<u>CSE : 343 Machine Learning</u>

Assignment 1

Arka Sarkar , 2018222

<u>Question 1</u>

Best Value of K , for K-Fold cross validation is **10** :

Analysis
We have only 4176 examples of the abalone dataset so if we choose a small k then we would end up with a small training set which is bad for our model training.
I ran multiple occurrences of K-fold cross validation using different values of k ranging from [2,10]. The K which gives us the best avg validation loss is selected for the further analysis.

```
The average loss with k =  2  is   train loss =  2.612144577361343  validation loss =  2.6135570905179266
The average loss with k =  3  is   train loss =  2.6116837273367985  validation loss =  2.6156052463190758
The average loss with k =  4  is   train loss =  2.611696186273858  validation loss =  2.6129862969808997
The average loss with k =  5  is   train loss =  2.6113593850326624  validation loss =  2.6129494802267663
The average loss with k =  6  is   train loss =  2.6121954304520307  validation loss =  2.6126436070823256
The average loss with k =  7  is   train loss =  2.612428760806144  validation loss =  2.609077512675159
The average loss with k =  8  is   train loss =  2.6126324169953046  validation loss =  2.61069115768441
The average loss with k =  9  is   train loss =  2.6126643989316816  validation loss =  2.610974447617996
The average loss with k =  10  is   train loss =  2.612464223152074  validation loss =  2.6094793832843246
```

Figure 1 : Abalone Dataset K Fold

```
The average loss with k =  2  is   train loss =  1.5141941588477252  validation loss =  1.5142285401432414
The average loss with k =  3  is   train loss =  1.5118390563271598  validation loss =  1.496362409969662
The average loss with k =  4  is   train loss =  1.5156101249895972  validation loss =  1.5076899028048625
The average loss with k =  5  is   train loss =  1.5142353697811832  validation loss =  1.4852774581327772
The average loss with k =  6  is   train loss =  1.514411538615004  validation loss =  1.4712364826041622
The average loss with k =  7  is   train loss =  1.5154532501396092  validation loss =  1.475578027842183
The average loss with k =  8  is   train loss =  1.515841989250715  validation loss =  1.4755463684221732
The average loss with k =  9  is   train loss =  1.5161058802250535  validation loss =  1.4808126886756798
The average loss with k =  10  is   train loss =  1.5158274512681829  validation loss =  1.457188220192617
```

Figure 2 : Video Game Dataset K Fold

The chart above is the is for the **abalone dataset** and **video game dataset** and and the validation loss for k = 10 is the minimum, though it is evident that all values of k give us more or less the same validation loss but I preferred k = 10 also on the fact that in the lectures it was told that k = 10 is a very common choice (Lecture 5).

Preprocessing strategy

1. **Abalone Dataset**
   We had 9 columns in the dataset from which we had to perform Linear regression to predict 'rings' from the features = ['sex','length', 'diameter', 'height', 'whole_weight', 'shucked_weight', 'viscera_weight', 'shell_weight'].
   The feature 'sex' had values 'M', 'F' and 'I' values and I replaced these by 1,2 and 3 numeric values.

The training columns were normalised i.e. subtract mean and divide by standard deviation for a particular column.

$$z_i = \frac{x_i - \bar{x}}{s}$$

- xi is a data point (x1, x2…xn).
  - $\bar{x}$ is the **sample mean.**
- s is the sample standard deviation.

At Last the data was shuffled.

2. **Video Game Dataset**

The Video Game dataset has 16 features. However, for the purpose of this assignment, you need to consider only the following features

a. Input Variables- Critic_Score, User_Score

b. Output Variable - Global_Sales

Out[23]:

	Global_Sales	Critic_Score	User_Score
0	82.53	76.0	8
1	40.24	NaN	NaN
2	35.52	82.0	8.3
3	32.77	80.0	8
4	31.37	NaN	NaN

In [24]:
```
df.isna().sum()
```

Out[24]:
```
Global_Sales        0
Critic_Score     8582
User_Score       6704
dtype: int64
```

In [27]:
```
df["Critic_Score"].value_counts()
```

Out[27]:
```
70.0    256
71.0    254
75.0    245
78.0    240
73.0    238
       ...
20.0      3
17.0      1
22.0      1
13.0      1
21.0      1
Name: Critic_Score, Length: 82, dtype: int64
```

In [28]:
```
df["User_Score"].value_counts()
```

Out[28]:
```
tbd    2425
7.8     324
8       290
8.2     282
8.3     254
       ...
0.2       2
9.6       2
0.5       2
0         1
9.7       1
Name: User_Score, Length: 96, dtype: int64
```

The dataset columns (Critic_Score and User) had multiple **na,** and **string** values values

Critic_Score had 8585 null values
User_Score had 6784 null values and 2425 'tbd' values.
These cannot be trained in my model so I filled all these values with the mean of that particular column.
The next step was to normalise the training columns :

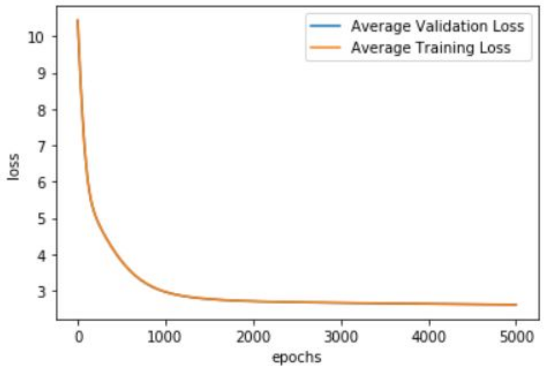
$$z_i = \frac{x_i - \bar{x}}{s}$$

- xi is a data point (x1, x2…xn).
  - $\bar{x}$ is the **sample mean.**
- s is the sample standard deviation.

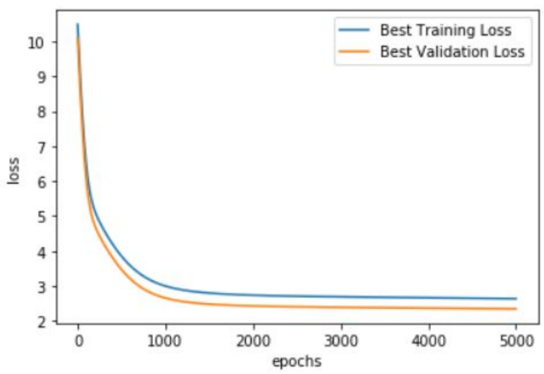
The Final Step was shuffling of the dataset.

Part (a) : Training Loss v.s. Iterations and Validation Loss v.s Iterations

1. **Abalone Dataset**
    a. For RMSE Loss

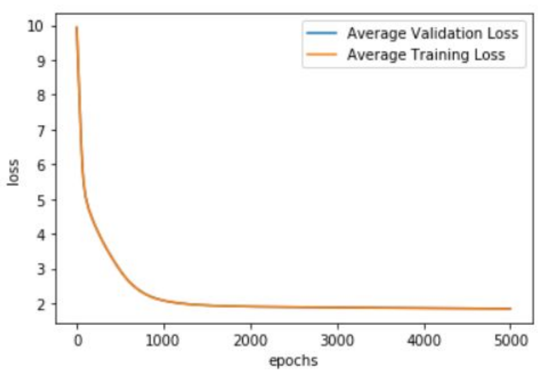


← Average RMSE Loss over all the K-Folds Plot

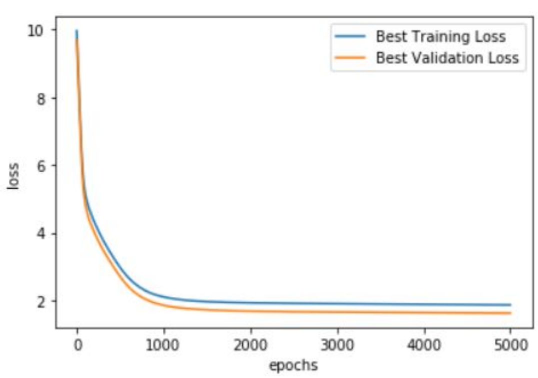


← Best Fold RMSE Loss Plot

    b. For MAE Loss

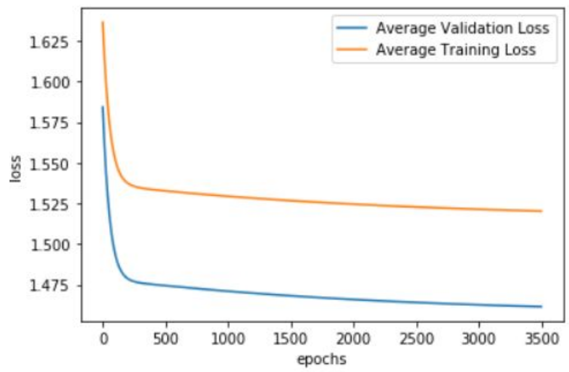


← Average MAE Loss over all K-folds Plot

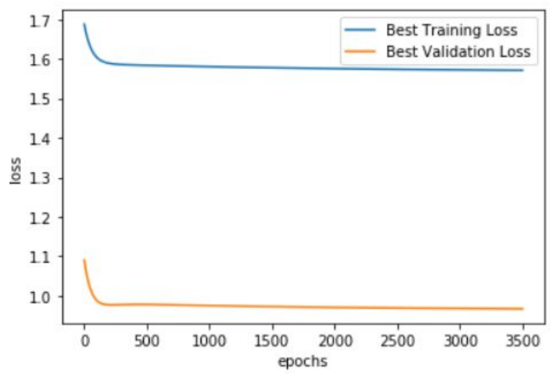


← Best Fold MAE Loss plot

2. **Video Game Dataset**
    a. For RMSE Loss

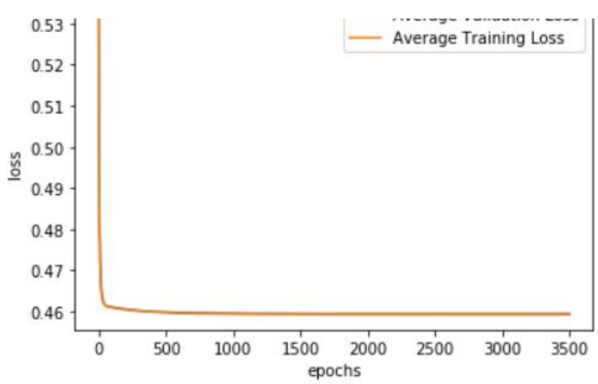


← Average RMSE Loss over all the K-Folds Plot

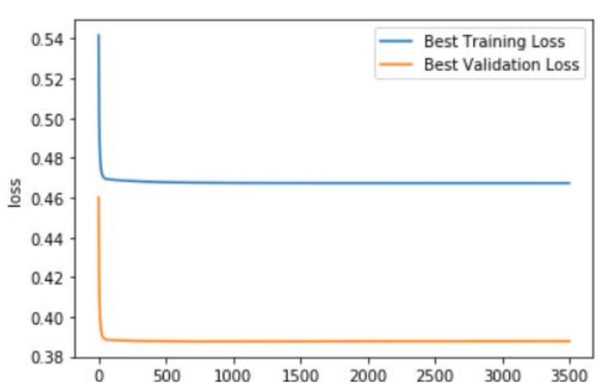


← Best Fold RMSE Loss Plot

    b. For MAE Loss



← Average MAE Loss over all K-folds Plot



← Best Fold MAE Loss plot

Part (b) : Best RMSE and MAE value achieved

1. **Video Game Dataset**
   The RMSE and MAE Values achieved across all Folds is :

```
Stats for LR MAE Loss
For CV number  0 the train loss =  0.4570705294598559  and the val loss =  0.47981963845536096
For CV number  1 the train loss =  0.46201269656039756  and the val loss =  0.43487858426953513
For CV number  2 the train loss =  0.45084234718446087  and the val loss =  0.5358408110548142
For CV number  3 the train loss =  0.459565379625231  and the val loss =  0.457260603111768
For CV number  4 the train loss =  0.4660512328311721  and the val loss =  0.3988677620028403
For CV number  5 the train loss =  0.4567942765219069  and the val loss =  0.48222056223054266
For CV number  6 the train loss =  0.4672929233511366  and the val loss =  0.3877298917992819
For CV number  7 the train loss =  0.45395515922545576  and the val loss =  0.50777767972726
For CV number  8 the train loss =  0.4578089983274051  and the val loss =  0.47283351046913547
For CV number  9 the train loss =  0.4616995771966845  and the val loss =  0.4374756103735225
Stats for LR RMSE Loss
For CV number  0 the train loss =  1.5334913798246819  and the val loss =  1.4088400332255244
For CV number  1 the train loss =  1.5543221012307915  and the val loss =  1.182684831327262
For CV number  2 the train loss =  1.3801048496440143  and the val loss =  2.4491568742256598
For CV number  3 the train loss =  1.5103473647453438  and the val loss =  1.6147350888898901
For CV number  4 the train loss =  1.5702087908648306  and the val loss =  0.9753923015042985
For CV number  5 the train loss =  1.5335452219715833  and the val loss =  1.4086423075494674
For CV number  6 the train loss =  1.5706973153144799  and the val loss =  0.9670855180351106
For CV number  7 the train loss =  1.470115329543413  and the val loss =  1.9200136081203985
For CV number  8 the train loss =  1.5391285455320443  and the val loss =  1.3508711746561313
For CV number  9 the train loss =  1.5404005426481786  and the val loss =  1.336652603031023
```

   The Lowest MAE and RMSE validation Loss was achieved in Fold 6 (0 based indexing).

   Train loss = 0.4672929233511366 and the val loss = 0.3877298917992819 ← MAE
   Train loss = 1.5706973153144799 and the val loss = 0.9670855180351106 ← RMSE

2. **Abalone Dataset**
   The RMSE and MAE Values achieved across all Folds is :

```
Stats for LR MAE Loss
For CV number  0 the train loss =  1.8477167574636362  and the val loss =  1.736364767583854
For CV number  1 the train loss =  1.8603356449328192  and the val loss =  1.6194507487277865
For CV number  2 the train loss =  1.8348234643751553  and the val loss =  1.833392456765055
For CV number  3 the train loss =  1.8284658820470958  and the val loss =  1.9229871995573304
For CV number  4 the train loss =  1.8187509108463003  and the val loss =  2.0159666237000478
For CV number  5 the train loss =  1.8501645427054196  and the val loss =  1.7105550493875565
For CV number  6 the train loss =  1.839663468885081  and the val loss =  1.8114407182725596
For CV number  7 the train loss =  1.8415982087180307  and the val loss =  1.7966541926342663
For CV number  8 the train loss =  1.8268650868936411  and the val loss =  1.9456485515974185
For CV number  9 the train loss =  1.8203173673361615  and the val loss =  1.9974029352388079
Stats for LR RMSE Loss
For CV number  0 the train loss =  2.6325630850486883  and the val loss =  2.431246718787734
For CV number  1 the train loss =  2.6383659819341414  and the val loss =  2.3483853409168307
For CV number  2 the train loss =  2.6114265896296733  and the val loss =  2.5959558922252817
For CV number  3 the train loss =  2.590002195577359  and the val loss =  2.837949365605624
For CV number  4 the train loss =  2.589701594375981  and the val loss =  2.8245829363898394
For CV number  5 the train loss =  2.6336885344725354  and the val loss =  2.405192489365533
For CV number  6 the train loss =  2.618450697393412  and the val loss =  2.5549080837163882
For CV number  7 the train loss =  2.6123277744478206  and the val loss =  2.614015259776798
For CV number  8 the train loss =  2.6073213527313435  and the val loss =  2.681478174155562
For CV number  9 the train loss =  2.5907944259097846  and the val loss =  2.801079571903656
```

The Lowest MAE and RMSE validation Loss was achieved in Fold 1 (0 based indexing).

Train loss = 1.8603356449328192 and the val loss = 1.6194507487277865 ← MAE
Train loss = 2.6383659819341414 and the val loss = 2.3483853409168307 ← RMSE

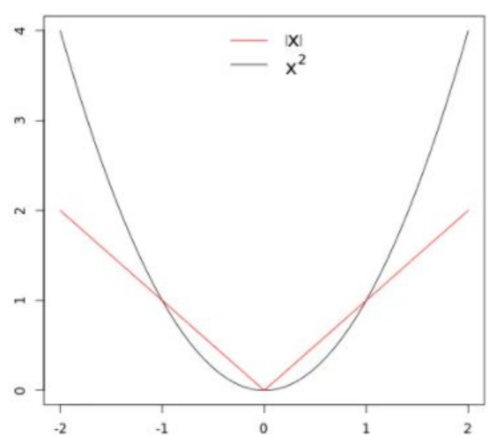
Part (c) : Which Loss leads to better performance.

RMSE and MAE follow the rule that RMSE >=MAE, but the 2 losses have some differences. SInce in RMSE the quantities are squared before they are averaged then it implies that RMSE should be much useful when we require to penalise large errors.
Now, SInce **Abalone dataset** is a regression problem to predict rings of the range 0-29, large errors need not be penalised to that extent therefore **MAE** is the better choice for this dataset. Even evident from part (a) and (b) the MAE received was **1.6194507487277865** which is very reasonable.
The **Video Game Dataset** is a regression problem to predict global sales of the range 0.01- 100, which is a very huge range and the data have a lot of outliers so we need to avoid getting large penalties thus **MAE** should be much better in this dataset.

Part (d)

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2} \qquad \mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

a. Relationship between RMSE and MAE : **RMSE >= MAE** and **RMSE =< MAE*(n)$^{0.5}$**
b. RMSE and MAE are expected to give similar values when the data is evenly distributed, and have low variance. This is evident from the graph of |x| and x^2.



c. When two values are similar then RMSE should be preferred simply because it is convex and differentiable at all points.

Part (e)
The Loss function considered : **MAE**
The best fold according to my analysis was **fold index 1.**
So Calculating the weights from the normal equation and calculating the training and test mae error I got the following result :

```
In [79]: W = get_analytical_sol(Xtrain_i,ytrain_i)
         print(W)

[[ -0.25435685]
 [ -0.05179893]
 [  7.38089846]
 [  6.93155226]
 [  6.32780996]
 [-13.90095662]
 [ -6.61419301]
 [  5.71294489]
 [  8.94980457]]
```

```
In [80]: y_pred_train = np.dot(Xtrain_i,W)
         error = (np.sum(abs(ytrain_i-y_pred_train))/y_pred_train.shape[0])
         print(error)

1.5972868051383804
```

```
In [81]: y_pred_test = np.dot(X_test,W)
         error = (np.sum(abs(y_test-y_pred_test))/y_pred_test.shape[0])
         print(error)

1.4951636472575514
```

Optimal Weights = [[-0.25435685]
[-0.05179893]
[7.38089846]
[6.93155226]
[6.32780996]
[-13.90095662]
[-6.61419301]
[5.71294489]
[8.94980457]]

Training Loss = 1.5972868051383804
Validation Loss = 1.4951636472575514

Question 2

Exploratory Data Analysis for the Bank Note Dataset.

1. I computed the **correlation matrix** for the features of the dataset.

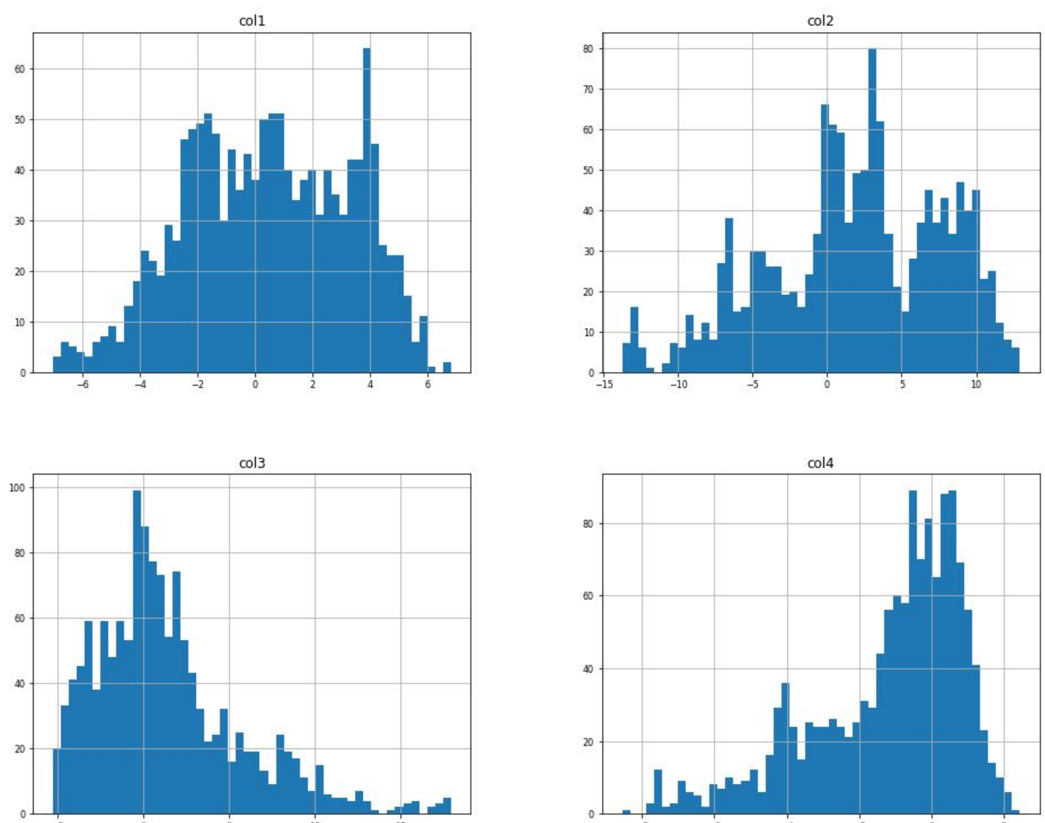
	col1	col2	col3	col4	val
col1	1.000000	0.264026	-0.380850	0.276817	-0.724843
col2	0.264026	1.000000	-0.786895	-0.526321	-0.444688
col3	-0.380850	-0.786895	1.000000	0.318841	0.155883
col4	0.276817	-0.526321	0.318841	1.000000	-0.023424
val	-0.724843	-0.444688	0.155883	-0.023424	1.000000

Correlation values signifies how the two variables are associated. The values of correlation range from -1.0 to 1.0.

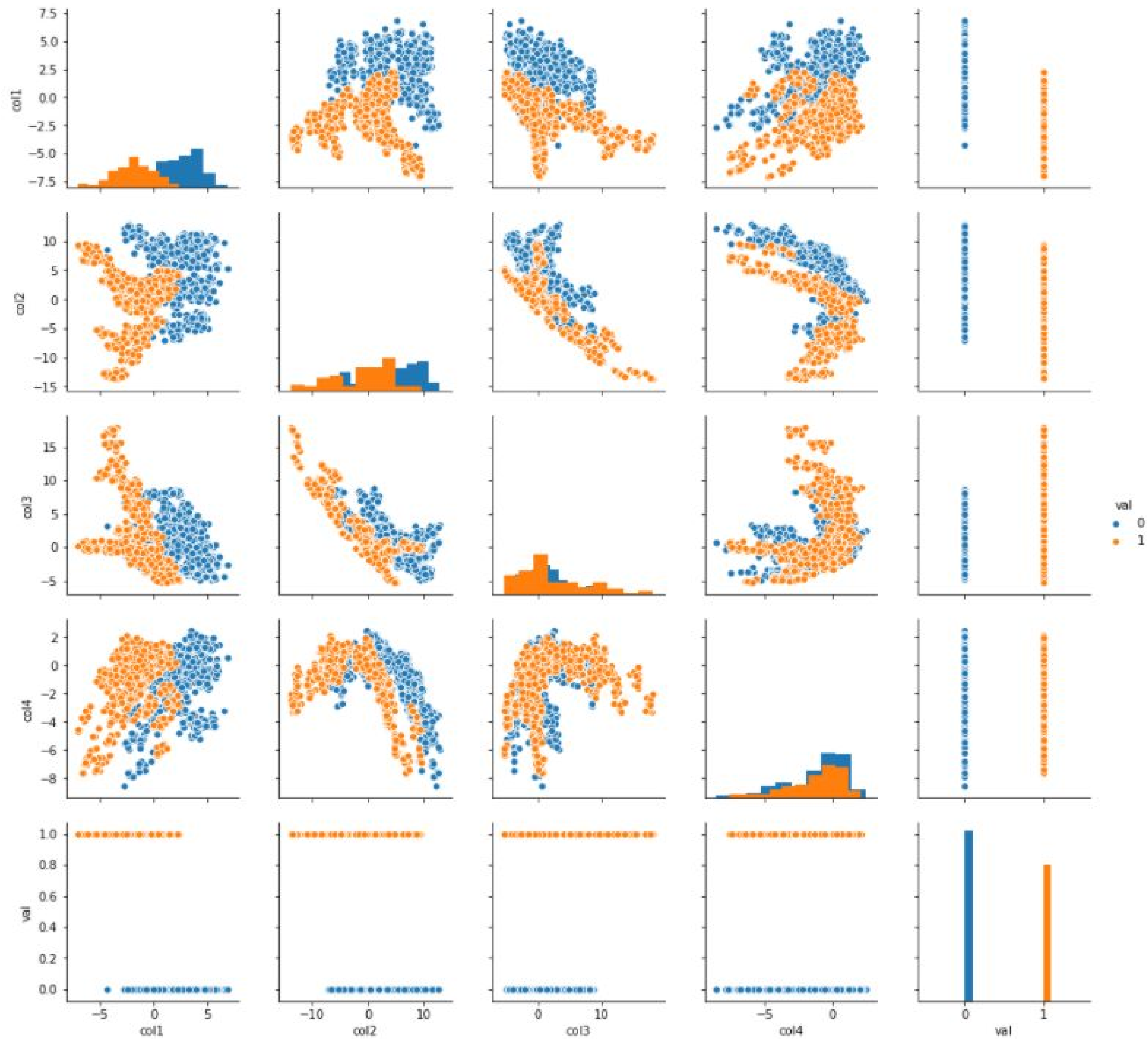
   a. Correlation **value 0** signifies that there is **no relation** between the two variables
   b. Correlation **value greater than 0** is called a Positive Correlation. Two variables having a positive correlation signifies that if one variable moves in a direction then the other variable also moves in the same direction.
   c. Correlation **value less than 0** is called a NegativeCorrelation. Two variables having a negative correlation signifies that if one variable moves in a direction then the other variable moves in the opposite same direction.

   Correlation matrix provides us with the overall picture of the dataset and how our features and labels are related.

2. I computed the histograms of each of the **features** which provides us an idea about the frequency and the nature of the data.



3. **Pair Plot Visualization** of the Data
   A Pair Plot allows us to visualize the relationship between different variables and also study the distribution of a single variable.
   The plot below shows the pairwise relationship between different features and histograms of each of the features. The plot is also color coded according to the binary classification in the dataset.

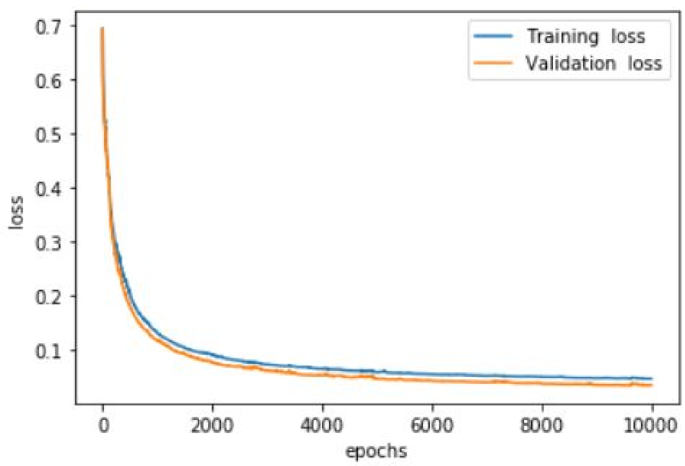


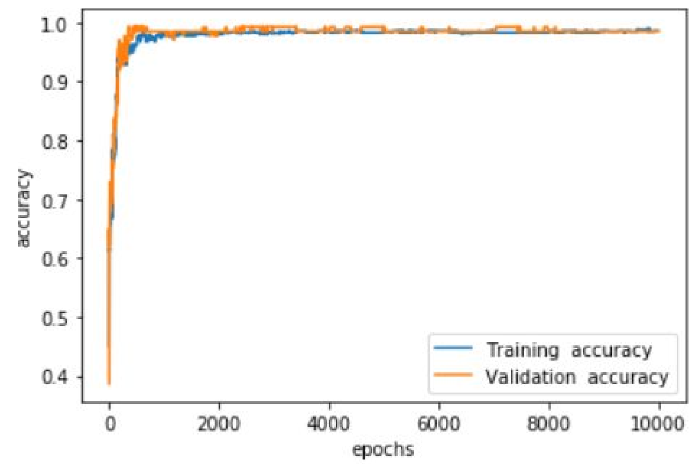
The data was further **normalised and shuffled** and then the logistic model was trained.

Analysis Using SGD (Stochastic Gradient Descent)

Part (a) and (b) Model Trained using SGD (Stochastic Gradient Descent) and loss v.s epochs plots

Learning Rate chosen = **0.05**
Epochs chosen = **10000**





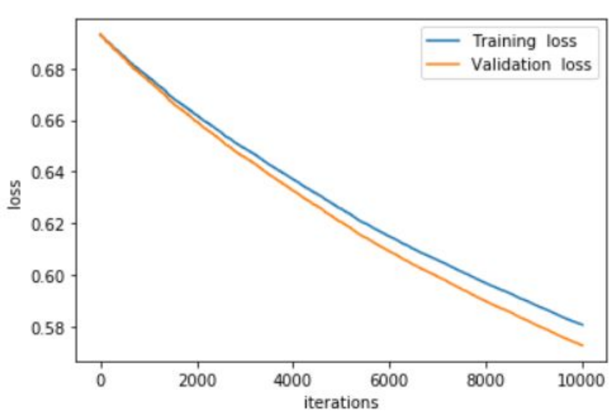
After experimenting from multiple learning rates and epochs I discovered that the losses stabilize after reaching a
training loss = 0.04685103917919263
validation loss = 0.03525275792366691

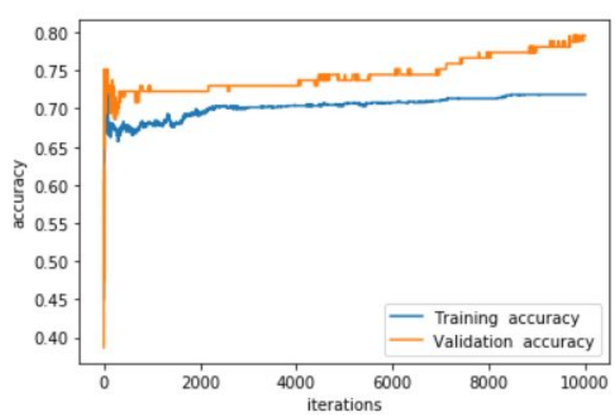
```
Training loss after  9500  sgd steps is :  0.04685103917919263  | validation loss is :  0.03525275792366691
Training accuracy after  9500  sgd steps is :  98.33159541188738 %  | validation accuracy is :  98.54014598540147 %
```

The Final accuracy achieved by the model :
Training Accuracy = 98.33159541188738 %
Validation Accuracy = 98.54014598540147 %

Part (c) Re-run the SGD model implementation for 3 variations in learning rates - 0.0001, 0.01, 10

1. **Learning rate = 0.0001**

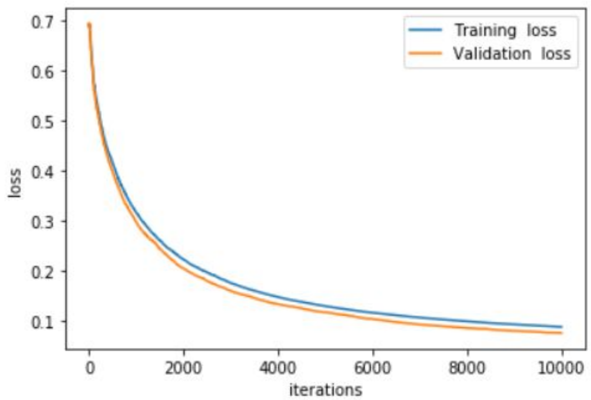


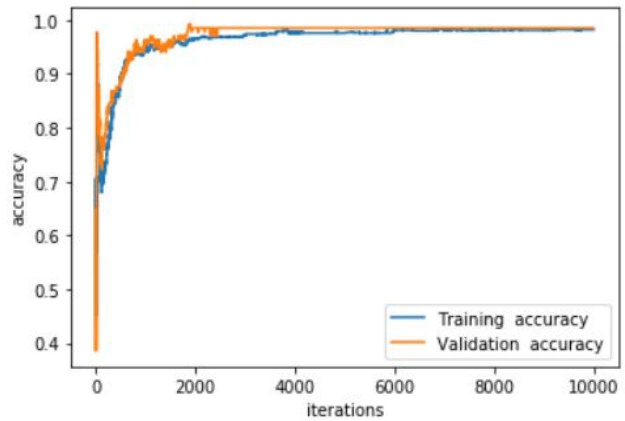


I ran 10k epochs for 0.0001 and it is clear that the loss didn't stabilize.

```
Training loss after  9500  sgd steps is :  0.5843827254732823  | validation loss is :  0.5765853795609763
Training accuracy after  9500  sgd steps is :  71.84567257559958 %  | validation accuracy is :  78.1021897810219 %
```

2. **Learning rate = 0.01**

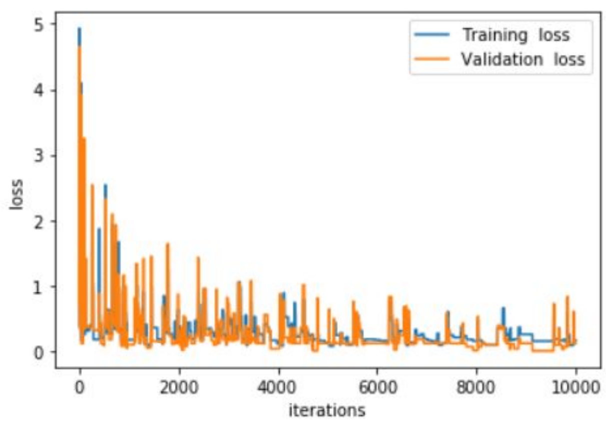


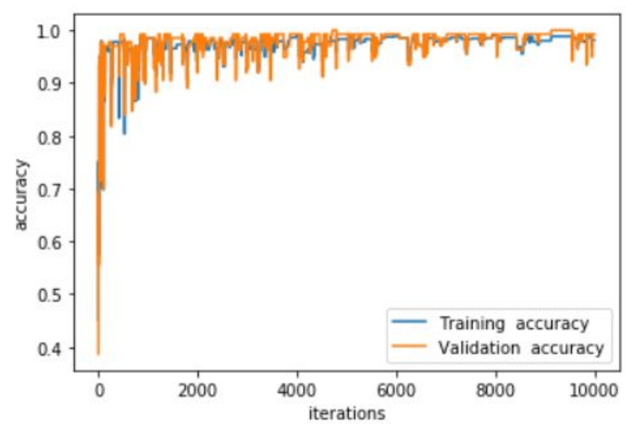


I ran 10k epochs and the final loss and accuracy is :

```
Training loss after  9500  sgd steps is :  0.09137139168173238  | validation loss is :  0.07844118696398661
Training accuracy after  9500  sgd steps is :  98.43587069864442 %  | validation accuracy is :  98.54014598540147 %
```

3. **Learning rate = 10**





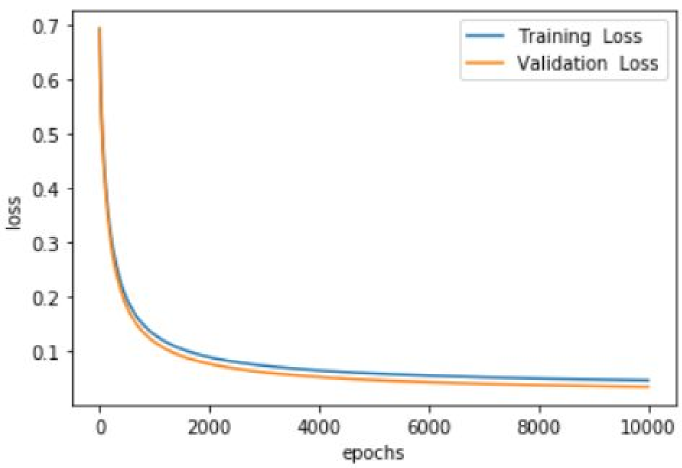
On running 10k epochs for 10 learning rate the loss never stabilised. The loss was fluctuating a lot which indicates that the gradient was jumping around the minimum due to the very high learning rate.

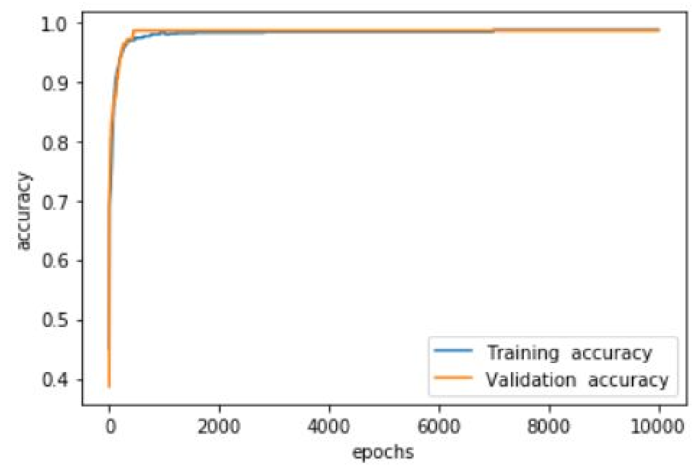
```
Training loss after  6500  sgd steps is :  0.3246916658386852  | validation loss is :  0.1176502344033065
Training accuracy after  6500  sgd steps is :  97.60166840458811 %  | validation accuracy is :  99.27007299270073 %

Training loss after  7000  sgd steps is :  0.3785525081297033  | validation loss is :  0.11768790951450223
Training accuracy after  7000  sgd steps is :  97.39311783107404 %  | validation accuracy is :  99.27007299270073 %

Training loss after  7500  sgd steps is :  0.23977450409186038  | validation loss is :  0.1176502547141521
Training accuracy after  7500  sgd steps is :  98.1230448383733 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8000  sgd steps is :  0.19904509344784774  | validation loss is :  0.08562716967657175
Training accuracy after  8000  sgd steps is :  98.1230448383733 %  | validation accuracy is :  98.54014598540147 %

Training loss after  8500  sgd steps is :  0.48662026591503754  | validation loss is :  0.26348477625058164
Training accuracy after  8500  sgd steps is :  95.9332638164755 %  | validation accuracy is :  97.8102189781022 %

Training loss after  9000  sgd steps is :  0.29323086738643844  | validation loss is :  0.2921859458981061
Training accuracy after  9000  sgd steps is :  97.8102189781022 %  | validation accuracy is :  97.8102189781022 %

Training loss after  9500  sgd steps is :  0.4609607774942761  | validation loss is :  0.19726055072739054
Training accuracy after  9500  sgd steps is :  96.66319082377477 %  | validation accuracy is :  97.08029197080292 %
```

Analysis Using BGD (Batch Gradient Descent)

Part (a) and (b) Model Trained using BGD (Batch Gradient Descent) and loss v.s epochs plots

Learning Rate chosen = **0.05**
Epochs chosen = **10000**





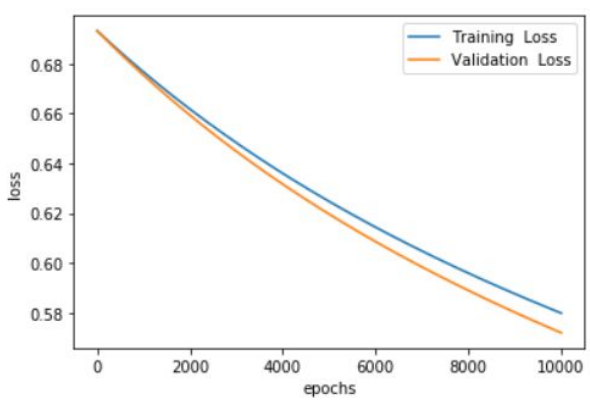
After experimenting from multiple learning rates and epochs I discovered that the losses stabilize after reaching a
training loss = 0.04670879098957566
validation loss = 0.034763497777172414

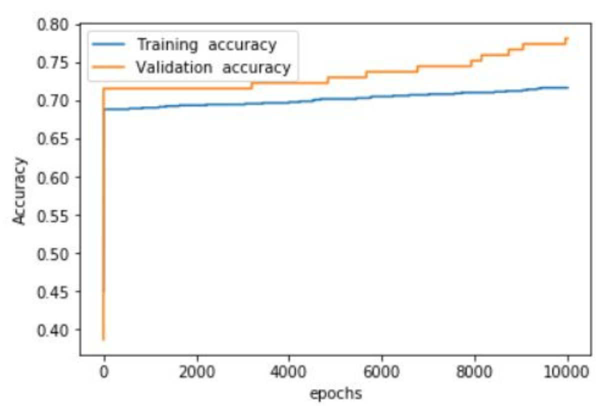
```
Training loss after  9500  bgd steps is :  0.04670879098957566  | validation loss is :  0.034763497777172414
Training accuracy after  9500  bgd steps is :  98.6444212721585 %  | validation accuracy is :  98.54014598540147 %
```

The Final accuracy achieved by the model :
Training Accuracy = 98.6444212721585 %
Validation Accuracy = 98.54014598540147 %

Part (c) Re-run the BGD model implementation for 3 variations in learning rates - 0.0001, 0.01, 10

1. **Learning rate = 0.0001**

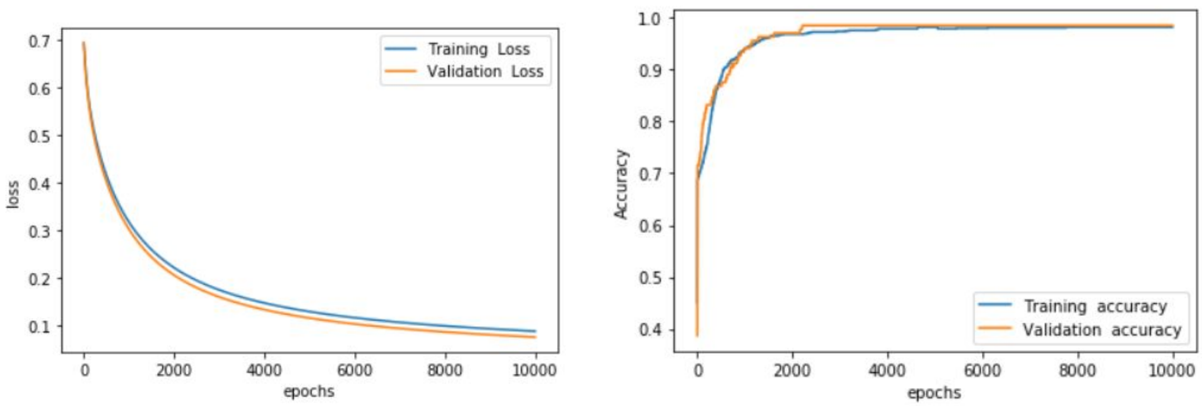




I ran 10k epochs for 0.0001 and it is clear that the loss didn't stabilize.

```
Training loss after  9500  bgd steps is :  0.5836633889284062  | validation loss is :  0.5759986504445361
Training accuracy after  9500  bgd steps is :  71.6371220020855 %  | validation accuracy is :  77.37226277372262 %
```

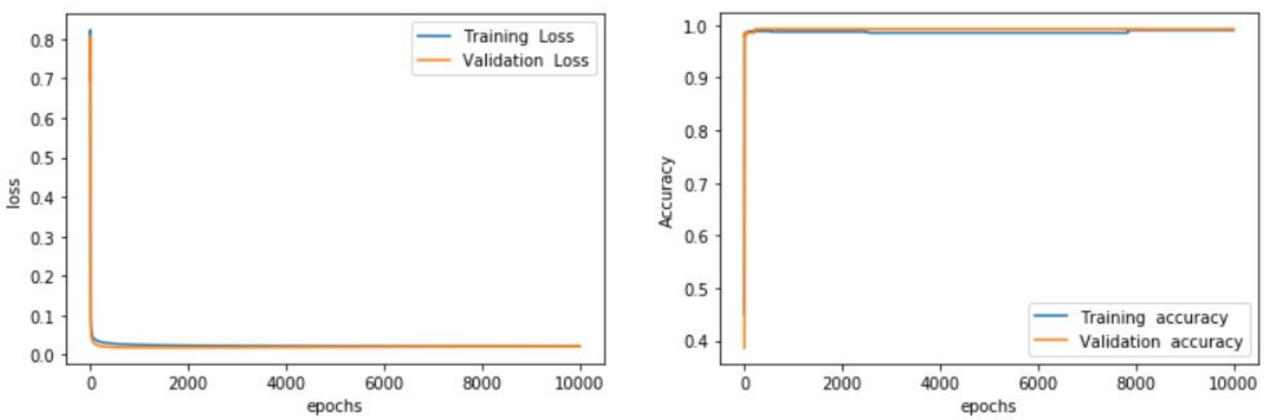
2. **Learning rate = 0.01**



I ran 10k epochs and the final loss and accuracy is :

```
Training loss after  9500  bgd steps is :  0.09119398914279395  | validation loss is :  0.07873036592325892
Training accuracy after  9500  bgd steps is :  98.22732012513035 %  | validation accuracy is :  98.54014598540147 %
```

4. **Learning rate = 10**



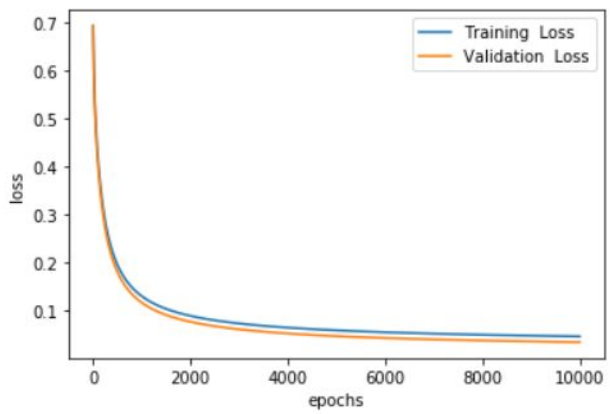
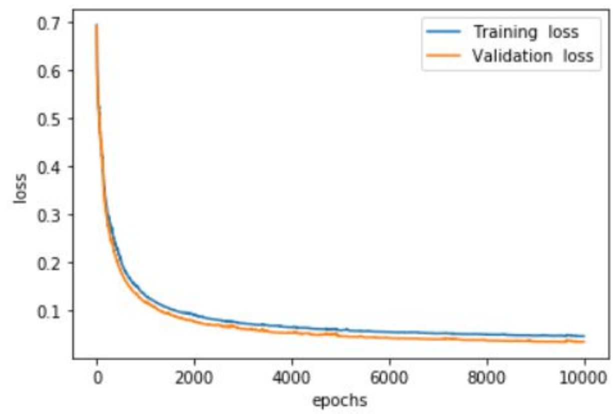
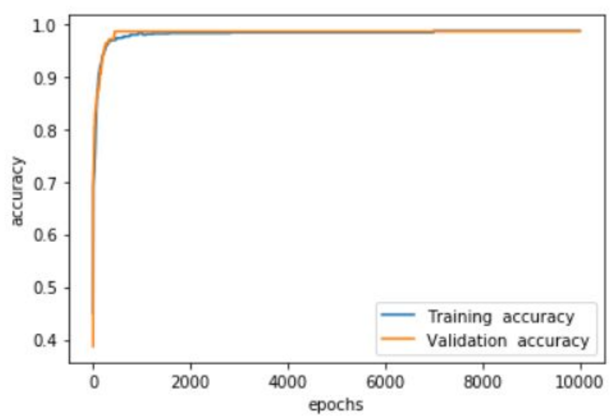
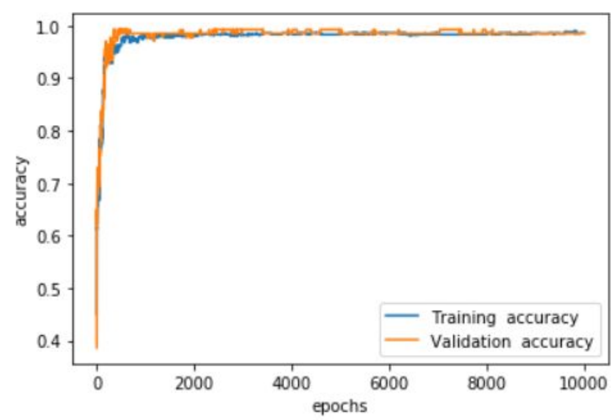
On running 10k epochs for 10 learning rate the loss stabilised extremely quickly.

```
Training loss after  7500  bgd steps is :  0.021068030912323246  | validation loss is :  0.0201172415591449
Training accuracy after  7500  bgd steps is :  98.54014598540147 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8000  bgd steps is :  0.021019373386096222  | validation loss is :  0.020240311942768527
Training accuracy after  8000  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8500  bgd steps is :  0.02097715498703789  | validation loss is :  0.0203563800449012
Training accuracy after  8500  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9000  bgd steps is :  0.02094034492255275  | validation loss is :  0.02046593960018023
Training accuracy after  9000  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9500  bgd steps is :  0.02090811132721569  | validation loss is :  0.020569445003227653
Training accuracy after  9500  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %
```

Comparison between SGD and BGD

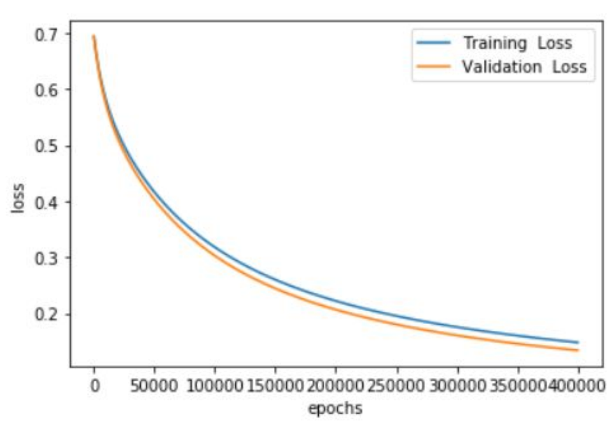
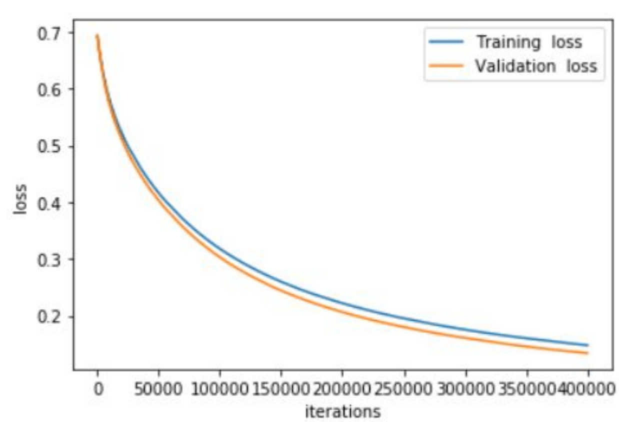
Part (a) and (b) : Loss plots and number of epochs to converge.

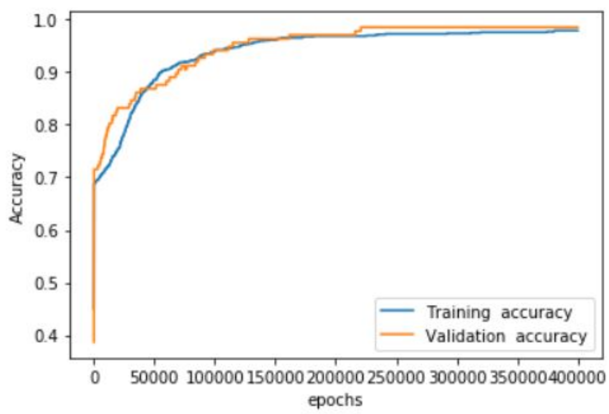
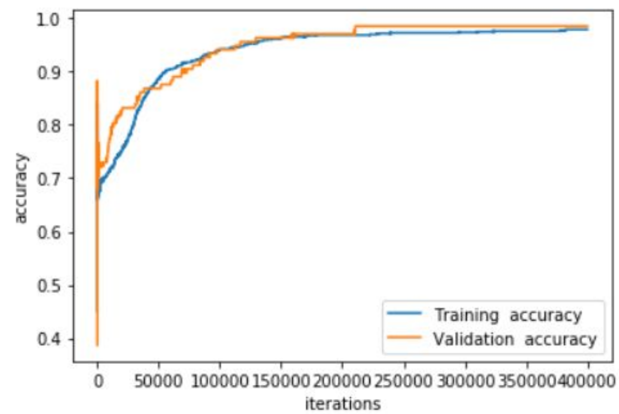
1. **Learning rate = 0.05**

<table>
<tr><th>BGD</th><th>SGD</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

Both SGD and BGD converge around 5000 epochs as seen from the above plots.

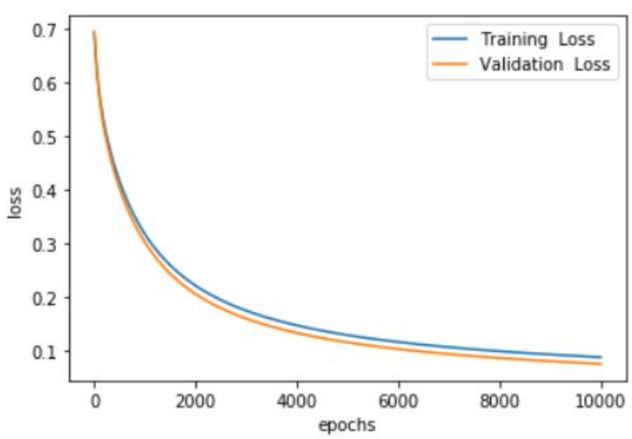
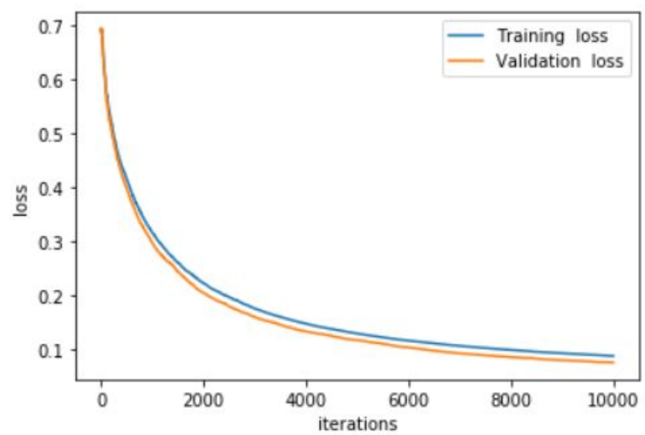
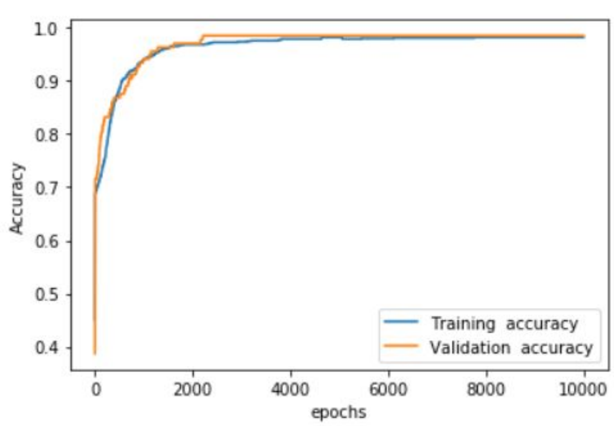
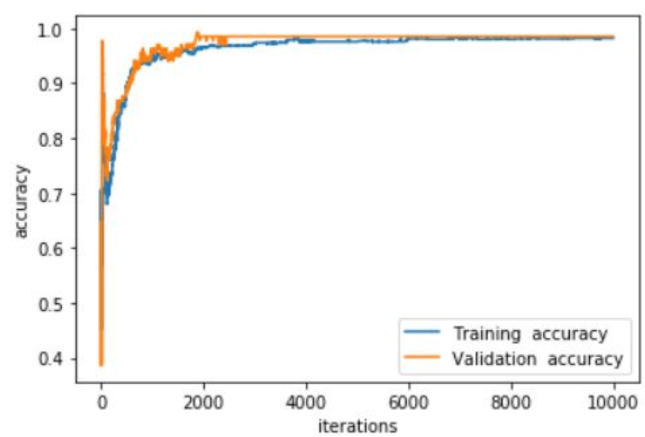
2. **Learning rate = 0.0001**

<table>
<tr><th>BGD</th><th>SGD</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

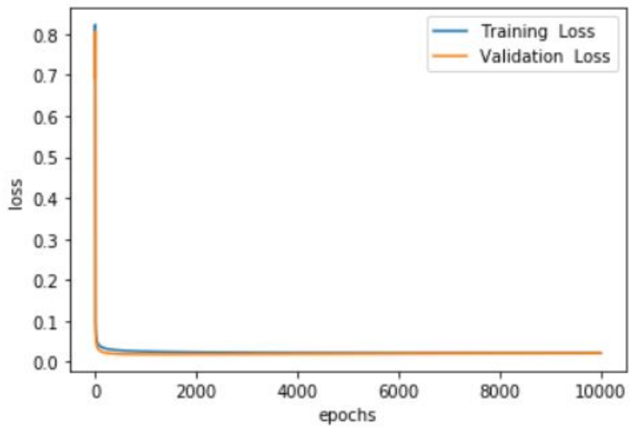
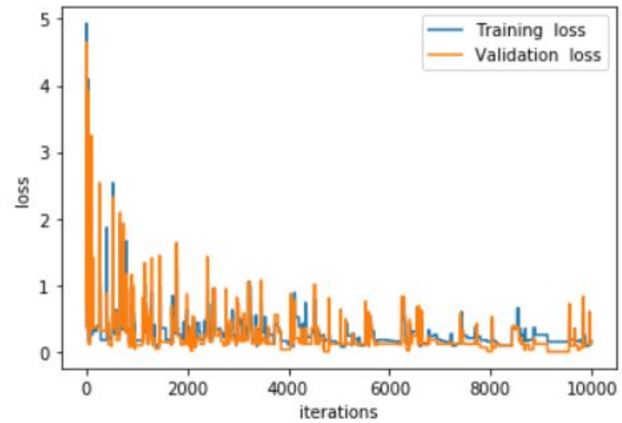
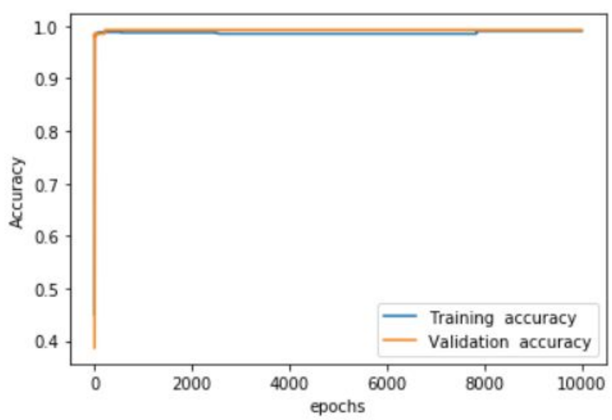
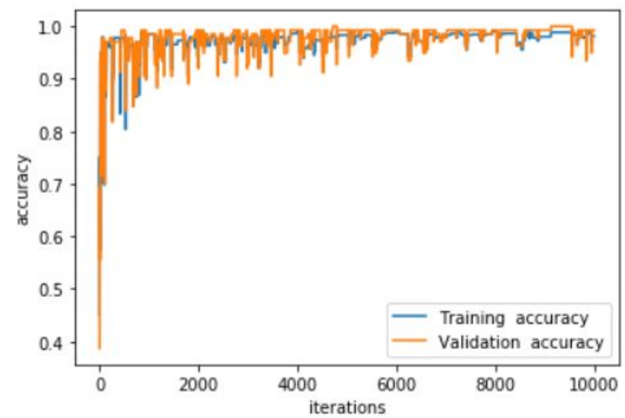
Both SGD and BGD converge > 400k epochs as seen from the above plots.

3. **Learning rate = 0.01**

BGD	SGD
	
	

Both SGD and BGD converge around 10k epochs as seen from the above plots.

4. **Learning rate = 10**

BGD	**SGD**
	
	

BGD converged very quickly around 100 epochs as seen from the above plots.
SGD fluctuated the entire duration without stabilizing.

Part (c) Sklearn implementation for Logistic Regression.

```
In [28]: from sklearn.linear_model import LogisticRegression
         from sklearn import metrics
```

```
In [29]: logistic_regression = LogisticRegression(max_iter=10000)
         logistic_regression.fit(X_train,y_train)
         y_pred = logistic_regression.predict(X_test)
         accuracy = metrics.accuracy_score(y_test, y_pred)
         print("Accuracy on the test set :",accuracy*100, " %")
         y_pred_train = logistic_regression.predict(X_train)
         accuracy = metrics.accuracy_score(y_pred_train, y_train)
         print("Accuracy Train:",accuracy*100, " %")

         Accuracy on the test set : 97.82608695652173  %
         Accuracy Train: 98.33159541188738  %
```

Part (d)

Accuracy on the test set : **97.82608695652173 %**
Accuracy Train: **98.33159541188738 %**

Q3 $y\_true = 1$ or $y\_true = 0$; $\hat{y} = \sigma(X\theta)$
$y\_pred = 0$ $y\_pred = 1$

Using MSE, we get an error of $\frac{\sum_{i=1}^{m} |y - \hat{y}|^2}{m}$
where m is the number of training ex.

$$Loss = (0-1)^2 = 1$$

where as when using binary cross entropy loss

$$Loss = -(1 * \log(0) + 0 * \log(1)) \longrightarrow \infty$$ — (3)

(I) **Now**

Gradient when using MSE.

$$\frac{\partial M}{\partial \theta} = \frac{\partial (y-\hat{y})^2}{\partial \theta} = \frac{\partial (y-\hat{y})^2}{\partial \hat{y}} \cdot \frac{\partial (\hat{y})}{\partial \theta} \quad \text{— (1)}$$

$$\frac{\partial M}{\partial \theta} = 2(y-\hat{y})(-1)\left[\frac{\partial(\sigma(X\theta))}{\partial \theta}\right]$$; $\sigma(x) \rightarrow$ sigmoid function.

$$= 2(y-\hat{y})(-1)\,\sigma(X\theta)(1-\sigma(X\theta)) \cdot X$$

$$\frac{\partial M}{\partial \theta} = -2(y-\hat{y})\,\hat{y}(1-\hat{y}) \cdot X \quad \text{— (2)}$$

Since $\hat{y}$ always gives the opposite value as y

$\therefore$ if $\hat{y} \Rightarrow 1$ and $y = 0$ $\rightarrow$ $\frac{\partial M}{\partial \theta} \longrightarrow 0$

and $\hat{y} \Rightarrow 0$ and $y = 1$ $\rightarrow$ $\frac{\partial M}{\partial \theta} \Rightarrow 0$

∴ gradient approaches 0.

The model won't be able to learn efficiently as the gradient is approaching 0, thus during ~~upad~~ updation.

$$\theta = \theta - \cancel{\alpha \cdot \frac{\partial J}{\partial \theta}} \to 0$$

θ won't get updated from the initial value, Hence no training.

(14) If we use cross entropy then, from (3) we know that the loss $\to \infty$.

let's check the gradient.

$$L = -\left(y \log \hat{y} + (1-y)\log(1-\hat{y})\right); \quad \hat{y} = \sigma(X\theta)$$

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \theta} = -\left(\frac{y}{\hat{y}} + \frac{(1-y)}{1-\hat{y}}(-1)\right) \cdot \frac{\partial(\sigma(X\theta))}{\partial \theta}$$

$$= -\left(\frac{y}{\hat{y}} - \frac{(1-y)}{1-\hat{y}}\right) \cdot \sigma(X\theta) \cdot (1-\sigma(X\theta)) \cdot \frac{\partial(X\theta)}{\partial \theta}$$

$$= -\left(\frac{y}{\hat{y}} - \frac{(1-y)}{1-\hat{y}}\right) \hat{y} \cdot (1-\hat{y}) \cdot X$$

$$= -\left(y(1-\hat{y}) - \hat{y}(1-y)\right) \cdot X$$

$$\frac{\partial L}{\partial \theta} = -(y-\hat{y}) \cdot X$$

Since $\hat{y}$ always gives the opposite values of y

$\therefore$ if $\hat{y} \to 1$ and $y = 0$ $\Rightarrow \frac{\partial L}{\partial \theta} \to X$

and $\hat{y} \to 0$ and $y = 1$ $\Rightarrow \frac{\partial L}{\partial \theta} \to -X$.

$\Rightarrow \frac{\partial L}{\partial \theta} \neq 0$.

Since the gradient is non-zero, the parameters

i.e $\theta = \theta - \alpha \cdot \frac{\partial L}{\partial \theta}$ would get updated.

Hence model would learn.

P.T.O

Q5

$$y_i = \beta_1 + \beta_2 x_{2i} + \beta_3 x_{3i} + \dots + \beta_k x_{ki} + \epsilon_i$$

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} 1 & x_{21} & \cdots & x_{k1} \\ \vdots & & & \\ \vdots & & & \\ 1 & x_{2n} & \cdots & x_{kn} \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_k \end{pmatrix}, \quad \epsilon = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

The linear model is $Y = X\beta + \epsilon$

Sum of square errors $= \epsilon_1^2 + \epsilon_2^2 + \dots \epsilon_n^2$

$$\therefore L = \epsilon^2 = (Y - X\beta)^2 = (Y - X\beta)^T (Y - X\beta) \quad \text{— (1)}$$

Our goal is to minimize this loss.

$$\Rightarrow \frac{dL}{d\beta} = 0 \quad \text{for the condition of minimum.}$$

$$\therefore \cancel{\frac{dL}{d\beta}} \quad L = (Y - X\beta)^T (Y - X\beta)$$

$$L = (Y^T - \beta^T X^T)(Y - X\beta)$$

$$L = Y^T Y - Y^T X\beta - \beta^T X^T Y + \beta^T X^T X\beta \quad \text{— (2)}$$

Now, we know that $Y = (n,1)$, $X = (n,k)$, $\beta = (k,1)$

$$\Rightarrow \beta^T X^T Y = (1,1)$$

and $Y^T X\beta = (1,1)$

Thus $\beta^T X^T Y$ and $Y^T X\beta$ are scalars.

and $\beta^T X^T Y = (Y^T X\beta)^T$

$$\therefore |\beta^T X^T Y| = |Y^T X\beta| \quad \text{— (3)}$$

Thus using (8), equation (2) becomes.

$$L = y^T y + \beta^T X^T X \beta - 2\beta^T X^T y$$

$\therefore$ for minimum $\frac{dL}{d\beta} = 0$

$$\therefore \frac{d(y^T y)}{d\beta} + \frac{d(\beta^T X^T X \beta)}{d\beta} - 2\frac{d(\beta^T X^T y)}{d\beta} = 0$$

$$\Rightarrow 0 + 2X^T X \beta - 2X^T y = 0$$

$$\Rightarrow \cancel{2} X^T X \beta - \cancel{2} X^T y = 0$$

$$\Rightarrow \boxed{\beta^* = (X^T X)^{-1} X^T y} \rightarrow \text{Solution}$$

The solution will exist only if the following conditions are satisfied.

1. X must be normalised to give the correct output.
2. $(X^T X)$ must be invertible.
3. $X^T X$ must be non-singular.

Scanned with CamScanner

QUESTION 4

Q4

We assume a bernoulli distribution for the dataset. as it is a binary classification (0/1)

$$\therefore \; P(Y = y \mid X = x) = \sigma(\theta^T x)^y \cdot \left[1 - \sigma(\theta^T x)\right]^{(1-y)}$$

↳ (Similar Done in Lectures)

Thus the likelihood of all data is

$$L = \prod_{i=1}^{n} P\left(Y = y^{(i)} \mid X = x^{(i)}\right)$$

$$= \cancel{\prod_{i=1}^{n} \sigma(\theta^T y)} \; \prod_{i=1}^{n} \sigma(\theta^T x^{(i)})^{y^{(i)}} \cdot \left[1 - \sigma(\theta^T x^{(i)})\right]^{1 - y^{(i)}} \quad \hookrightarrow (1)$$

Thus taking log of (1) we get the log likelihood for logistic regression.

$$\log(L) = \underbrace{\sum_{i=1}^{n} \left[y^{(i)} \log\left(\sigma(\theta^T x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - \sigma(\theta^T x^{(i)})\right) \right]}_{\text{Loss fn for logistic regression (cross-entropy)}}$$

Now training the dataset given for Q4 in my logistic regression model

```
In [34]: df=pd.read_csv('Q4_Dataset.txt',delim_whitespace=True,header=None)
         X=df[[1,2]].to_numpy()
         y=df[[0]].to_numpy()
         X = (X - X.mean())/X.std()
```

```
In [39]: log_regressor = MyLogisticRegression()
         log_regressor.fit(X,y, epochs = 10000 ,learning_rate = 0.05)

         Training loss after  7000  iterations is  0.585928727887266
         Training accuracy after  7000  iterations is :  66.66666666666667 %

         Training loss after  7500  iterations is  0.5855034372555866
         Training accuracy after  7500  iterations is :  66.66666666666667 %

         Training loss after  8000  iterations is  0.5850843839010013
         Training accuracy after  8000  iterations is :  66.66666666666667 %

         Training loss after  8500  iterations is  0.5846714753210276
         Training accuracy after  8500  iterations is :  66.66666666666667 %

         Training loss after  9000  iterations is  0.584264620301743
         Training accuracy after  9000  iterations is :  66.66666666666667 %

         Training loss after  9500  iterations is  0.5838637289050078
         Training accuracy after  9500  iterations is :  66.66666666666667 %

Out[39]: <scratch.MyLogisticRegression at 0x25c6802e088>
```

```
In [40]: print(log_regressor.W)
         print(log_regressor.b)

         [[1.09481727]
          [2.22579592]]
         0.5313976598954194
```

1. **Beta_2 = 1.09481727**
   Beta_1 = 2.22579592
   Beta_0 = 0.5313976598954194

2. The fitted response function is the hypothesis of our model i.e.

   y_hat = 1/(1 + exp-(beta_0 + beta_1*X1 + beta_2*X2))

3. Taking **exp(beta_1)** and **exp(beta_2)** we get : **[[2.98863653] [9.26085077]]**
   a. **E1 = exp(beta_1)** signifies that with a unit increase in the age of the child the odds of disease occurring (y=1) to disease not occuring (y=0) increases by **9.26085077** times**.**
   b. **E2 = exp(beta_2)** signifies that with a unit increase in the percentage spread of the disease the odds of disease occurring (y=1) to disease not occuring (y=0) increases by **2.98863653** times**.**

The estimated probability that a patient with 75% of disease spread and an age of 2 years will have a recurrence of disease in the next 5 years is **0.8981808926762442**