nature. As the batch size increases, the curve becomes smoother. With a batch size of 80, some noisy nature is still visible, while batch sizes of 8,000 and 800,000 look completely smooth. This makes intuitive sense also because different batches of small numbers of samples will have much more variation than larger numbers of samples. Larger batch sizes involve averaging over more samples, leading to more uniformity. The large variation in the loss results in a large variation in the gradients, which in turn leads to the large variation in the movement of θ. In the extreme case of a batch size of 1, the loss changes after every sample, and thus, has the most variation.



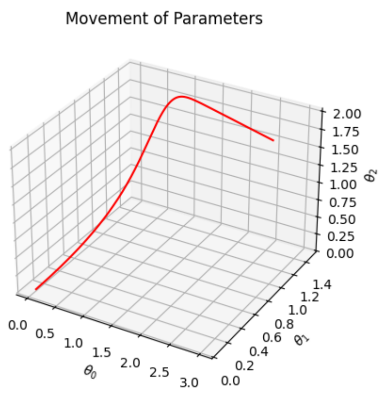
(a) Batch Size 1



(b) Batch Size 80



(c) Batch Size 8000



(d) Batch Size 800000

3 Logistic Regression

3.1 Part (a)

Using Log-likelihood loss function:

$$LL(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} log h_\theta(x^{(i)}) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))) \tag{4}$$

where Hypothesis function is:

$$h_\theta(x) = \frac{1}{1 + e^{-X\theta}} \tag{5}$$

Using Newton's Method of update discussed in the class:

$$\theta := \theta - H^{-1}\nabla_\theta.l(\theta) \tag{6}$$

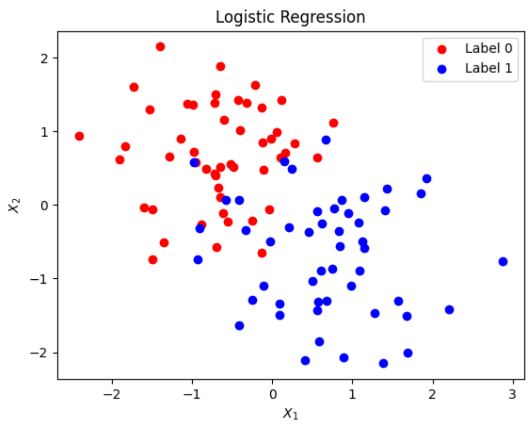
where the Hessian function is:

$$H_\theta(LL(\theta) = X^T.diag(\sigma(X.\theta)(1 - \sigma(X.\theta))).X \tag{7}$$

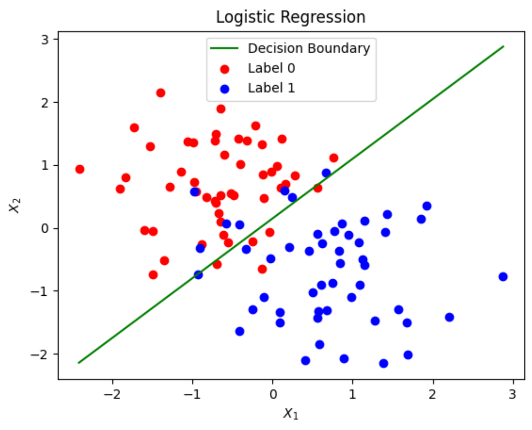
- **Convergence Criteria:** $|LL(\theta_i) - LL(\theta_{i-1})| < 10^{-6}$
- **Learned** θ = [0.40125316 2.5885477 -2.72558849]
- **Training Accuracy:** 88.00%

3.2 Plot of training data and Decision Boundary

The decision boundary is derived from the equation $h_\theta(x(i)) = 0.5$, also expressed as $\theta^T X = 0$. We will use the latter formulation to implement the decision boundary. This plot has been depicted in the below Figure.



(a) Training Data



(b) Training Data with Decision Boundary

4 Gaussian Discriminant Analysis

4.1 Part (a)

Following the equations in discussed in class for finding μ_0 and μ_1. The expressions are below:

$$\mu_0 = \frac{\sum_{i=1}^{m} 1\left\{y^{(i)} = 0\right\} x^{(i)}}{\sum_{i=1}^{m} 1\left\{y^{(i)} = 0\right\}}, \quad \mu_1 = \frac{\sum_{i=1}^{m} 1\left\{y^{(i)} = 1\right\} x^{(i)}}{\sum_{i=1}^{m} 1\left\{y^{(i)} = 1\right\}} \tag{8}$$

Assuming that both the classes have the same co-variance matrix i.e. $\Sigma_0 = \Sigma_1 = \Sigma$, we have its expression as below:

$$\Sigma = \frac{\sum_{i=1}^{m}(x^{(i)} - \mu_y^{(i)})(x^{(i)} - \mu_y^{(i)})^T}{m} \tag{9}$$

where,

$$\mu_y^{(i)} = 1\left\{y^{(i)} = 0\right\}\mu_0 + 1\left\{y^{(i)} = 1\right\}\mu_1 \tag{10}$$

Using above equations, the parameters obtained on the given data are:

- $\mu_0 = [-0.75529433\ 0.68509431]$
- $\mu_1 = [0.75529433\ \ -0.68509431]$
- $\Sigma = \begin{bmatrix} 0.42953048 & -0.02247228 \\ -0.02247228 & 0.53064579 \end{bmatrix}$
- $TrainingAccuracy = 93.00\%$

4.2 Part (b)

Here(5), the points with blue dots are data points that have label 'Canada' while those with red dots have label 'Alaska'. Below is the plot for normalised data:

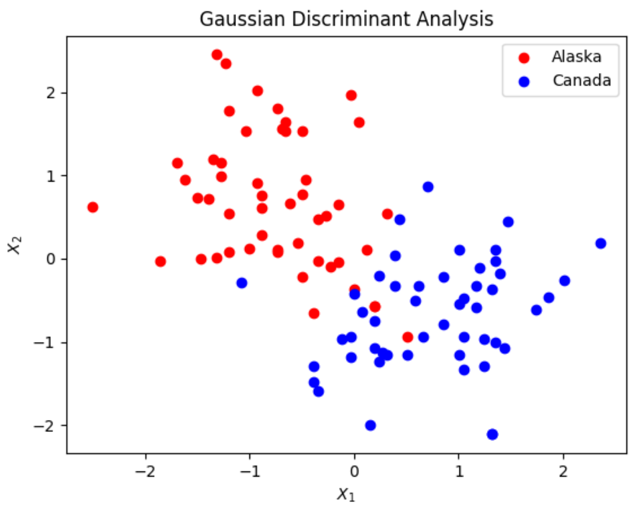


Figure 5: Normalised data for GDA

4.3 Part (c)

As discussed in class, at the decision boundary, $\theta^T X$ = h = 0. Thus, the decision boundary can be described in terms of μ_0, μ_1 and Σ as in below equation, which has been implemented directly in the code.

$$-(\mu_1 - \mu_0)^T\Sigma^{-1}x + \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0) + \log\frac{(1-\phi)}{\phi} = 0 \tag{11}$$

where,

$$\phi = \frac{\sum_{i=1}^{m} 1\left\{y^{(i)} = 1\right\}}{m} \tag{12}$$

```python
def plot_linear(self):
    sigma_inv = np.linalg.pinv(self.sigma)
    coef = np.dot((self.mu_1 - self.mu_0).T, sigma_inv)
    # Note: constant term is moved to the right side.
    const = 0.5 * (np.dot(np.dot(self.mu_1.T, sigma_inv), self.mu_1) - np.dot(np.dot(
    self.mu_0.T, sigma_inv), self.mu_0)) - np.log(self.phi/(1-self.phi))
    # For 2-D features, let x = [x1, x2]. We plot x2 in terms of x1.
    x1_vals = np.linspace(-3, 3, 100)
    x2_vals = (const - coef[0] * x1_vals) / coef[1]
    return x1_vals, x2_vals
x1_vals, x2_vals = plot_linear()
plt.plot(x1_vals, x2_vals, color='green')
```

Below is the plot (6) of the decision boundary found by using parameters we get in Part A.

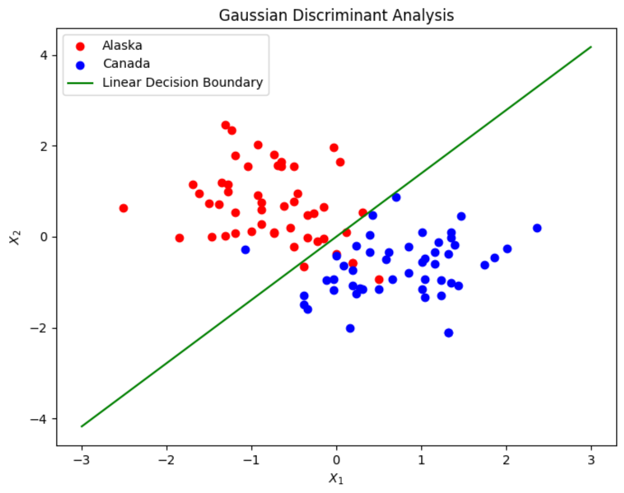


Figure 6: Linear Decision boundary for GDA

4.4 Part (d)

In this part we will calculate parameters for general GDA, where $\Sigma_0 \neq \Sigma_1$ and calculated using expressions below:

$$\Sigma_0 = \frac{\sum_{i=1}^{m} 1\left\{y^{(i)} = 0\right\}(x^{(i)} - \mu_y^{(i)})(x^{(i)} - \mu_y^{(i)})^T}{\sum_{i=1}^{m} 1\left\{y^{(i)} = 0\right\}} \tag{13}$$

$$\Sigma_1 = \frac{\sum_{i=1}^{m} 1\left\{y^{(i)} = 1\right\}(x^{(i)} - \mu_y^{(i)})(x^{(i)} - \mu_y^{(i)})^T}{\sum_{i=1}^{m} 1\left\{y^{(i)} = 1\right\}} \tag{14}$$

where $\mu_y^{(i)}$ is the same as that used in eq. 10 and also for finding μ_0 and μ_1 we can use same equation as eq. 8. Using above equations, the parameters obtained on the given data are:

- $\mu_0 = [-0.75529433\ 0.68509431]$
- $\mu_1 = [0.75529433\ \ -0.68509431]$
- $\Sigma_0 = \begin{bmatrix} 0.38158978 & -0.15486516 \\ -0.15486516 & 0.64773717 \end{bmatrix}$
- $\Sigma_1 = \begin{bmatrix} 0.47747117 & 0.1099206 \\ 0.1099206 & 0.41355441 \end{bmatrix}$
- $TrainingAccuracy = 93.00\%$

4.5 Part (e)

In this case, we get a parabolic decision boundary which can be described in terms of μ_0, μ_1, Σ_0 and Σ_0 as in below equation:

$$\frac{1}{2}(x^T(\Sigma_1^{-1} - \Sigma_0^{-1})x) - (\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1})x + \frac{1}{2}(\mu_1^T\Sigma_1^{-1}\mu_1 - \mu_0^T\Sigma_0^{-1}\mu_0) + \log(\frac{(1-\phi|\Sigma_1|^{\frac{1}{2}})}{\phi|\Sigma_0|^{\frac{1}{2}}}) = 0 \quad (15)$$

The direct implementation of above quadratic equation in the code, can be seen in the code snippet below. And fig. 7 illustrates the plot.

```python
def plot_quadratic(self):
    sigma_0_inv = np.linalg.pinv(self.sigma_0)
    sigma_1_inv = np.linalg.pinv(self.sigma_1)
    d_0 = np.sqrt(np.abs(np.linalg.det(self.sigma_0)))
    d_1 = np.sqrt(np.abs(np.linalg.det(self.sigma_1)))
    coeff_term = 0.5 * (sigma_1_inv - sigma_0_inv)
    linear_term = np.dot(self.mu_1.T, sigma_1_inv) - np.dot(self.mu_0.T, sigma_0_inv)
    const_term = (0.5 * (np.dot(np.dot(self.mu_1.T, sigma_1_inv), self.mu_1) - np.dot(np
    .dot(self.mu_0.T, sigma_0_inv), self.mu_0))
                 + np.log((1-self.phi)*d_0/(self.phi*d_1)))
    # Create a meshgrid for plotting
    x1_vals = np.linspace(-3, 3, 200)
    x2_vals = np.linspace(-3, 3, 200)
    x1, x2 = np.meshgrid(x1_vals, x2_vals)
    term1 = coeff_term[0, 0]*x1**2 + (coeff_term[0, 1] + coeff_term[1, 0])*x1*x2 +
    coeff_term[1, 1]*x2**2
    term2 = linear_term[0]*x1 + linear_term[1]*x2
    h = term1 - term2 + const_term
    contour_plot = plt.contour(x1, x2, h, levels=[0], colors='orange')
    return contour_plot
```

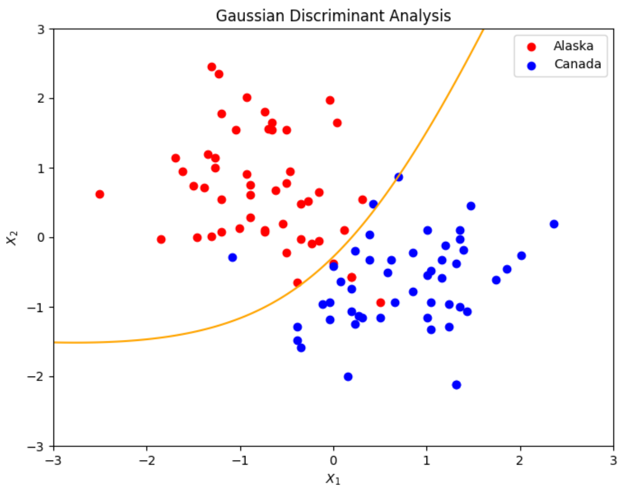


Figure 7: Quadratic Decision Boundary for GDA

4.6 Part (f) : Analysis of Decision Boundaries

- In figure 8, we can observe a small gap between the two decision boundaries. Within this gap, approximately 2-3 samples are labeled as "Alaska". Comparing this to figure with linear decision boundary, it becomes evident that the quadratic decision boundary correctly classifies these few samples, which were previously misclassified by the linear boundary.
- Regarding the curvature of the decision boundaries, it's notable that the curvature tends to point towards the center of the cluster with the smaller magnitude of $|\Sigma_i|$ or, equivalently, the larger magnitude of $|\Sigma_i^{-1}|$. Here in our case, $|\Sigma_0| < |\Sigma_1|$, the boundary curves towards the cluster labeled as 0, which corresponds to the Alaska cluster.

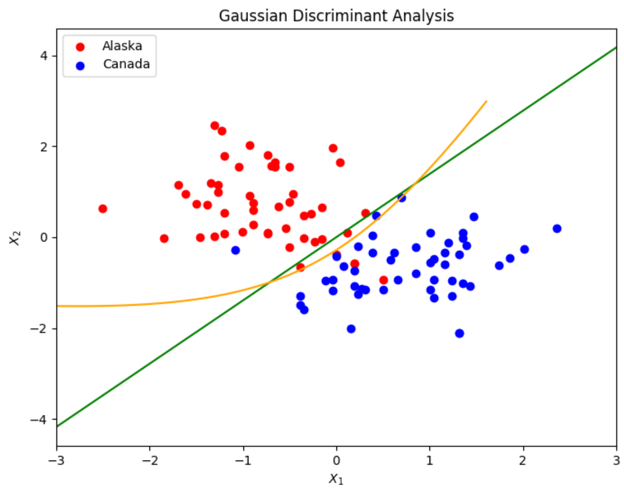


Figure 8: Both Linear and Quadratic Boundaries together

- Intuitively, the magnitude of curvature of the boundary is inversely proportional to the absolute difference between the respective $|\Sigma_i^{-1}|$ values. When both $|\Sigma_i^{-1}|$ values are the same, the curvature is expected to increase infinitely, which is precisely what occurs when the linear boundary is obtained.