COL774 Assignment 1

Mustafa Chasmai

September 2021

1. Linear Regression

(a) Batch Gradient Descent

- Implemented as a class 'GradientDescent', allowing me to have a simple API reusable with subsequent quesitons.
- The main parameter update step during training is in this function:

```
def training_step(self, data):
    (X, Y) = data
    m = Y.shape[0]
    # computing gradient
    self._grad = - np.dot(np.transpose(X), (Y -  np.dot(X, self._theta))) / m
    self._grad = self._grad.reshape(-1, 1)
    # optimisation step
    self._theta = self._theta - self.lr * self._grad
```

Thus, it is effectively implementing the vectorised versions of the standard gradient descent equations:

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

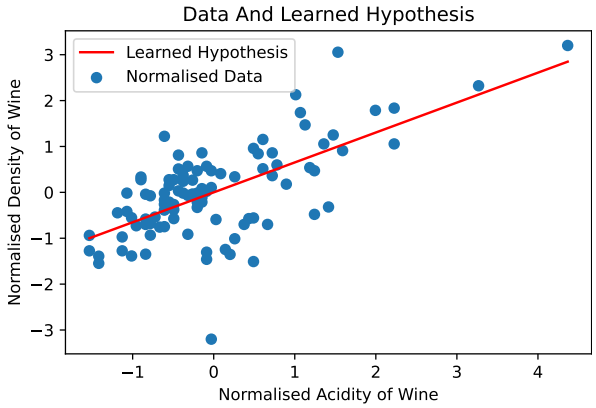
$$\theta = \theta - \alpha \nabla J(\theta) \tag{2}$$

where the symbols follow the same notation used in the course lectures.

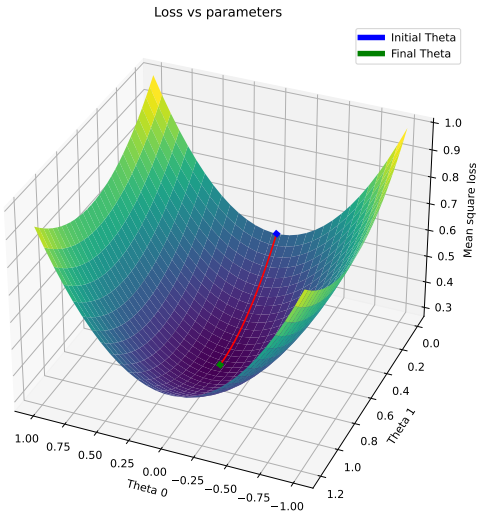
- The other relevant functions in the class are: **load_data()**, that reads the dataset files and saves X and Y matrices after normalising and preprocessing; **loss()**, that implements the least squares error metric that is used; and **train()**, that loops over the training iterations, calls the training step with appropriate arguments, stores the parameter values for each iteration and ensures proper termination of training.
- **Learning rate:** 0.1
- **Stopping Criteria:** $J(\theta_i) - J(\theta_{i-1}) \leq 0.00001$
- **Parameters Learned:** $\theta = \begin{bmatrix} -2.43933661e-15 \\ 6.61088296e-01 \end{bmatrix} \approx \begin{bmatrix} 0 \\ 0.66 \end{bmatrix}$

(b) Data and Learned Hypothesis function

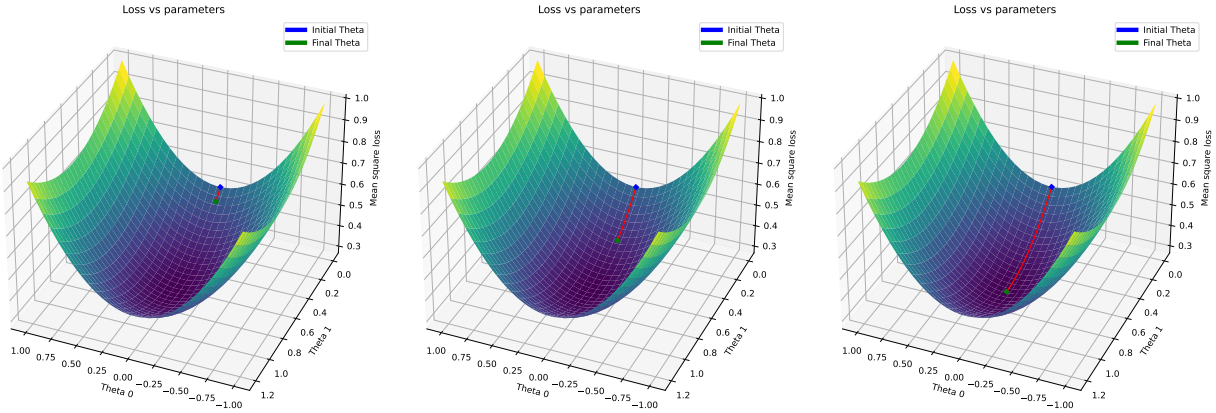
The data points are drawn as blue dots at the corresponding (x, y) coordinates, using scatter plot functionality provided by matplotlib. The learned hypothesis is drawn by getting y' values corresponding to each of the data x values. These y' values are obtained by the formula $\mathbf{Y} = \mathbf{X}\theta$ using θ parameters learned by the gradient descent algorithm. Matplotlib's line plot functionality was used to plot these (x, y') pairs on the graph.



(c) Loss vs Parameters 3D plot

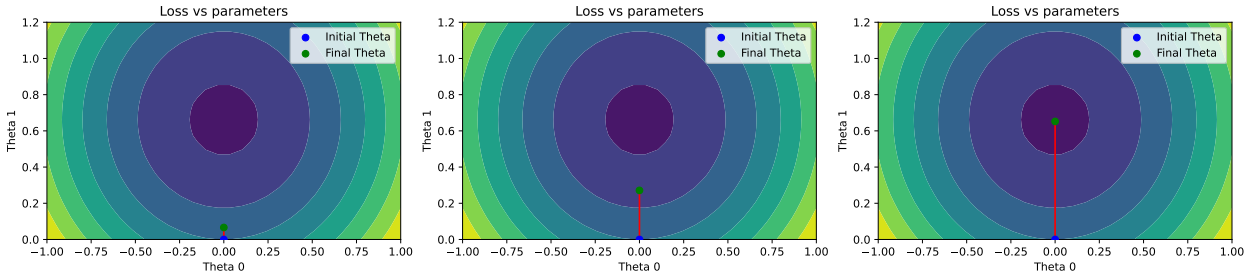


The loss is plotted by considering an interval of theta parameters for the x and y axes, and then calculating loss values for uniformly samples points in this interval. Matplotlib's plot_surface() function was used for the same. On top of the 3D mesh/surface, the (θ_0, θ_1, J(θ_0, θ_1)) points are plotted as a line plot. These values of parameters are the same ones stored during training, having one pair (θ_0, θ_1) learned after each iteration. The blue point corresponds to the initial parameters (initialised to 0s), while the green point corresponds to the final parameters.

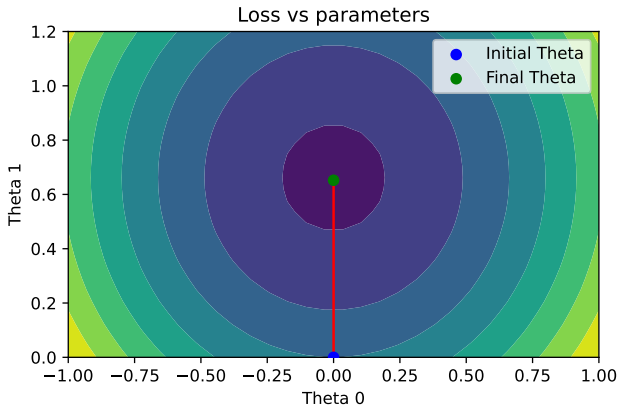


The plotting was animated to show the learning stage in each iteration. Some intermediate plots are shown above. The complete animation can be seen by running the submitted script for this question.

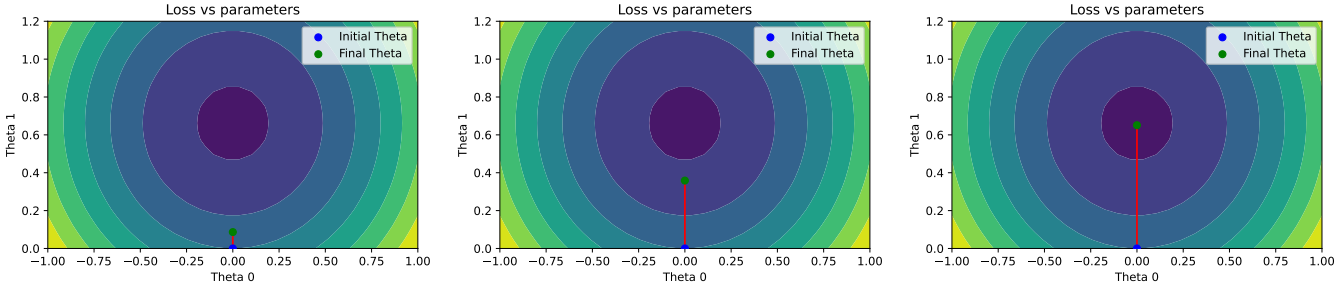
(d) Loss vs Parameters Contour plot



The same (θ_0, θ_1, J(θ_0, θ_1)) points obtained in the last part were used to plot the contour plot as well. The matplotlib contourf function was used to get filled contours, and the line plot was used to visualise the learned parameters. Again, this plot was animated, and some intermediate plots were shown above. The final plot obtained after the complete training is shown below.

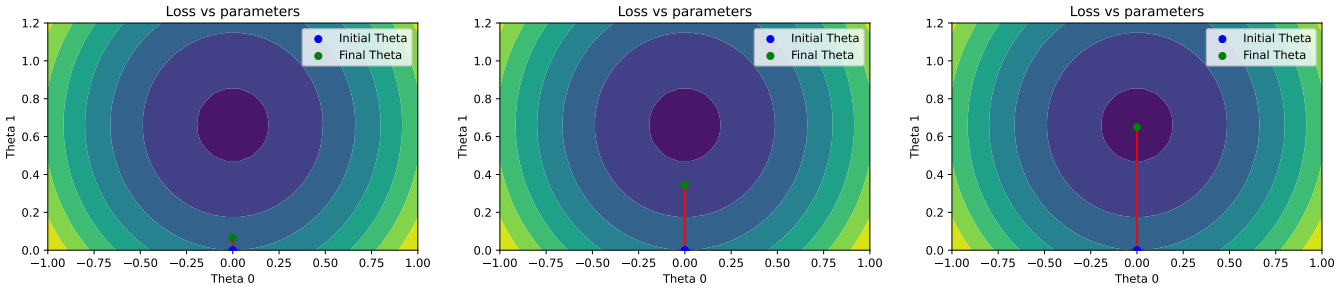


(e) Contours for different step sizes



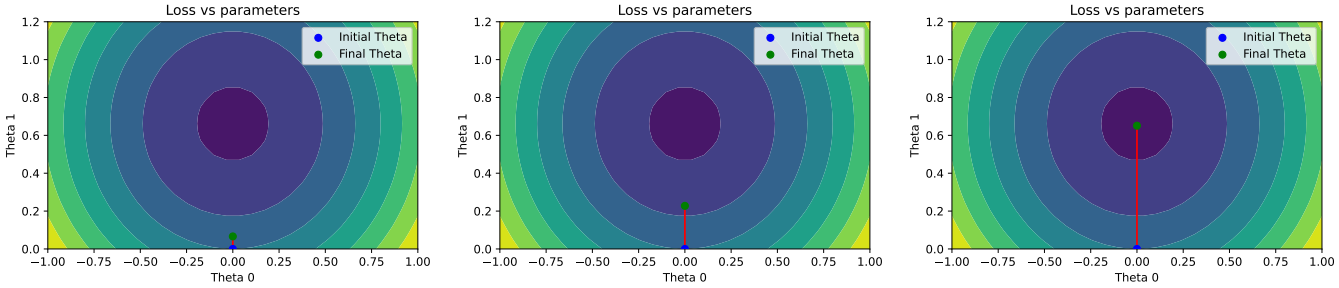
(a) At Iteration 160 (b) At Iteration 800 (c) At Iteration 4150

Figure 1: Contour plots at different iterations for step size 0.001. Total 4192 Iterations before stopping



(a) At Iteration 5 (b) At Iteration 30 (c) At Iteration 160

Figure 2: Contour plots at different iterations for step size 0.025. Total 167 Iterations before stopping



(a) At Iteration 2 (b) At Iteration 5 (c) At Iteration 40

Figure 3: Contour plots at different iterations for step size 0.1. Total 41 Iterations before stopping

As can be seen in Fig 1, Fig 2 and Fig 3, 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. The major difference between the three experiments was the rate at which the algorithm converged. The three experiments required 4192, 167 and 41 iterations respectively to converge, implying that a larger step size allows the Gradient Descent algorithm to converge faster.

2. Sampling and Stochastic Gradient Descent

(a) Sampling

For sampling from a general Gaussian distribution, a simple class NormalDistribution was implemented:

```
1  class NormalDistribution():
2      def __init__(self, mean, var, seed = 0):
3          self.mu = mean
4          self.sigma = np.sqrt(var)
5          np.random.seed(seed)
6
7      def sample(self):
8          return np.random.normal(self.mu, self.sigma)
9
10 # Initialising the Normal Distribution
11 x1 = NormalDistribution(3, 4)
12 # Sampling a value from the distribution
13 sample = x1.sample()
```

This provides a very intuitive interface. The statement x $\sim \mathcal{N}(3, 4)$ is translated directly to line 10 in the value snippet. Line 11 demonstrates how to sample a single point. This function was used to sample the million values of $x_1 \sim \mathcal{N}(3, 4)$, $x_2 \sim \mathcal{N}(-1, 4)$ and $\epsilon \sim \mathcal{N}(0, 2)$. Then these values were used to get values of y according to the formula $\vec{y} = \vec{3} + \vec{x_1} + 2\vec{x_2} + \vec{\epsilon}$, where each million-dimensional vector contains the million sampled values. Since x_1, x_2 and ϵ correspond to different objects in the code, and since they are initialised with different seeds, the sampling process is independent.

(b) Stochastic Gradient Descent

To implement SGD, the training step where the parameter update takes place, was taken directly from the gradient descent implementation in question 1. Here, only the driver train() function was modified, along with the corresponding stopping criteria. The relevant modifications in train() can be seen in the code snippet below.

```
no_batches = self.X.shape[0] // self.batch_size
for iter in range(max_iter):
    self.shuffle()
    for batch in range(no_batches):
        x_batch = self.X[batch * self.batch_size : (batch + 1) * self.batch_size]
        y_batch = self.Y[batch * self.batch_size : (batch + 1) * self.batch_size]
        self.training_step((x_batch, y_batch))
```

In summary, at each iteration, the data is randomly shuffled, and then divided into batches. The training update step is called with the data in each of these batches separately.

Experiments:

With the same data, and the same learning rate of 0.001, the learned parameters and stopping criteria for each batch size are given below:

Batch Size	Stopping Criteria	Learned Parameters	Approx
1	some filler	$\theta = []$	$\theta = []$
100	some filler	$\theta = []$	$\theta = []$
10000	some filler	$\theta = []$	$\theta = []$
1000000	some filler	$\theta = []$	$\theta = []$

The convergence criteria were chosen using some weird logic, which is described here.

(c) The parameters learned with all of these batch sizes are very similar, and so are the difference from the original hypothesis. The speed of convergence and the number of iterations, however are quite different. The observations obtained are reported below:

Batch Size	Distance b/w θs	Time to converge	No. of iterations	Time per iteration
1				
100				
10000				
1000000				

3. Logistic Regression

(a) For this question, the training step of Gradient Descent used in the first question was modified, and the rest of the training process was kept the same.

```
def training_step(self, data):
    (X, Y) = data
    # computing hessian
    sigmoid = 1/(1+np.exp(-np.dot(X, self._theta)))
    self.hessian = np.dot(np.transpose(X), X) * np.diag(sigmoid) * np.diag(1 - sigmoid)
    hessian_inv = np.linalg.pinv(self.hessian)
    # computing gradient
    self._grad = - np.dot(np.transpose(X), (Y -  sigmoid))
    self._grad = np.mean(self._grad, axis=1).reshape(-1, 1)
    # optimisation step
    self._theta = self._theta - self.lr * np.dot(hessian_inv, self._grad)
```

The formula used for calculating the hessian for the log-likelihood loss is in eq 3, and is derived rigorously in Appendix **??**. The vectorised version of this equation, that is actually implemented in the code, is in eq 4

$$H = \sum_{i=1}^{m} x^{(i)}(x^{(i)})^T h_\theta(x^{(i)})(1 - h_\theta(x^{(i)})) \tag{3}$$

$$\mathbf{H} = \mathbf{X}^T\mathbf{X}\ Diag(h_\theta(\mathbf{X}))\ Diag(1 - h_\theta(\mathbf{X})) \tag{4}$$

where $Diag(M_{n\times n})$ is the diagonal operator, which returns a n dimensional vector containing the diagonal elements of $M_{n\times n}$. The remaining symbols follow notations used in the lectures.

Other than the core training, another modification was that normalisation was only applied on the $\mathbf{X}$ inputs, and not the $\mathbf{Y}$ labels. The mean squared loss was also replaced with the log-likelihood loss.

The hyperparameters used, and final parameters obtained are:

- Learning Rate: 0.1
- Total Iterations: 42
- Losses: Inital 0.693 Final 0.225
- Stopping Criteria: Difference in loss ≤ 0.001
- Learned Parameters: $\theta_0 = 0.434$, $\theta_1 = 2.471$, $\theta_2 = -2.684$

(b) The decision boundary is obtained by the equation $h_\theta(x^{(i)}) = 0.5$ or equivalently, $\theta^T x = 0$. The latter equation is easier to implement and hence, used to plot the decision boundary here. Similar to question 1, a grid of intervals in the two dimensions of the input is considered, and uniformly sampled points are plotted as a contour, with a single level of value 0. Thus, effectively, the $(1, x_1, x_2)$ vectors from the grid that have $\theta x^T = 0$ are plotted as a line, which gives us the decision boundary in Fig 5.

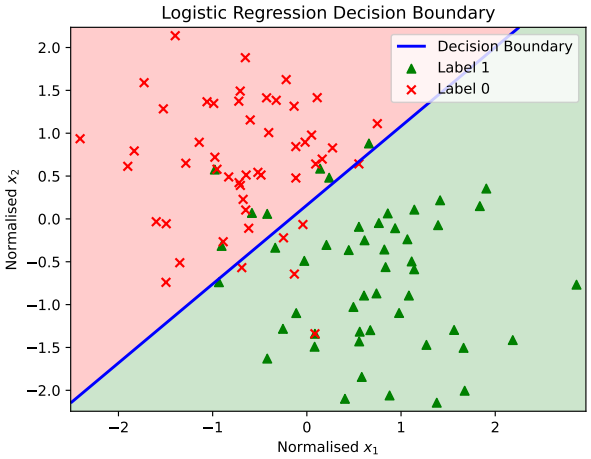


Figure 4: Decision Boundary and Normalised data for Logistic Regression

4. Gaussian Discriminant Analysis

The core implementation of the algorithm is in the train() function as described below. It uses the data to calculate all the parameters needed for GDA using the closed form solutions discussed in course lectures.

```
1  def train(self):
2      indicator_0 = 1 * (self.Y == 0).reshape(-1, 1)
3      indicator_1 = 1 * (self.Y == 1).reshape(-1, 1)
4      m = self.Y.shape[0]
5
6      self._phi = np.sum(indicator_1) / m
7
8      self._mu_0 = np.sum(indicator_0 * self.X, axis=0) / np.sum(indicator_0)
9      self._mu_1 = np.sum(indicator_1 * self.X, axis=0) / np.sum(indicator_1)
10
11     # with the assumption that sig_0 = sig_1
12     self._mu = indicator_0 * self._mu_0 + indicator_1 * self._mu_1
13     self._sig = np.dot(np.transpose(self.X - self._mu), self.X - self._mu) / m
14
15     # General case when sig_0 is not the same as sig_1
16     self._sig_0 = np.dot(np.transpose(self.X - self._mu_0), indicator_0 * (self.X - self._mu_0)) /
        np.sum(indicator_0)
17     self._sig_1 = np.dot(np.transpose(self.X - self._mu_1), indicator_1 * (self.X - self._mu_1)) /
        np.sum(indicator_1)
```

Listing 1: Calculating parameters for GDA

(a) In the above code snippet, lines 8 and 9 are used to calculate the two means μ_0 and μ_1. They follow exactly the equations in eq 5 discussed in class. This step is common for both part (a) and part (d).

$$\mu_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}, \qquad \mu_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \tag{5}$$

The lines 12 and 13 calculate the covariance matrix Σ. Again, this follows the same equations in eq 6 discussed in class. Note that the closed form solution of $\mu_{y^{(i)}}$ is equivalent to the definition discussed in class.

$$\mu_{y^{(i)}} = 1\{y^{(i)} = 0\}\mu_0 + 1\{y^{(i)} = 1\}\mu_1, \qquad \Sigma = \frac{\sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{m} \tag{6}$$

Using these equations, the actual parameter values obtained by using the given data are:

- μ_0: [-0.75529433, 0.68509431]
- μ_1: [0.75529433, -0.68509431]
- Σ: [[0.42953048, -0.02247228], [-0.02247228, 0.53064579]]

(b) Here, the normalised growth ring parameter is fresh and marine water provided in the data are visualised along with the labels 'Alaska' or 'Canada'. The points with green triangles have label 'Canada' while those with red crosses have label 'Alaska'.

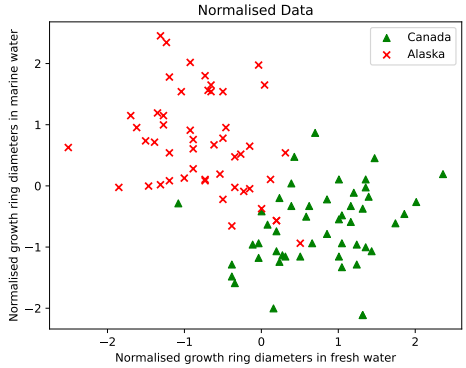


Figure 5: Normalised data for Gaussian Discriminant Analysis

(c) As has been discussed in class, at the decision boundary, $log(A) = -\theta^T x = h = 0$. Thus, the decision boundary can be described in terms of μ_0, μ_1 and Σ as:

$$-(\mu_1 - \mu_0)^T \Sigma^{-1} x + \frac{1}{2}(\mu_1^T \Sigma^{-1} \mu_1 - \mu_0^T \Sigma^{-1} \mu_0) + log(\frac{1-\phi}{\phi}) = 0 \tag{7}$$

This is implemented directly in the code, as can be seen in the code snippet below.

```
coefficient_term = -1 * np.dot(np.transpose(self._mu_1 - self._mu_0), sig_inv)
linear_term = np.dot(X, coefficient_term)

constant_term_1 = np.dot(np.dot(np.transpose(self._mu_1), sig_inv), self._mu_1)
constant_term_2 = np.dot(np.dot(np.transpose(self._mu_0), sig_inv), self._mu_0)
constant_term_3 = np.log((1 - self._phi)/self._phi)
constant_term = (constant_term_1 - constant_term_2) / 2 + constant_term_3

h = linear_term + constant_term
```

Here h is calculated term by term for uniformly sampled values of x in an interval. Then a contour is plotted with the level at 0 to get the decision boundary, as in the case of question 3.

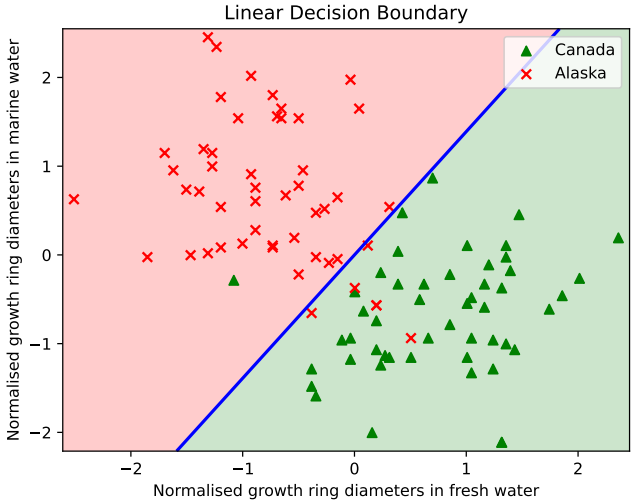


Figure 6: Linear Decision boundary for Gaussian Discriminant Analysis

(d) The parameters for the general case of GDA are also calculated in a similar way as in the above case. The implementation can be seen in lines 16 and 17 in Code 1. The values of μ_0 and μ_1 are exactly same as given by eq 5. The values of Σ_0 and Σ_1 are computed using eq 8.

$$\Sigma_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}, \quad \Sigma_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \tag{8}$$

where $\mu_{y^{(i)}}$ is the same as that used in eq 6, i.e. the one given in eq 9.

$$\mu_{y^{(i)}} = 1\{y^{(i)} = 0\}\mu_0 + 1\{y^{(i)} = 1\}\mu_1 \tag{9}$$

Using these equations, the actual parameter values obtained by using the given data are:

- 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]]

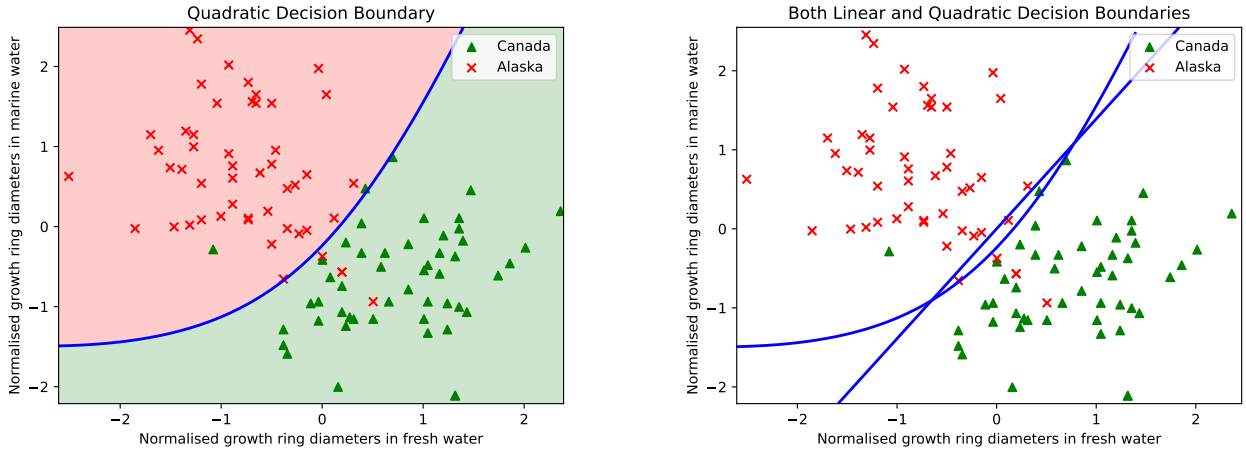
(e) Following the similar process as used in part (c), we get the quadratic decision boundary in eq 10.

$$\frac{1}{2}(x^T(\Sigma_1^{-1} - \Sigma_0^{-1})x) - (\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1})x + \frac{1}{2}(\mu_1^T\Sigma_1^{-1}\mu_1 - \mu_0^T\Sigma_0^{-1}\mu_0) + log(\frac{1-\phi}{\phi}\frac{|\Sigma_1|^{\frac{1}{2}}}{|\Sigma_0|^{\frac{1}{2}}}) = 0 \tag{10}$$

This is implemented directly in the code, as can be seen in the code snippet below.

```
quadratic_term = np.dot(np.dot(X, sig_1_inv - sig_0_inv), np.transpose(X))

coefficient_term = -2 * (np.dot(sig_1_inv, self._mu_1) - np.dot(sig_0_inv, self._mu_0))
linear_term =  np.dot(X, coefficient_term)

constant_term_1 = np.dot(np.dot(np.transpose(self._mu_1), sig_1_inv), self._mu_1)
constant_term_2 = np.dot(np.dot(np.transpose(self._mu_0), sig_0_inv), self._mu_0)
constant_term_3 = np.log((1 - self._phi)/self._phi)
constant_term_4 = (np.log(np.linalg.norm(self._sig_1) / np.linalg.norm(self._sig_0)))/2

constant_term =  1/2(constant_term_1 - constant_term_2) + constant_term_3 + constant_term_4

h = np.diag(quadratic_term) + linear_term + constant_term
```

Here h is calculated term by term for uniformly sampled values of x in an interval. Then a contour is plotted with the level at 0 to get the decision boundary, as in the case of part(c).



(a) Quadratic Decision Boundary for GDA

(b) Both Linear and Quadratic Boundaries together

Figure 7: Quadratic Decision Boundary obtained by Gaussian Discriminant Analysis

(f) In Fig 7b, in the small gap between the two decision boundaries, we can see around 2-3 samples with labels Alaska. As can be seen from Fig 6 and Fig 7a, The quadratic decision boundary correctly classifies these few samples which had been missed out by the linear boundary.

For the direction of curvature of the boundaries, it can be observed that the boundary's curvature is towards the center of the cluster with smaller $|\Sigma_i|$ magnitude. In case of Fig 7a, $|\Sigma_0| < |\Sigma_1|$, and thus, it curves towards the cluster with label 0, i.e., the Alaska cluster.

Intuitively, the curvature magnitude of the boundary seems to be inversely proportional to the absolute difference of the respective Σ_i or more precisely, Σ_i^{-1}. In the case when both Σ_i are the same, the curvature would be expected to go to ∞, which is exactly what happens when we get the linear boundary. This relation of the radius of curvature of the boundary can be formally derived using the general boundary equation in eq 10.