INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 1

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL774

Prof. Parag Singla

Due September 12, 2021

Contents

Chapter 1

Linear Regression using Batch Gradient Descent

1.1 Objective

To find θ that *minimizes* the loss function ($J(\theta)$) for the given training set.

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

1.2 Method

First, I normalized input data with respect to feature x_1 (Acidity of wine):

$$\mu_1 = 8.062, \sigma_1 = 1.72312391$$

Then, I determined the vectorized expressions for $J(\theta)$ and $\nabla_\theta J(\theta)$:

$$J(\theta) = \frac{1}{2m}(X\theta - Y)^T(X\theta - Y)$$

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

where $X \in \mathbb{R}^{m\times(n+1)}$ and $Y \in \mathbb{R}^{m\times 1}$ are as defined in class.

After this, I set the learning rate, $\boldsymbol{\eta = 0.3}$ and convergence threshold, $\boldsymbol{\epsilon = 10^{-18}}$, and applied **Batch Gradient Descent** to find θ that minimizes $J(\theta)$.

1.3 Algorithm

Algorithm 1 Batch Gradient Descent for finding θ that minimizes $J(\theta)$

$\theta \leftarrow (0, 0, 0)^T$
while `true` **do**
  $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta)$
  `// Stopping Criteria: stop when change in loss value is less than` ϵ
  if $\delta(J) < \epsilon$ **then**
    break
  end if
end while

1.4 Result

Number of iterations = 58
Optimum value of parameter obtained, $\boldsymbol{\theta = (0.9966201, 0.0013402)^T}$.
Hence, $J(\theta)$ is successfully optimised. **Linear Regression Model trained:**

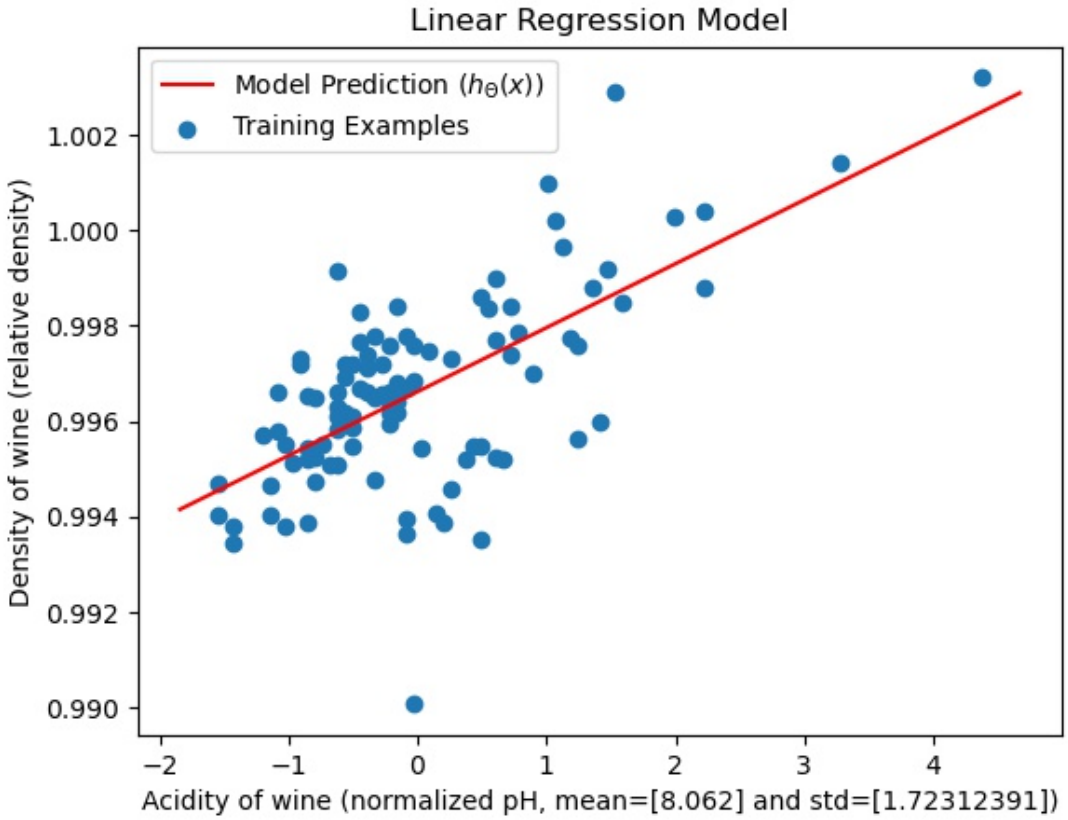

Figure 1.1: Training Data and Linear Regression Model Prediction

Note: I also verified the value of optimal parameter using the relation $\boldsymbol{\theta = (X^T X)^{-1} X^T Y}$.

Note: While experimenting with learning rate, I observed that it is possible to converge to the optimal parameter in just **two** iterations. This corresponds to $\boldsymbol{\eta = 1}$ **and** $\boldsymbol{\epsilon = 10^{-4}}$.
However, in such a case, the movement of gradient descent across iterations cannot be analyzed. On the following page, I present the *mesh* and *contour* plot of $J(\theta)$ in 3D and 2D respectively, on which I trace the movement of Gradient Descent Algorithm.

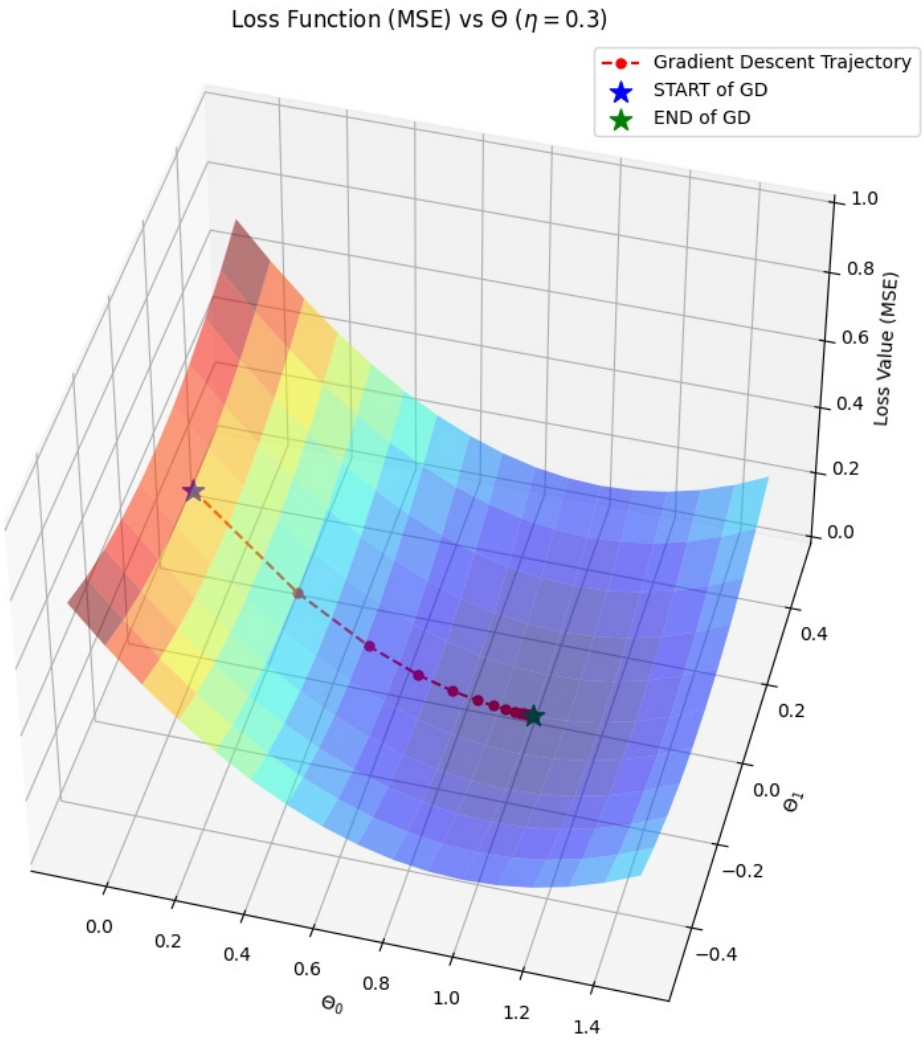

Figure 1.2: Plot of $J(\theta)$ vs θ (3D) and Trajectory of Gradient Descent

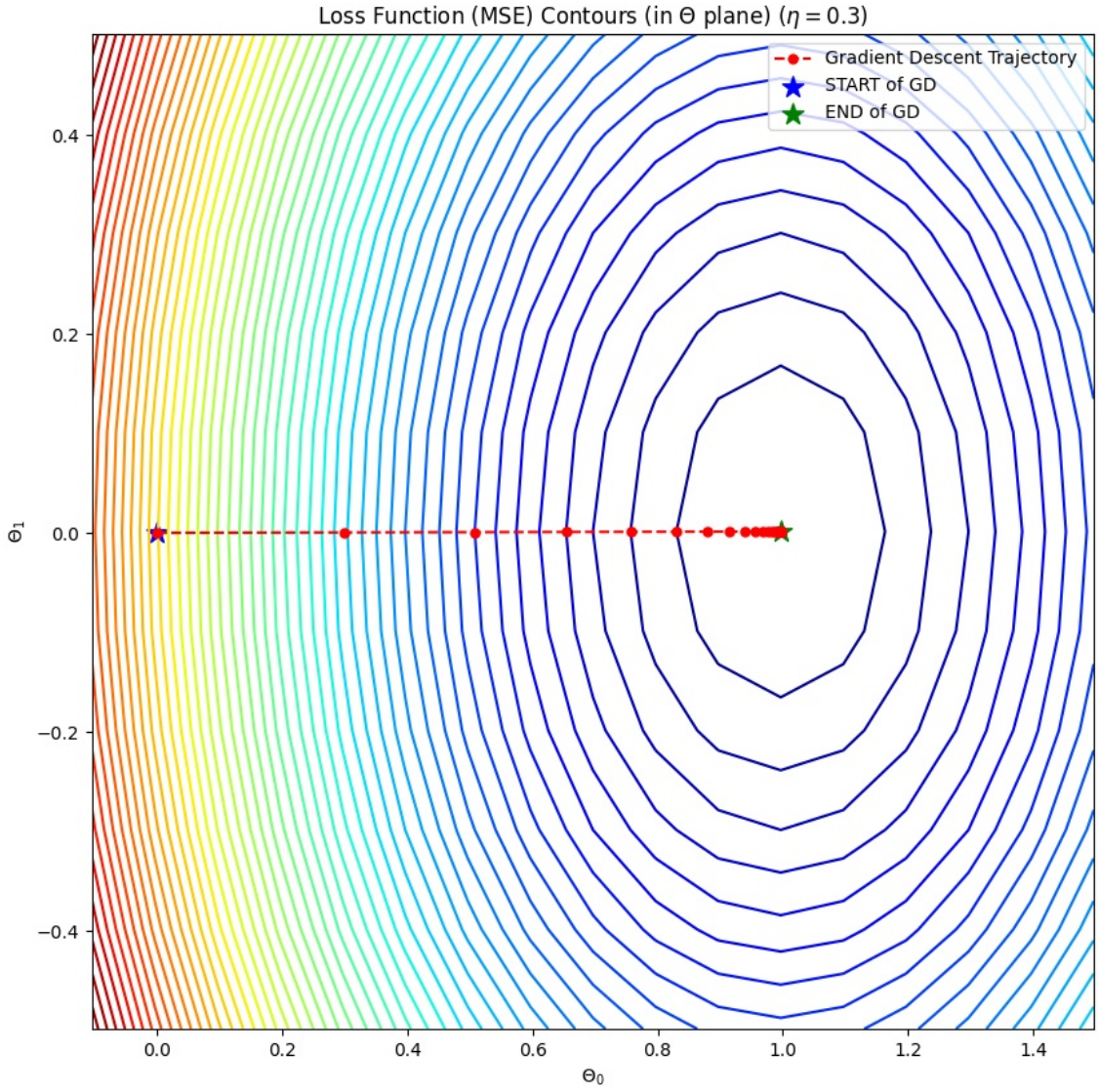

Figure 1.3: Contours of $J(\theta)$ in θ-plane ($\eta = 0.3$)(2D) and Trajectory of Gradient Descent

1.5 Effect of lowering the learning rate (η)

I lowered the learning rate from $\eta = 0.3$ to $\eta = 0.1, 0.025$ and 0.001, set $\epsilon = 10^{-20}$, and plotted the contour of $J(\theta)$ in θ-plane and animated the movement of Gradient Descent Algorithm. I present the final plots in this section:

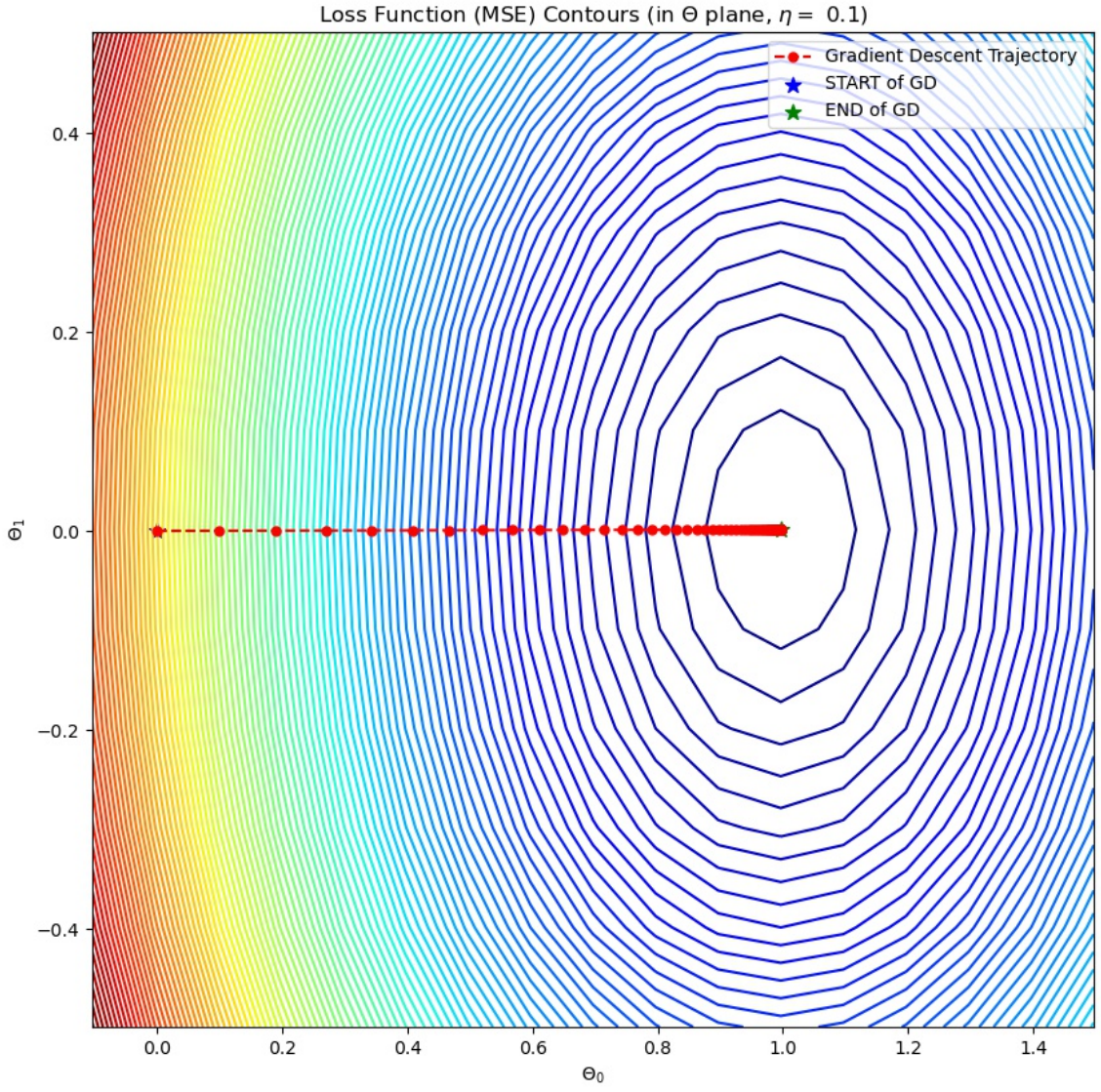

Figure 1.4: Contours of $J(\theta)$ in θ-plane ($\eta = 0.1$)(2D) and Trajectory of Gradient Descent

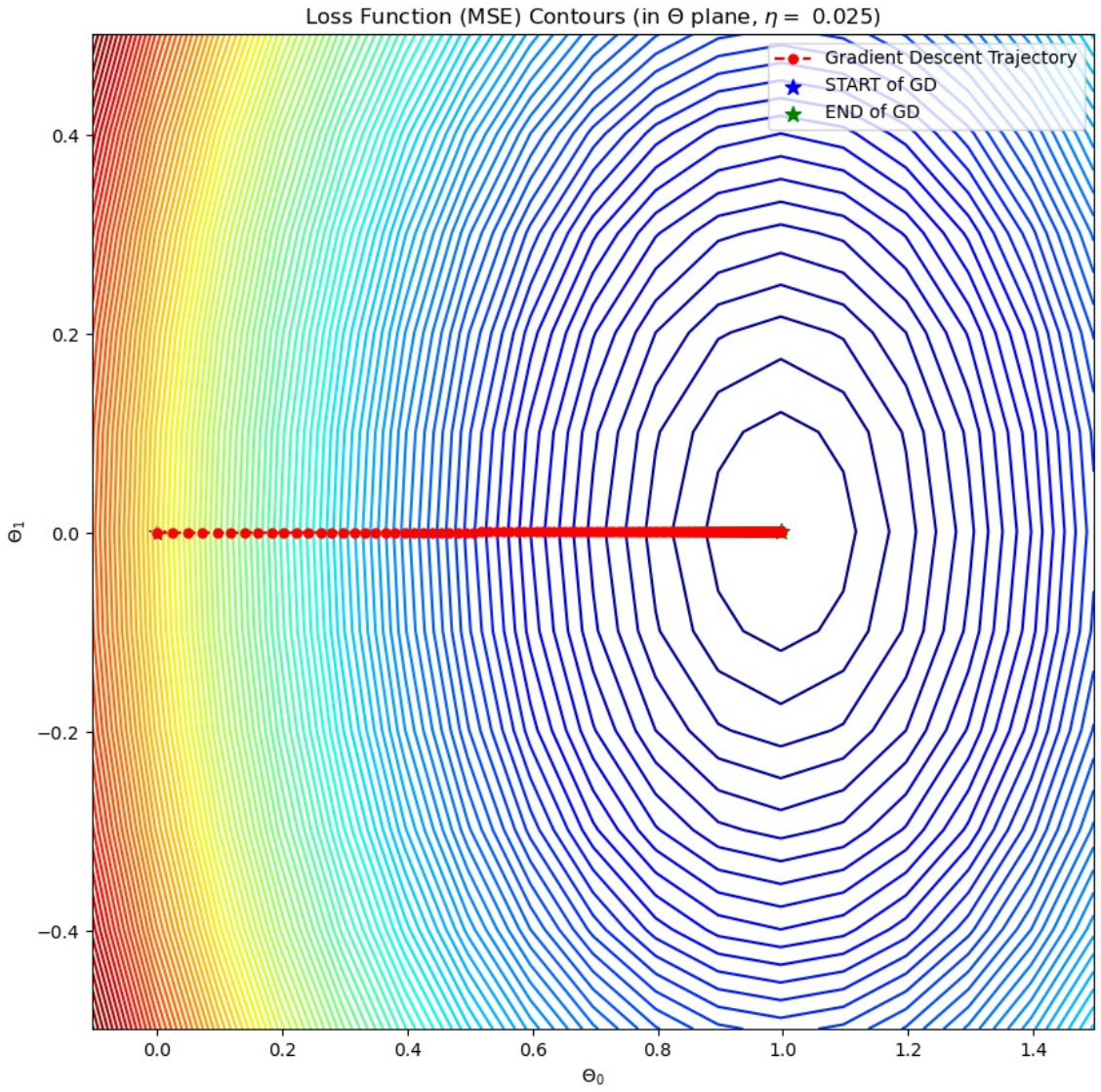

Figure 1.5: Contours of $J(\theta)$ in θ-plane ($\eta = 0.025$)(2D) and Trajectory of Gradient Descent

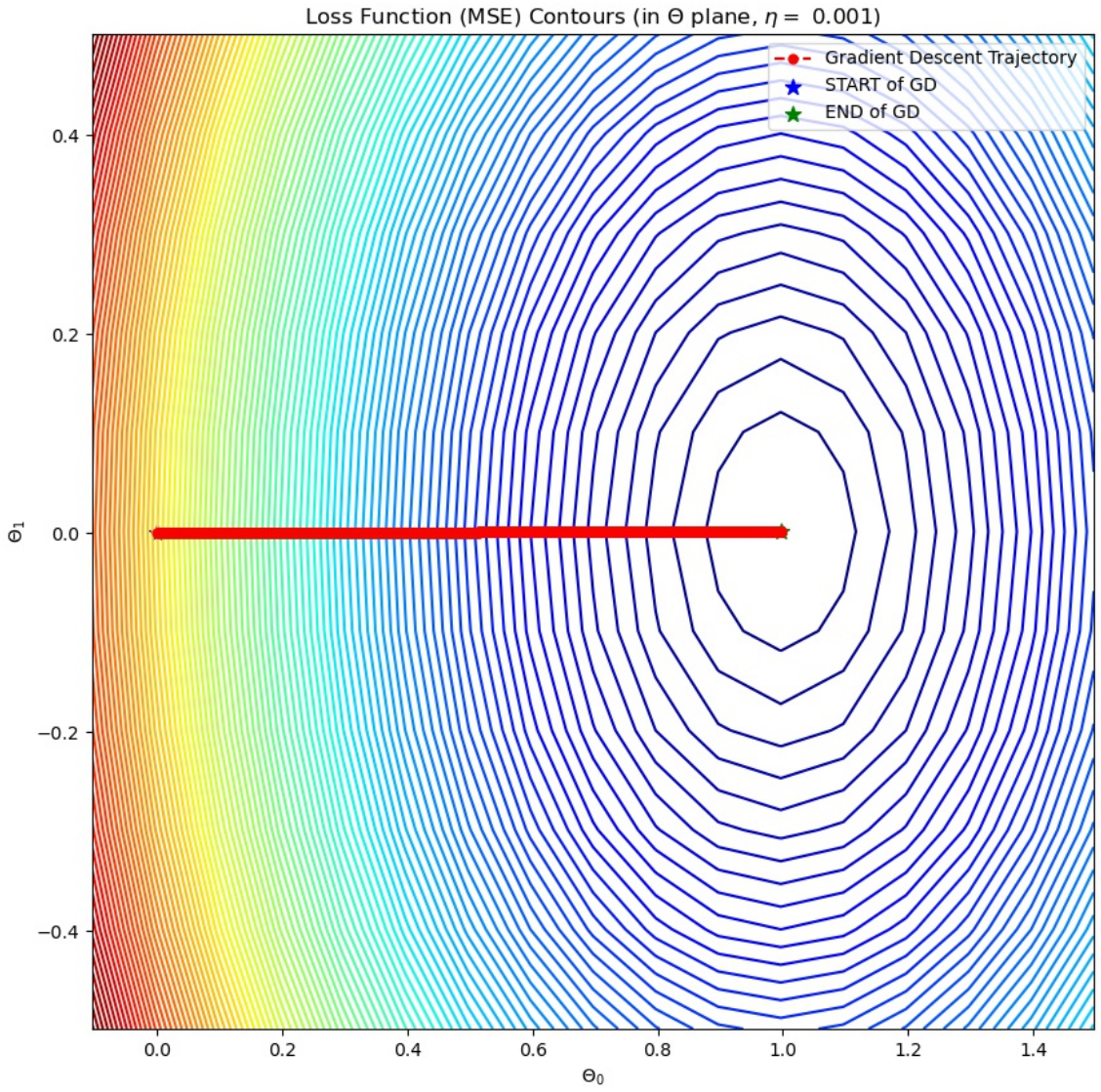

Figure 1.6: Contours of $J(\theta)$ in θ-plane ($\eta = 0.001$)(2D) and Trajectory of Gradient Descent

First thing I observe is the number of iterations that the Gradient Descent Algorithm takes. For $\eta = 0.1$, the algorithm iterates 208 times and for $\eta = 0.025$, it iterates 829 times, whereas for $\eta = 0.001$, it iterates a total of 18972 times! This means that as I decrease the learning rate, the parameter θ moves more slowly towards the optimal value. This thing can also be observed from the contour plots (Figure 1.4, 1.5 and 1.6). When $\eta = 0.1$, the change in θ (equivalently J) value in the beginning can be observed without strain, as the step-size is big enough. Gradually, the step-size decreases as the parameter moves towards optimum value, due to corresponding decrease in gradient. However, when $\eta = 0.025$, the step-size is even smaller in the beginning. Still, the change in θ (or J) value can be observed during the first few iterations. The smallest step-size results when I make $\eta = 0.001$. In this case, the update in parameter (equivalently J) per-iteration is so small, that it is very straining (nearly impossible) to observe the change in θ (or J) in its corresponding plot (Figure 1.6). Consequently, the algorithm takes a large number of iterations to converge to the optimal value.

Hence, I conclude that as the learning rate decreases, the size of the parameter update also decreases, which in turn makes gradient descent algorithm slow. Therefore, for small η, the rate of convergence of gradient descent algorithm can be very slow and it may take a large number of iterations before parameter achieves the optimal value. In many scenarios, this slow rate can make the algorithm impractical and hence, it is always good to choose a higher learning rate, if the convergence of algorithm is not disturbed.

Note: Animations for Figure 1.2 to Figure 1.6 can be found here.

Chapter 2

Linear Regression using Stochastic Gradient Descent

2.1 Objective

To sample data and find θ that *minimizes* $J(\theta)$ for that data using Stochastic Gradient Descent.

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

2.2 Method

First, I sampled training data (x_1, x_2, y) using the relation:

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

where, $\theta = (3, 1, 2)^T$, $x_1 \sim \mathcal{N}(3, 4)$, $x_2 \sim \mathcal{N}(-1, 4)$ and $\epsilon \sim \mathcal{N}(0, 2)$.

Then, I determined the vectorized expressions for $J_b(\theta)$ and $\nabla_\theta J_b(\theta)$:

$$J_b(\theta) = \frac{1}{2r}(X_b\theta - Y_b)^T(X_b\theta - Y_b)$$

$$\nabla_\theta J_b(\theta) = \frac{1}{r}X_b^T(X_b\theta - Y_b)$$

where $\boldsymbol{r}$ is the batch size (mini-batch), and $X_b \in \mathbb{R}^{r\times(n+1)}$ and $Y_b \in \mathbb{R}^{r\times 1}$ represent the training data corresponding to the b^{th} batch ($1 \leq b \leq \frac{m}{r}$).

After this, I set the learning rate, $\boldsymbol{\eta = 0.001}$ and convergence threshold, $\boldsymbol{\epsilon = 10^{-5}}$, and used **Stochastic Gradient Descent** to find θ that minimizes $J(\theta)$. Convergence criteria: I exit from the loop when change in average mini-batch loss over epoch is less than ϵ.

Next, I tested the learnt parameter for different batch sizes ($r \in \{1, 100, 10000, 1000000\}$) on test data (coming from same distribution) and compared the results with each other as well as with the original hypothesis ($\theta = (3, 1, 2)^T$).

2.3 Algorithm

Algorithm 2 Stochastic Gradient Descent for finding θ that minimizes $J(\theta)$

$\theta \leftarrow (0,0,0)^T$
`Randomly Shuffle training set (X, Y)`
while `true` **do**
  for $1 \leq b \leq \frac{m}{r}$ **do**
    $\theta \leftarrow \theta - \eta \nabla_\theta J_b(\theta)$
  end for
  `//` J_{avg} `is the average mini-batch loss over epoch`
  if $\delta(J_{avg}) < \epsilon$ **then**
    break
  end if
end while

2.4 Result

Original Parameter, $\boldsymbol{\theta_0 = (3,1,2)^T}$
Test Error using Original Parameter, $\boldsymbol{J_0 = 0.9829469215}$ **units**
Optimal Parameter for given set $= \boldsymbol{(3.00097807, 0.999769, 1.99958665)^T}$ (determined analytically)

Batch Size, r = 1
Parameter Learnt, $\boldsymbol{\theta_a = (3.02349861, 1.04243743, 2.0508789)^T}$
Number of iterations in SGD, $\boldsymbol{n_a = 3}$ **epochs** $=$ **3000000 updates**
Time taken to achieve convergence, $\boldsymbol{t_a = 29.55}$ **seconds**
Test Error using θ_a, $\boldsymbol{J_a = 1.1928419523152285}$ **units**
Difference: $J_a - J_0 = 0.20989503081522842$ units

Batch Size, r = 100
Parameter Learnt, $\boldsymbol{\theta_b = (3.00055492, 0.99582437, 2.00426414)^T}$
Number of iterations in SGD, $\boldsymbol{n_b = 4}$ **epochs** $=$ **40000 updates**
Time taken to achieve convergence, $\boldsymbol{t_b = 1.58}$ **seconds**
Test Error using θ_a, $\boldsymbol{J_b = 0.9844533472045975}$**units**
Difference: $J_b - J_0 = 0.0015064257045974694$ units

Batch Size, r = 10000
Parameter Learnt, $\boldsymbol{\theta_c = (2.96668297, 1.00718054, 1.9971089)^T}$
Number of iterations in SGD, $\boldsymbol{n_c = 162}$ **epochs** $=$ **16200 updates**
Time taken to achieve convergence, $\boldsymbol{t_c = 5.47}$ **seconds**
Test Error using θ_a, $\boldsymbol{J_c = 0.986349624909432}$ **units**
Difference: $J_c - J_0 = 0.0034027034094319397$ units

Batch Size, r = 1000000
Parameter Learnt, $\boldsymbol{\theta_d = (2.64188294, 1.07820566, 1.9735064)^T}$
Number of iterations in SGD, $\boldsymbol{n_d = 7544}$ **epochs** $\boldsymbol{= 7544}$ **updates**
Time taken to achieve convergence, $\boldsymbol{t_d = 234.24}$ **seconds**
Test Error using θ_a, $\boldsymbol{J_d = 1.3543873195741343}$ **units**
Difference: $J_d - J_0 = 0.3714403980741342$ units

2.5 Analysis

As is evident from the data mentioned before, the learning algorithm does not converge to the same parameter for varying values of **r** (keeping η and ϵ fixed). If I compare these parameters with the original value (θ_0), I observe that θ_a, θ_b and θ_c are in good agreement with θ_0 with component-wise absolute difference being less than 0.05. However, θ_d does not compare well with θ_0, especially in the intercept term, where the absolute difference is around 0.4.

Another way in which we can compare these parameters and decide which model is better, is to test the parameters on a *test* set. Such testing was also performed by my code and the results have been mentioned in the *Result* section. I observe that all the learnt parameters give a higher test error than the original parameter. This means that θ_0 fits the test data better than θ_a, θ_b, θ_c and θ_d, which is expected as the data itself has been generated using θ_0. Now, among the learnt models, θ_b has the least error on the test data. Also, component-wise absolute difference in θ_b and θ_0 is at most 0.005, which an order of 10 smaller than the corresponding differences for other parameters. Hence, it is safe to say that θ_b is giving us the best hypothesis in terms of prediction accuracy among all the learnt models (batch size = 100).

Now, I analyse the relative speed of convergence and number of iterations that SGD took in each of the four cases. First thing I observe is that as I increase the batch size, the number of *epochs* increases (3 - 4 - 162 - 7544). Multiplying the number of epochs with $\frac{m}{r}$ gives me the total updates done. Therefore, in this case, the number of updates done to parameter decreases with batch size (3×10^6 - 4×10^4 - 16200 - 7544). Next thing I observe is that even though the number of updates decreases with batch size, the execution time does not. Initially the execution time drops from 29.55 seconds to 1.58 seconds on changing r from 1 to 100 (due to high oscillations and function overhead in the case when $r = 1$) Thereafter, the execution time increases with increase in batch size- 5.47 seconds ($r = 10000$) and 234.24 seconds ($r = 1000000$) (due to increase in number of epochs). Thus, on comparing the execution times, I can say that SGD convergence is fastest when batch size is 100 and the slowest when batch size is 1000000 (entire set). Convergence with batch size 10000 is faster than with batch size 1. Hence, the convergence is slow in both of the extreme cases and is fastest somewhere between them (like $r = 100$).

Hence, model b (θ_b) is both most accurate and fastest among all other learnt models, and is therefore, a clear victor.

Note: In all my analysis, I have taken the learning rate and convergence threshold to be the same, so that I can analyze the performance of models w.r.t to batch size (r). Also, contrary to what was expected, θ_d is the worst estimate out of all the models, even though we considered the entire batch while doing an update. This is because the convergence threshold was fixed and the algorithm was only able to do 7544 updates, as compared to 16200, 40000 and 3000000 in other cases. Hence, θ_d was not sufficiently updated before convergence, but still model d took the longest time to converge. This is because it has the largest number of *epochs* and each epoch has the same complexity ($O(mn)$) irrespective of batch size. This is the true essence of SGD, to be able to make large number of updates to the parameter in the same time and reach a *near* optimal solution quickly. Hence, when data sets are very large, batch gradient descent can be a huge bottleneck, and thus, inefficient.

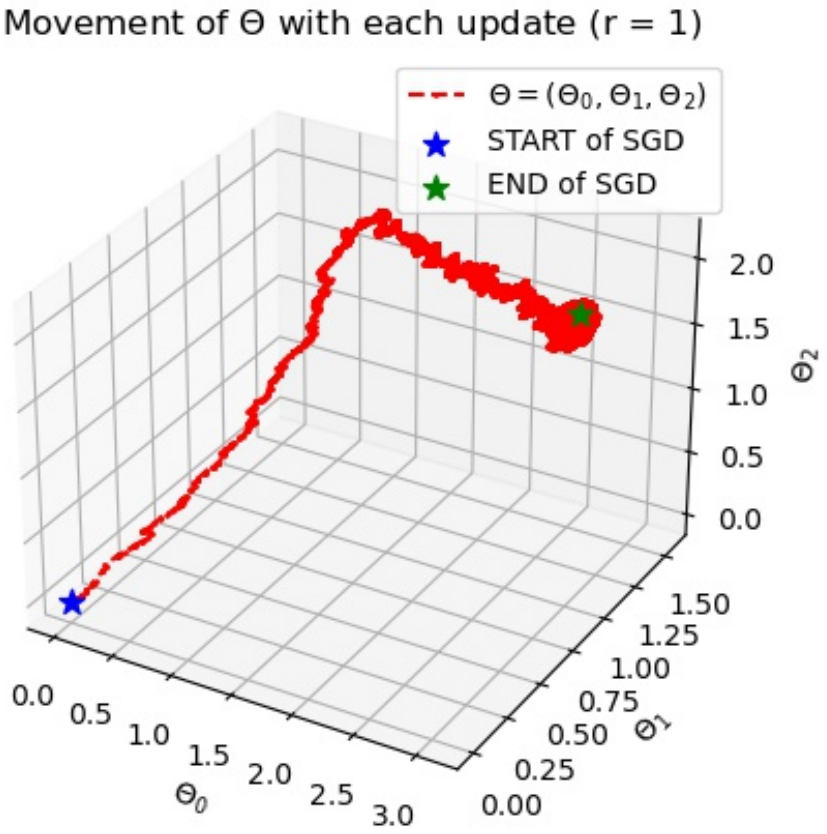

Figure 2.1: Parameter plot (SGD, r = 1)

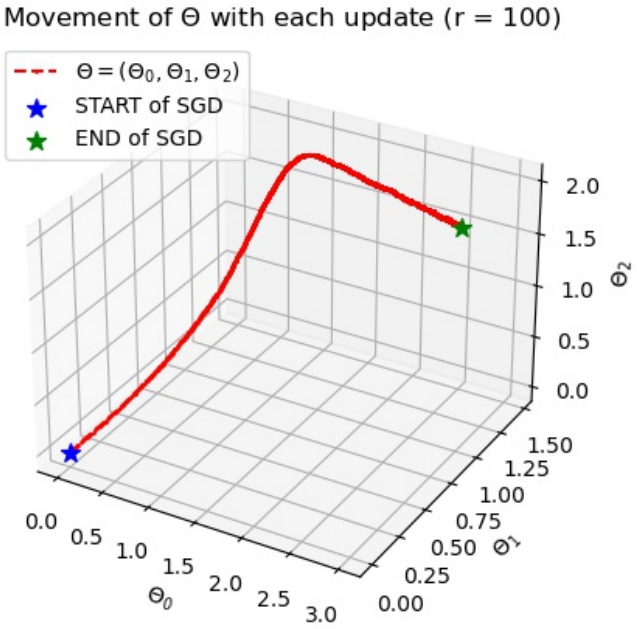

Figure 2.2: Parameter plot (SGD, r = 100)

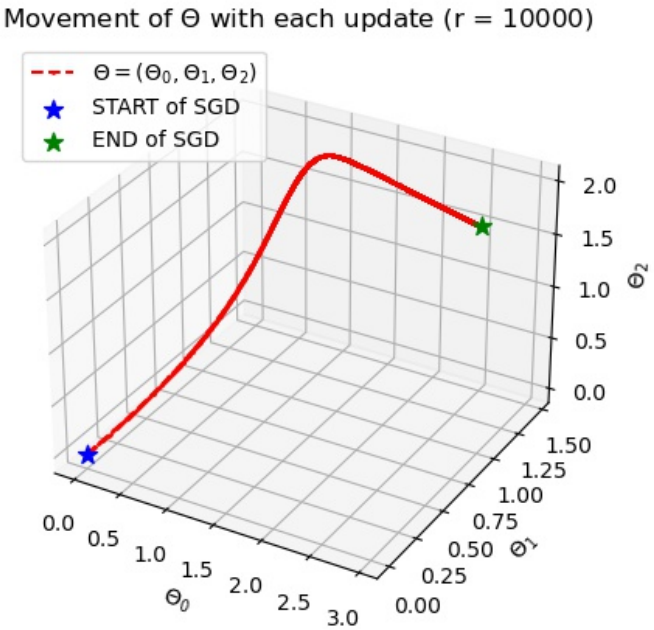

Figure 2.3: Parameter plot (SGD, r = 10000)

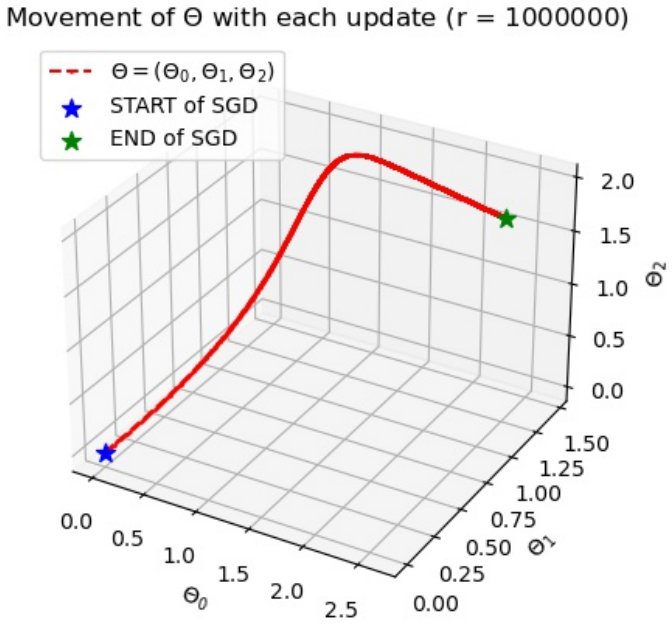

Figure 2.4: Parameter plot (SGD, r = 1000000)

From these plots, I can observe that when the batch size is very small ($r = 1$), there is a significant noise or oscillatory movement of the parameter throughout the plot (Figure 2.1). This oscillation becomes larger when the gradient starts decreasing, and the parameter reaches closer to the optimal value. This can be explained as when the gradient becomes smaller, the step size also decreases proportionally. Hence, the parameter only gets updated by a small amount which fails to generalize the decrease in loss value for other examples present in the training set. In other words, since the batch size is 1, each example is trying to move the parameter in the direction which minimizes the loss for that example only. Hence, when gradient is small, there is more of nit-picking going on due to example individualism rather than a smooth decrease to optimal value. This gives rise to oscillations in the vicinity of the optimal value.

Hence, it is logical to observe that as the batch size increases, these oscillations eventually fade out. This can be observed in Figure 2.2 to Figure 2.4. Some minute oscillations exist when batch size is 100, but these are significantly less than those present in Figure 2.1 (mostly because we have got rid of example individualism). The oscillations are almost non-existent in Figure 2.3, and completely die out in Figure 2.4, where we take the entire batch for doing an update.

Also, the overall movement of the parameter in all the plots is the same (converge to similar values) which can be explained as we are taking the contribution of all the training examples but in a round-robin fashion, hence, we can *expect* the gradient descent to converge to *near* optimal values in all the cases. Hence, the plots make intuitive sense. For animations, see here.

Chapter 3

Logistic Regression using Newton's Method

3.1 Objective

To find θ that *maximizes* the log-likelihood ($LL(\theta)$) for the given training set.

$$LL(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)})), \text{where } h_\theta(x^{(i)}) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

3.2 Method

Finding θ that maximizes log-likelihood is equivalent to finding θ that *minimizes* the negative log-likelihood ($NLL(\theta)$).

$$NLL(\theta) = -\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))$$

So, I decided to solve the given problem by optimising $NLL(\theta)$.
First, I normalized input data with respect to both the features, x_1 and x_2:

$$\mu_1 = 4.618717, \sigma_1 = 1.318928$$

$$\mu_2 = 4.522868, \sigma_2 = 1.387166$$

Then, I determined the vectorized expressions for $NLL(\theta)$, $\nabla_\theta NLL(\theta)$ and $\nabla^2_\theta NLL(\theta)$:

$$NLL(\theta) = -(Y^T \log h(X\theta) + (1 - Y)^T \log(1 - h(X\theta))$$

$$\nabla_\theta NLL(\theta) = X^T(h(X\theta) - Y)$$

$$\nabla^2_\theta NLL(\theta) = X^T D X$$

where log, h and $-$ represent element-wise logarithm, sigmoid and subtraction functions.
$X \in \mathbb{R}^{m \times (n+1)}$ and $Y \in \mathbb{R}^{m \times 1}$ are as defined in class. $D \in \mathbb{R}^{m \times m}$ is a diagonal matrix given by:

$$D = diag(h(\theta^T x^{(1)})(1 - h(\theta^T x^{(1)})), .., h(\theta^T x^{(m)})(1 - h(\theta^T x^{(m)})))$$

After this, I set the convergence threshold, $\boldsymbol{\epsilon = 10^{-5}}$, and applied **Newton's** method to find θ that minimizes $NLL(\theta)$.

3.3 Algorithm

Algorithm 3 Newton's Method for finding θ that minimizes $NLL(\theta)$

1: $\theta \leftarrow (0, 0, 0)^T$
2: **while** `true` **do**
3: $\quad \theta \leftarrow \theta - (\nabla^2_\theta NLL(\theta))^{-1} \nabla_\theta NLL(\theta)$
4: $\quad$ **if** $\delta(NLL) < \epsilon$ **then**
5: $\quad\quad$ **break**
6: $\quad$ **end if**
7: **end while**

3.4 Result

Optimum value of parameter obtained, $\boldsymbol{\theta = (0.40125316, 2.5885477, -2.72558849)^T}$. Hence, $LL(\theta)$ is successfully optimised. **Logistic Regression Model trained:**

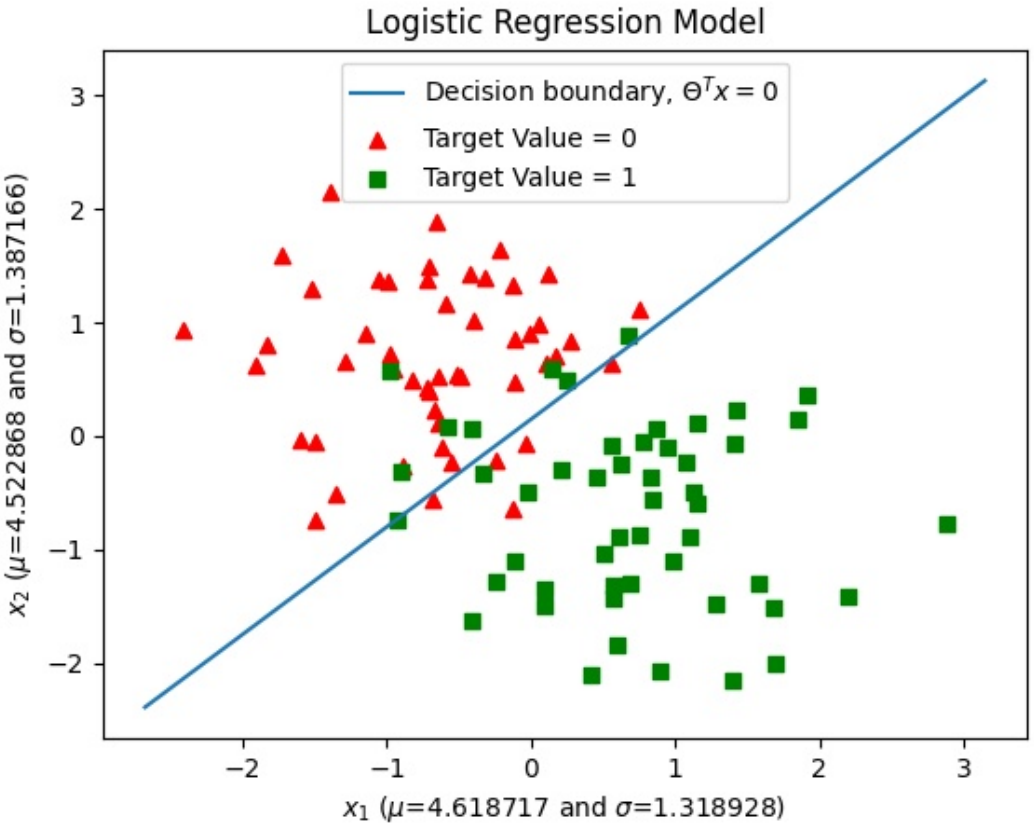

Figure 3.1: Training Data and Logistic Regression Decision Boundary

Chapter 4

Gaussian Discriminant Analysis

4.1 Objective

To implement **GDA** for separating out salmons from Alaska (1) and Canada (0).

4.2 Method

GDA is characterized by parameter, $\Theta = (\phi, \mu_1, \Sigma_1, \mu_0, \Sigma_0)$ (or $(\phi, \mu_1, \mu_0, \Sigma)$).
First, I normalized input data with respect to both the features, x_1 and x_2:

$$\mu_1 = 117.92, \sigma_1 = 25.87071$$

$$\mu_2 = 398.14, \sigma_2 = 46.00826$$

Then, I used formulae derived in class to determine Θ for given data, $X \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{m \times 1}$. I learnt two models. In one, I assumed the covariance matrices of both the classes to be equal, i.e. $\Sigma_1 = \Sigma_0 = \Sigma$. In other, I relaxed this assumption and assumed Σ_1 and Σ_2 to be independent of each other.

After learning all model parameters for both the models, I plotted the decision boundary for each model. These boundaries are further described and analyzed in *Analysis* section.

4.3 Result

Parameter values learnt from training data: (ϕ, μ_1 and μ_0 are the same for both models)

$$\phi = 0.5$$

$$\mu_1 = \begin{bmatrix} -0.75529433 \\ 0.68509431 \end{bmatrix} \in \mathbb{R}^{2 \times 1}, \mu_0 = \begin{bmatrix} 0.75529433 \\ -0.68509431 \end{bmatrix} \in \mathbb{R}^{2 \times 1}$$

$$\Sigma_1 = \begin{bmatrix} 0.38158978 & -0.15486516 \\ -0.15486516 & 0.64773717 \end{bmatrix} \in \mathbb{R}^{2 \times 2}, \Sigma_0 = \begin{bmatrix} 0.47747117 & 0.1099206 \\ 0.1099206 & 0.41355441 \end{bmatrix} \in \mathbb{R}^{2 \times 2}$$

$$\Sigma = \begin{bmatrix} 0.42953048 & -0.02247228 \\ -0.02247228 & 0.53064579 \end{bmatrix} \in \mathbb{R}^{2 \times 2}$$

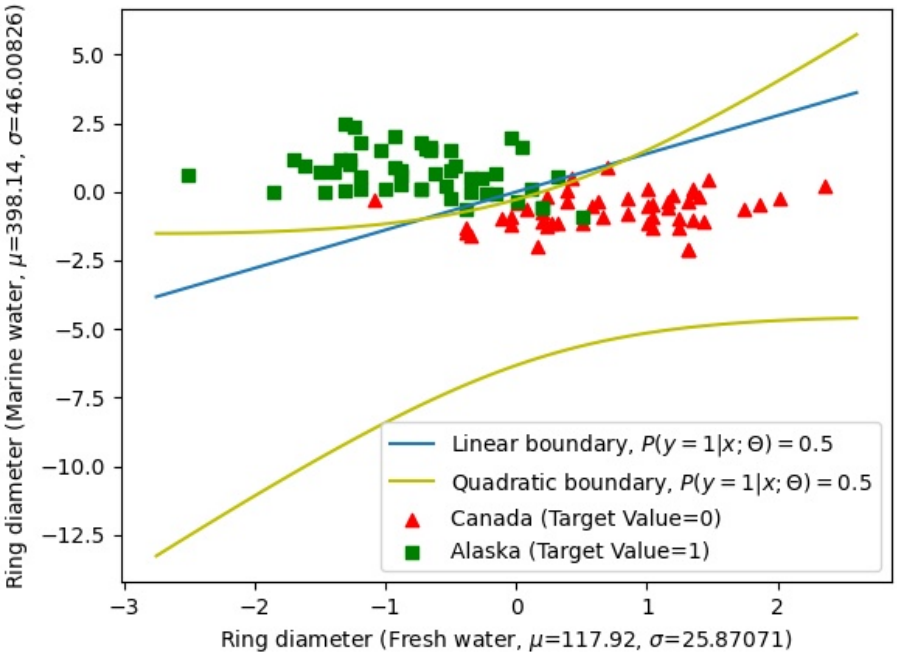

Figure 4.1: Training Data and GDA boundaries

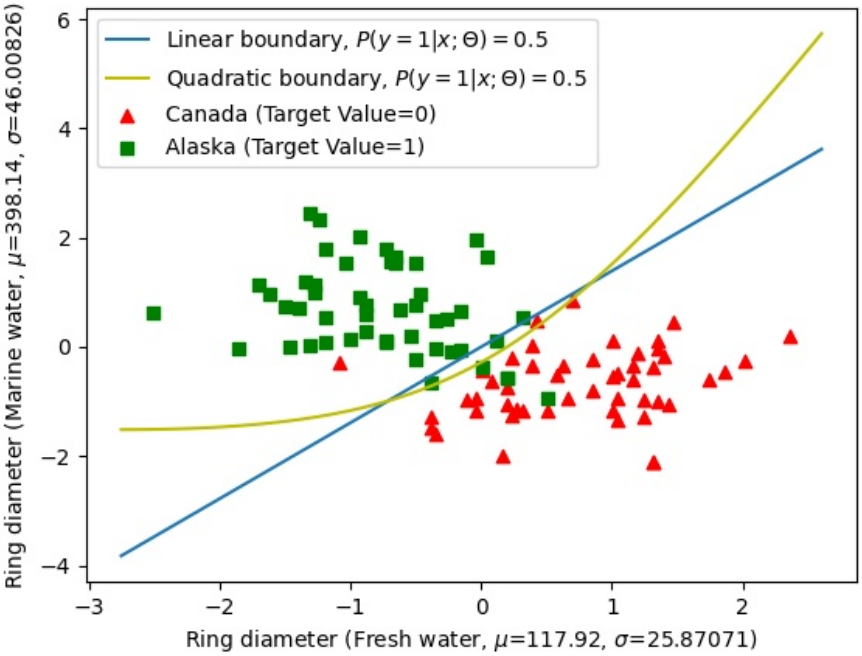

Figure 4.2: Training Data and GDA boundaries (simplified quadratic boundary)

4.4 Analysis

In my model above, first I assumed that the covariance matrix characterizing the features for both the classes is the same (Σ). Under this assumption, I was able to successfully learn a classification model in which the decision boundary was a straight line in the feature plane (or x_1-x_2 plane). The equation of this line can be determined in terms of the parameters ϕ, μ_1, μ_0 and Σ, by setting $P(y = 1|x; \Theta) = 0.5$ in GDA:

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

where $x = (x_1, x_2)^T$ is the feature vector, or a point in the feature plane. This curve is coloured *blue* in Figure 4.1 and Figure 4.2.

After learning a linear decision boundary, I relaxed my assumption of same covariance matrix, and allowed both of the target classes to have their own covariance matrix (Σ_1 and Σ_0). Under this assumption, I was able to successfully learn a classification model in which the decision boundary was *quadratic* in terms of x_1 and x_2. The equation of this curve can be determined in terms of parameters $\phi, \mu_1, \mu_0, \Sigma_1$ and Σ_0, by setting $P(y = 1|x; \Theta) = 0.5$ in GDA:

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

where again, $x = (x_1, x_2)^T$ represents a point in the feature plane. This curve is coloured *yellow* in Figure 4.1 and Figure 4.2. Also, from Figure 4.1, it can be observed that the *quadratic* form is that of a *general* **hyperbola** in (x_1, x_2)-plane, which is separating one class (outer region of hyperbola) from another (between the two branches of hyperbola).

If we analyse the two decision boundaries further, we see that one of the two branches of the quadratic boundary is actually playing no role in classifying the data, assuming that the test data also comes from the same distribution as that of the training set. Hence, we can omit that branch and get a better view of the boundaries and its interaction with the data. This simplification has been done in Figure 4.2. Further in Figure 4.2, we observe that both the decision boundaries are having almost identical fit when it comes to training data. The quadratic model can be seen as fitting the salmons from Alaska slightly better than the linear one (Figure 4.2), however, I do not have enough data to determine if the quadratic model is actually a better model for this problem, or it is just over-fitting the data due to availability of more parameters. Hence, it cannot be seen whether relaxing the assumption of equal covariance is leading to a better model or not. In fact, there is no observable or clear advantage of using the *quadratic* model over the *linear* one, as the linear model is also doing fairly well in classifying salmons (Figure 4.2).

To pick one out of them, we can analyze the utilities of both models further, by running these models on some unseen data (test data) and noting down their prediction accuracy.

Chapter 5

Program Execution

5.1 q1.py

Type in the command in the terminal: `python q1.py <filename1> <filename2>`
Here, <filename1> and <filename2> are placeholders for name of files in the same directory containing input data and output data respectively (CSV).

5.2 q2.py

Type in the command in the terminal: `python q2.py <testfile>`
Here, <testfile> is the placeholder for name of file in the same directory containing test data (both X and Y) (CSV).

5.3 q3.py

Type in the command in the terminal: `python q3.py <filename1> <filename2>`
Here, <filename1> and <filename2> are placeholders for name of files in the same directory containing input data and output data respectively (CSV).

5.4 q4.py

Type in the command in the terminal: `python q4.py <filename1> <filename2>`
Here, <filename1> and <filename2> are placeholders for name of files in the same directory containing input data and output data respectively (DAT).