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:

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

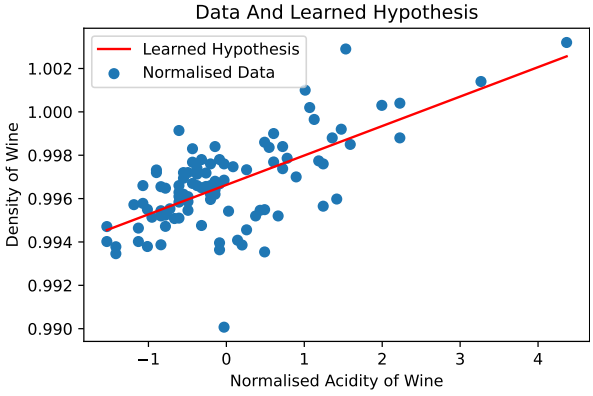
$$\theta = \theta - \eta \nabla J(\theta) \tag{2}$$

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

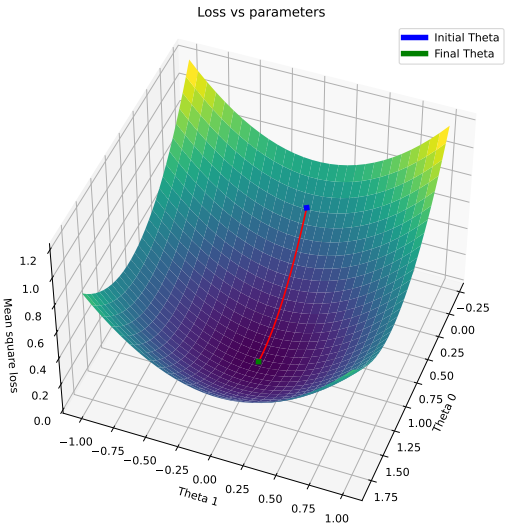
- The other relevant functions in the class are: **load_data()**, that reads the dataset files and saves X and Y matrices after normalising and preprocessing and **train()**, that loops over the training iterations, stores the parameter values for each iteration and ensures proper termination of training.
- **Learning rate:** 0.1
- **Stopping Criteria:** $|J(\theta_i) - J(\theta_{i-1})| \leq 10^{-6} J(\theta_{i-1})$
- **Parameters Learned:** $\theta = \begin{bmatrix} 0.99663124 \\ 0.00135794 \end{bmatrix}$

(b) Data and Learned Hypothesis function

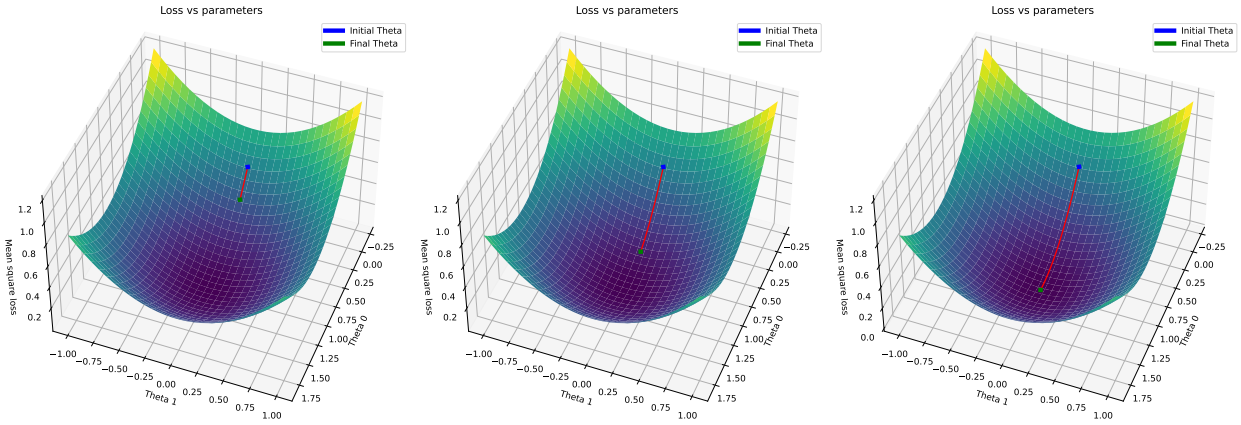
The data points are drawn as blue dots at the corresponding (x, y) coordinates, using scatter plot functionality provided by matplotlib. The learned hypothesis is drawn by getting y' values corresponding to each of the data x values. These y' values are obtained by the formula $\mathbf{Y} = \mathbf{X}\theta$ 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

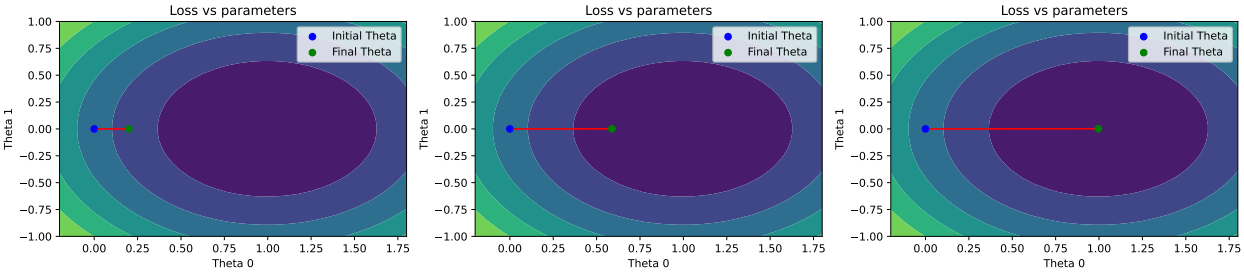


The loss is plotted by considering an interval of theta parameters for the x and y axes, and then calculating loss values for uniformly samples points in this interval. Matplotlib's plot_surface() function was used for the same. On top of the 3D mesh/surface, the (θ_0, θ_1, J(θ_0, θ_1)) points are plotted 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.

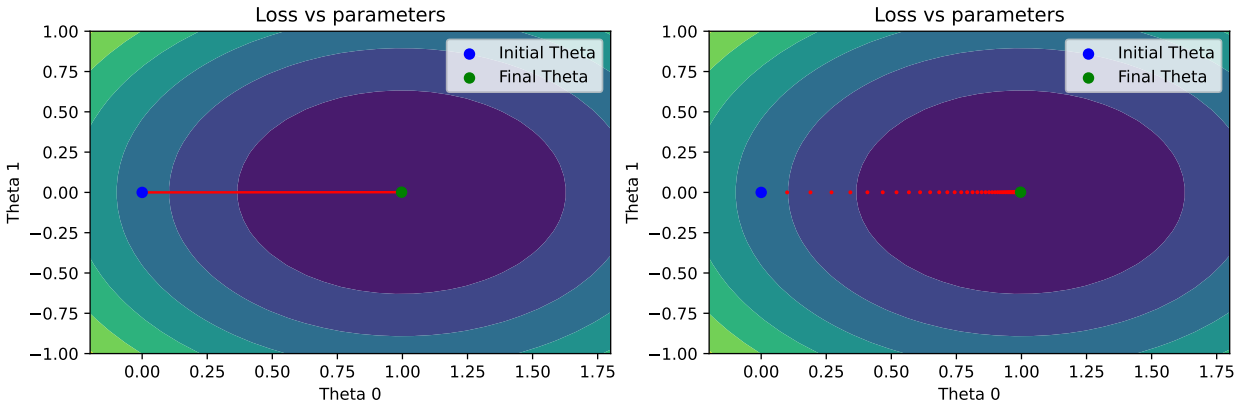


The plotting was animated to show the learning stage in each iteration. Some intermediate plots are shown above. The complete animation can be seen by running the submitted script for this question.

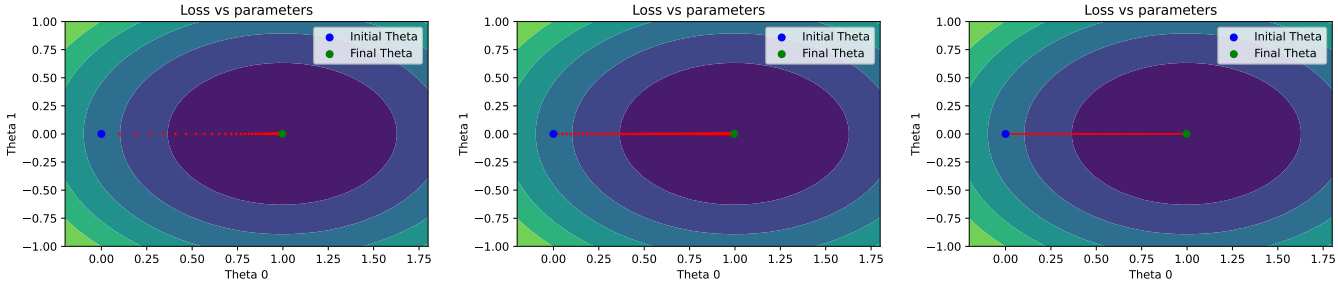
(d) Loss vs Parameters Contour plot



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 and the plot showing each iteration at a time gap of 0.2s are shown below.



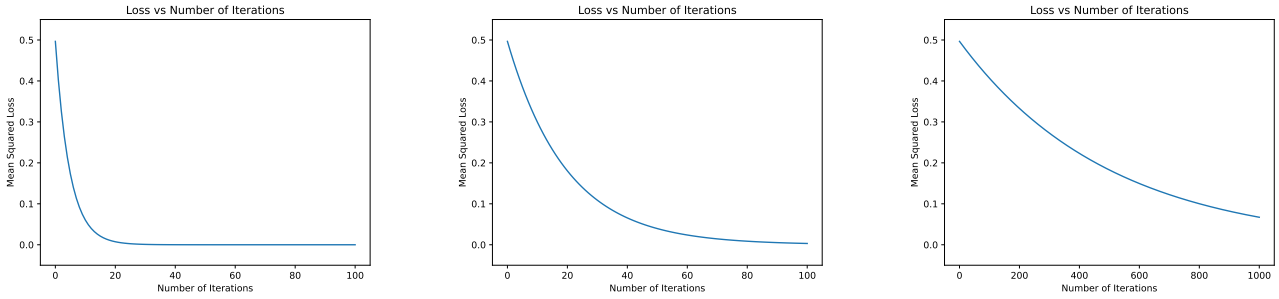
(e) Contours for different step sizes



(a) Step Size 0.1 (b) Step Size 0.025 (c) Step Size 0.001

Figure 1: Contour plots for different step sizes

As can be seen in Fig 1, 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.



(a) Step Size 0.1 (b) Step Size 0.025 (c) Step Size 0.001

Figure 2: Rate of Convergence with different step sizes

As can be seen from Fig 2, With a step size of 0.1, the loss has almost converged in 100 iterations, while with a step size of 0.025, it has not. With a step size of 0.001, the loss is not close to convergence, even after 1000 iterations. Allowing them to converge completely, the three experiments required 121, 474 and 10354 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
1	$AD(i, 1000) \leq 10^{-5}$	$\theta = [\ 2.95972864\ 1.00316614\ 2.02654619\ ]^T$
100	$AD(i, 1000) \leq 10^{-5}$	$\theta = [\ 2.99892128\ 0.99732906\ 1.99844308\ ]^T$
10000	$AD(i, 100) \leq 10^{-5}$	$\theta = [\ 2.99780613\ 1.00060674\ 2.00062604\ ]^T$
1000000	$\lvert J(\theta_i) - J(\theta_{i-1})\rvert \leq 10^{-5}$	$\theta = [\ 2.99433888\ 1.00124883\ 2.00039978\ ]^T$

Here the average difference $AD(i,k)$ is the difference between the averages of the last k and 2k $\rightarrow$ k parameter update steps before the current i^{th} step. It can be calculated as:

$$AD(i,k) = \left|\frac{1}{k}\sum_{j=i-k}^{i} J(\theta_j) - \frac{1}{k}\sum_{j=i-2k}^{i-k} J(\theta_j)\right| = \frac{1}{k}\sum_{j=i-k}^{i} \left|J(\theta_j) - J(\theta_{j-k})\right| \tag{3}$$

The convergence criteria were chosen keeping in mind both time taken to complete training and the nature of convergence. Because SGD in nature has some fluctuations, Using similar stopping criteria as in the last

question was not sufficient to ensure convergence. To make stopping robust over fluctuations, average losses of last k training steps were taken, and the difference between two subsequent such averages was used to decide stopping. A uniform threshold of 10^{-5} was used for all batch sizes, but the value of k was varied. Value of k taken were 1000, 1000, 100 and 1 respectively for the four batch sizes. In the case of batch size 1000000, the problem reduces to batch gradient descent, and also, with a k of 1, the stopping criteria reduces to one used in question 1.

(c) As could be seen from the table above, the parameters learned with all of these batch sizes, although very close, are not exactly identical. The speed of convergence and the number of iterations, however are significantly different. The observations obtained are reported below:

Batch Size	$\lvert\theta - \theta_{original}\rvert$	Time to converge	No. of training steps	No. of iterations
1	0.04834	69.552 sec	37654	1
100	0.00327	69.150 sec	37372	4
10000	0.00236	155.261 sec	35915	360
1000000	0.00581	5746.377 sec	24537	24537

Comments:

- As can be seen in the table above, all four experiments took about the same number of training steps to converge. However, with a larger batch size, each training step takes longer since larger amount of data is processed. Thus, as the batch size increases, the time to converge increases significantly.
- As the batch size increases, the number of iterations increases. An iteration is basically a complete pass of the data, and so, larger batch sizes need more number of passes to properly learn the data. In the extreme case of batch size 1, the entire data is not even covered once, and this may lead to some errors if the data is of higher variance.

The losses on the training set and the test set are summarised below:

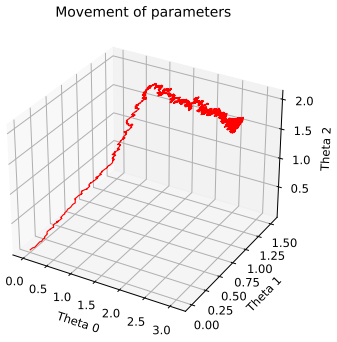
Experiment	Training loss	Test loss	Error difference
Original Hypothesis	1.0002404	0.9829469	-
Batch size 1	1.0031556	1.0174413	0.0344944
Batch size 100	1.0002869	0.9838140	0.0008671
Batch size 10000	1.0002383	0.9828770	0.0000699
Batch size 1000000	1.0002401	0.9829457	0.0000012

Comments:

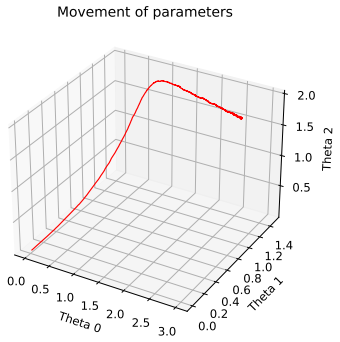
- The training losses as well as the test losses are very close to 1. Upon closer inspection, it may be seen that the differences between true and predicted labels on the training set using the original hypothesis are actually the sampled values of ϵ. Then, by its formulation, the MSE loss of the model should be simply half of the variance of ϵ. Since ϵ values were sampled from a distribution of variance 2, the losses being close to 1 are to be expected.
- The errors on training and test sets for each of the batch size are also very close to that of the original hypothesis, all lying within a range of $L_{orig} \pm 0.003$ for training set and $L_{orig} \pm 0.03$ for the test set.
- As the batch size increases, the difference of test error with the original hypothesis decreases. This indicates that the model is able to learn better with larger batch sizes. This is expected since as observed before, with larger batch size, the model passes through the entire data larger number of times.
- Thus, with a larger batch size, the performance of the model improves, but the time taken to train also increases. Between batch size 10000 and 1000000, the difference in errors is $\approx 6 \times 10^{-5}$, while the training time increases almost 40 times. Thus, considering the tradeoff between time and performance, an intermediate batch size can be most ideal. In our case, a batch size of 10000 or 100 may be the best algorithm choice.

(d) Movement of θ

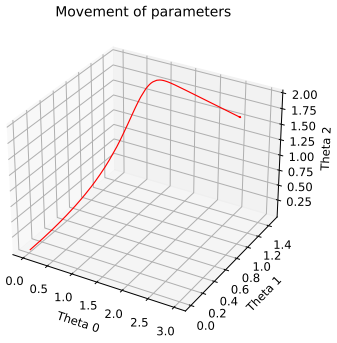
The movement of θ in 3D space as the algorithm learns for all the four batch sizes (1, 100, 10000, 1000000) considered can be seen in Fig 3.



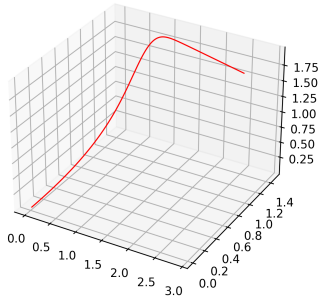
(a) Batch Size 1



(b) Batch Size 100



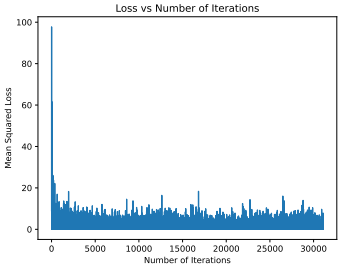
(c) Batch Size 10000



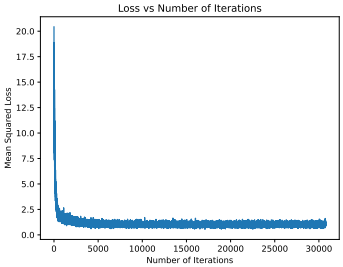
(d) Batch Size 1000000

Figure 3: Movement of θ for different batch sizes.

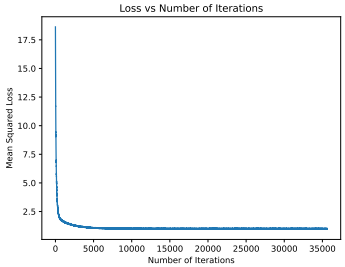
- The movement for the case of batch size 1, as can be seen in Fig 3a is a bit jagged, in a more zig-zag fashion. As the batch size increases, the curve becomes smoother. With batch size 100, some amount of zig-zag nature can still be seen, while batch size 10000 and 1000000 look completely smooth.
- This makes intuitive sense because with smaller batch sizes, the losses computed would be highly variable as different batches of small number of samples will have much more variation than larger numbers of samples. This is because the latter will involve averaging over more samples, leading to more uniformity. This 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 batch size 1, the loss changes after ever sample, and thus, has the most variation. Very similar trends can be observed in Fig 4.



(a) Batch Size 1



(b) Batch Size 100



(c) Batch Size 10000

Figure 4: Losses vs iterations for different batch sizes.

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 4. The vectorised version of this equation, that is actually implemented in the code, is in eq 5.

$$H = \sum_{i=1}^{m} x^{(i)}(x^{(i)})^T h_\theta(x^{(i)})(1 - h_\theta(x^{(i)})) \tag{4}$$

$$\mathbf{H} = \mathbf{X}^T \; Diag(h_\theta(\mathbf{X}) \times (1 - h_\theta(\mathbf{X})) \; \mathbf{X} \tag{5}$$

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

Other than the core training, another modification was that the mean squared loss was replaced with the log-likelihood loss.

The final parameters obtained are:

- Learned Parameters: $\theta = \begin{bmatrix} 0.4165822 \\ 2.40661151 \\ -2.61234944 \end{bmatrix}$

(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.

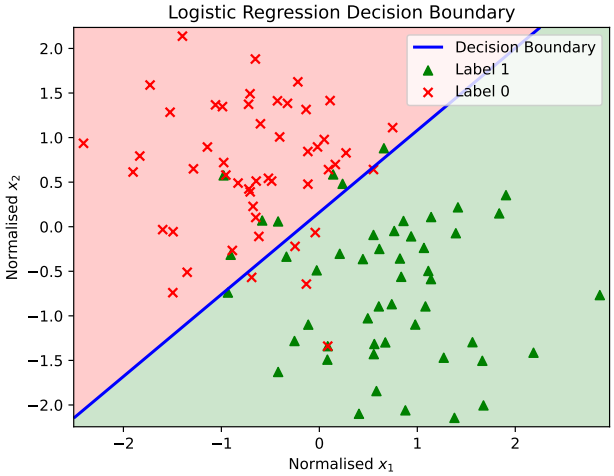


Figure 5: Decision Boundary and Normalised data for Logistic Regression

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      self._phi = np.sum(indicator_1) / m
6      self._mu_0 = np.sum(indicator_0 * self.X, axis=0) / np.sum(indicator_0)
7      self._mu_1 = np.sum(indicator_1 * self.X, axis=0) / np.sum(indicator_1)
8
9      # with the assumption that sig_0 = sig_1
10     self._mu = indicator_0 * self._mu_0 + indicator_1 * self._mu_1
11     self._sig = np.dot(np.transpose(self.X - self._mu), self.X - self._mu) / m
12
13     # General case when sig_0 is not the same as sig_1
14     self._sig_0 = np.dot(np.transpose(self.X - self._mu_0), indicator_0 * (self.X - self._mu_0)) /
        np.sum(indicator_0)
15     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 6 discussed in class. This step is common for both part (a) and part (d).

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

The lines 12 and 13 calculate the covariance matrix Σ. Again, this follows the same equations in eq 7 discussed in class. Note that the closed form solution of $\mu_{y^{(i)}}$ is equivalent to the definition discussed in class.

$$\mu_{y^{(i)}} = 1\{y^{(i)} = 0\}\mu_0 + 1\{y^{(i)} = 1\}\mu_1, \qquad \Sigma = \frac{\sum_{i=1}^{m}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{m} \tag{7}$$

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

(b) Here, the points with green triangles are data points that have label 'Canada' while those with red crosses have label 'Alaska'.

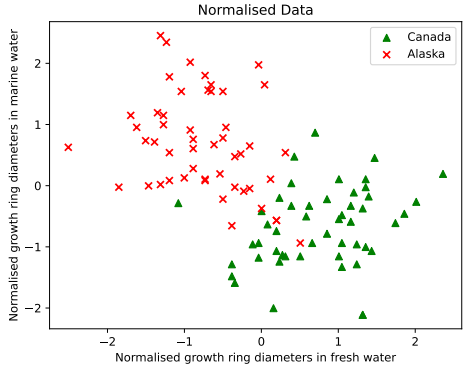


Figure 6: Normalised data for Gaussian Discriminant Analysis

(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 in eq 8. This is implemented directly in the code.

$$-(\mu_1 - \mu_0)^T \Sigma^{-1} x + \frac{1}{2}(\mu_1^T \Sigma^{-1} \mu_1 - \mu_0^T \Sigma^{-1} \mu_0) + log(\frac{1-\phi}{\phi}) = 0 \tag{8}$$

```
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.

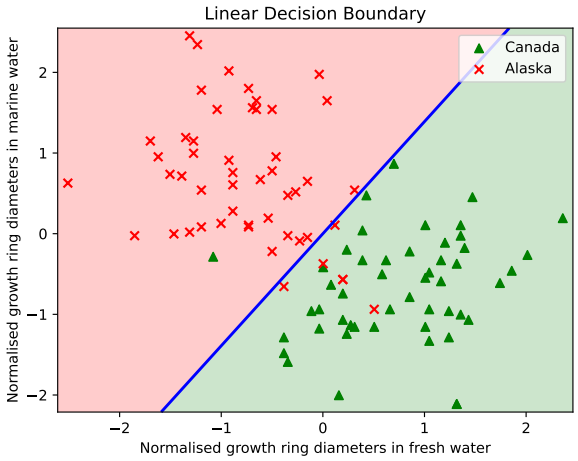


Figure 7: Linear Decision boundary for Gaussian Discriminant Analysis

(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 6. The values of Σ_0 and Σ_1 are computed using eq 9.

$$\Sigma_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}, \quad \Sigma_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \tag{9}$$

where $\mu_{y^{(i)}}$ is the same as that used in eq 7, i.e. the one given in eq 10.

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

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: $\begin{bmatrix} 0.38158978 & -0.15486516 \\ -0.15486516 & 0.64773717 \end{bmatrix}$
- Σ_1: $\begin{bmatrix} 0.47747117 & 0.1099206 \\ 0.1099206 & 0.41355441 \end{bmatrix}$

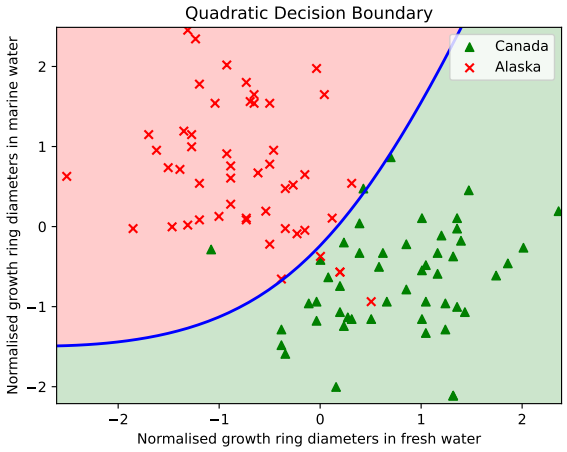
(e) Following the similar process as used in part (c), we get the quadratic decision boundary in eq 11.

$$\frac{1}{2}(x^T(\Sigma_1^{-1} - \Sigma_0^{-1})x) - (\mu_1^T\Sigma_1^{-1} - \mu_0^T\Sigma_0^{-1})x + \frac{1}{2}(\mu_1^T\Sigma_1^{-1}\mu_1 - \mu_0^T\Sigma_0^{-1}\mu_0) + log(\frac{1-\phi}{\phi}\frac{|\Sigma_1|^{\frac{1}{2}}}{|\Sigma_0|^{\frac{1}{2}}}) = 0 \tag{11}$$

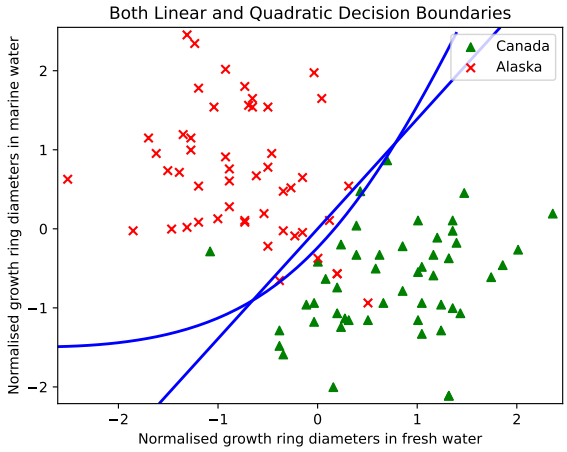
This is implemented directly in the code, as can be seen in the code snippet below.

```
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 8: Quadratic Decision Boundary obtained by Gaussian Discriminant Analysis

(f) Comments on the Decision boundaries obtained

- In Fig 8b, 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 7 and Fig 8a, 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 8a, $|\Sigma_0| < |\Sigma_1|$, 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 11.