CSE : 343 Machine Learning

Assignment 1

Arka Sarkar , 2018222

Question 1

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 =  2.229849698620617  validation loss =  2.2556926183537986
The average loss with k =  3  is   train loss =  2.2314885848865598  validation loss =  2.2524277996500284
The average loss with k =  4  is   train loss =  2.2309664698155514  validation loss =  2.257347751227494
The average loss with k =  5  is   train loss =  2.2314455100122847  validation loss =  2.255293611505139
The average loss with k =  6  is   train loss =  2.232600791143001  validation loss =  2.256092005783986
The average loss with k =  7  is   train loss =  2.2327921822046193  validation loss =  2.2575755910303568
The average loss with k =  8  is   train loss =  2.232910815564624  validation loss =  2.2566897090788074
The average loss with k =  9  is   train loss =  2.2333792985994214  validation loss =  2.2570576995427825
The average loss with k =  10  is   train loss =  2.2334369389349833  validation loss =  2.255565748030461
```

Figure 1 : Abalone Dataset K Fold

```
The average loss with k =  2  is   train loss =  1.5128465554619095  validation loss =  1.5129411073792254
The average loss with k =  3  is   train loss =  1.5104759993792245  validation loss =  1.4951161066004357
The average loss with k =  4  is   train loss =  1.5142599191765818  validation loss =  1.5063851029349744
The average loss with k =  5  is   train loss =  1.51287203041841  validation loss =  1.4839413774937251
The average loss with k =  6  is   train loss =  1.5130574456788002  validation loss =  1.4699168337071662
The average loss with k =  7  is   train loss =  1.5140970070754494  validation loss =  1.4742793679498507
The average loss with k =  8  is   train loss =  1.5144919267438042  validation loss =  1.474297176929585
The average loss with k =  9  is   train loss =  1.5147533157694322  validation loss =  1.4794465646425081
The average loss with k =  10  is   train loss =  1.5144726027047515  validation loss =  1.4559566789845473
```

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 :

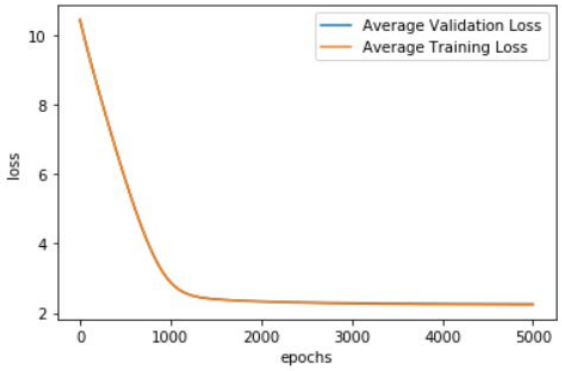
$$z_i = \frac{x_i - \bar{x}}{s}$$

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

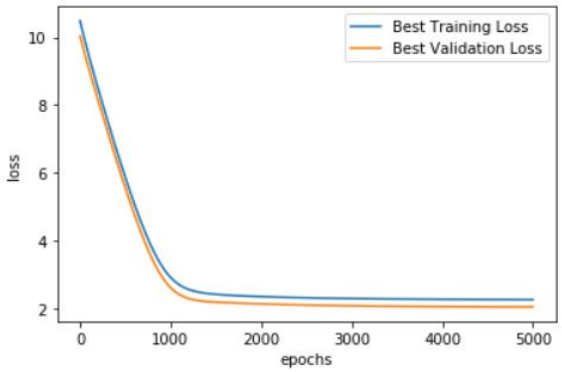
The Final Step was shuffling of the dataset.

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

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

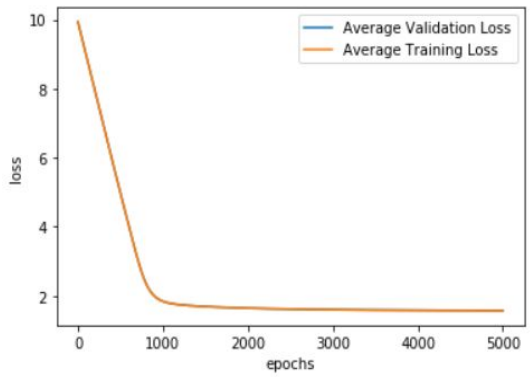


← Average RMSE Loss over all the K-Folds Plot

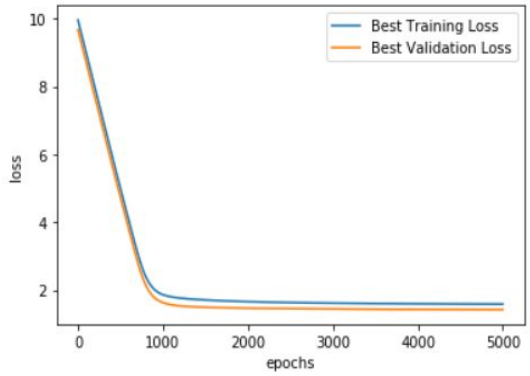


← Best Fold RMSE Loss Plot

b. For MAE Loss



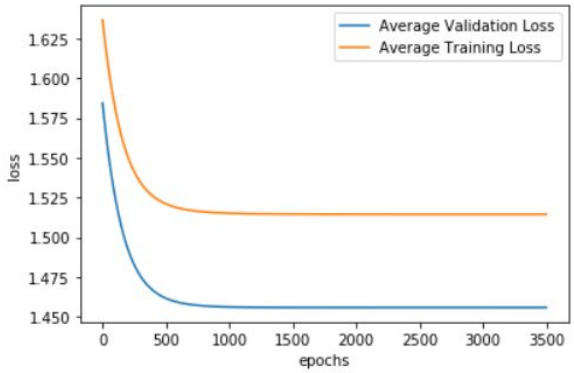
← Average MAE Loss over all K-folds Plot



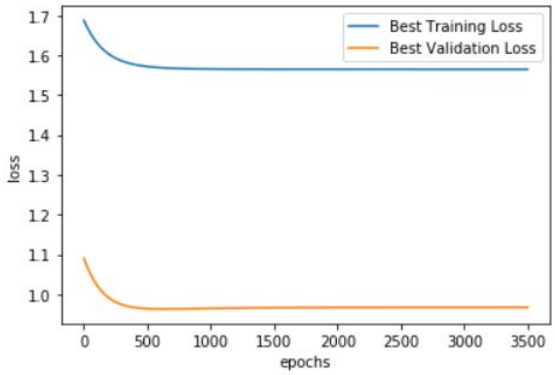
← Best Fold MAE Loss plot

2. Video Game Dataset

a. For RMSE Loss

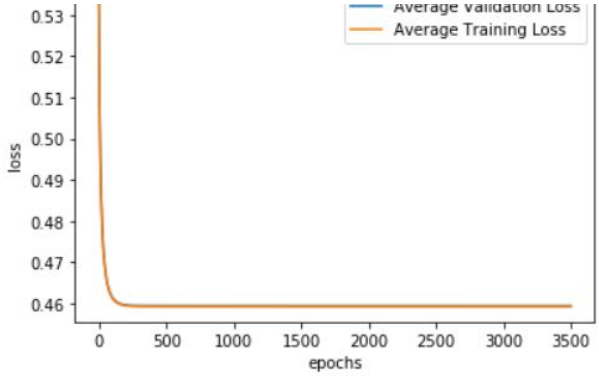


← Average RMSE Loss over all the K-Folds Plot

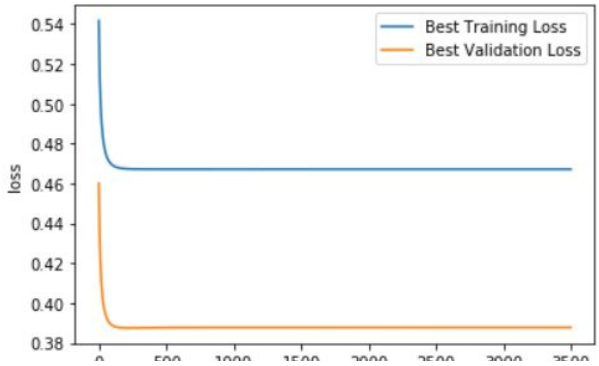


← Best Fold RMSE Loss Plot

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 =  0.45699777280210174  and the val loss =  0.4794305875998526
For CV number  1 the train loss =  0.46190686364648736  and the val loss =  0.4348008539326736
For CV number  2 the train loss =  0.4507377454538373  and the val loss =  0.5357182538972228
For CV number  3 the train loss =  0.4594885958767579  and the val loss =  0.45695310333692807
For CV number  4 the train loss =  0.46594191784987354  and the val loss =  0.3988062440940079
For CV number  5 the train loss =  0.45667520159297104  and the val loss =  0.4822479994492268
For CV number  6 the train loss =  0.4671727997697697  and the val loss =  0.3877950464553649
For CV number  7 the train loss =  0.4538603841330389  and the val loss =  0.5075830230387078
For CV number  8 the train loss =  0.45770713510567584  and the val loss =  0.4727399062916709
For CV number  9 the train loss =  0.4615709789592122  and the val loss =  0.43761736361008263
Stats for LR RMSE Loss
For CV number  0 the train loss =  1.5279267556929685  and the val loss =  1.3987164153298155
For CV number  1 the train loss =  1.5484332965112981  and the val loss =  1.1783536368466818
For CV number  2 the train loss =  1.3747277165905787  and the val loss =  2.4418883289997884
For CV number  3 the train loss =  1.5045506547703018  and the val loss =  1.610740976830723
For CV number  4 the train loss =  1.5643001370387803  and the val loss =  0.9715999612618064
For CV number  5 the train loss =  1.5280310795873362  and the val loss =  1.39775618857003
For CV number  6 the train loss =  1.564615604952  and the val loss =  0.9671759118931997
For CV number  7 the train loss =  1.4645146020700524  and the val loss =  1.914486024992358
For CV number  8 the train loss =  1.5330973483086257  and the val loss =  1.3468281615026443
For CV number  9 the train loss =  1.5345302015463846  and the val loss =  1.3319848041491635
```

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

Train loss = 0.4671727997697697 and the val loss = 0.3877950464553649 ← MAE

Train loss = 1.564615604952 and the val loss = 0.9671759118931997 ← 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 =  1.5790079453446764  and the val loss =  1.4946051438626606
For CV number  1 the train loss =  1.5862642965124074  and the val loss =  1.4240977434729953
For CV number  2 the train loss =  1.5677558915056704  and the val loss =  1.5922469082822315
For CV number  3 the train loss =  1.5686591348646493  and the val loss =  1.5981470753429103
For CV number  4 the train loss =  1.5591642739969063  and the val loss =  1.705209587161556
For CV number  5 the train loss =  1.5774588854337188  and the val loss =  1.5066208910780743
For CV number  6 the train loss =  1.5724052579682588  and the val loss =  1.5507165483914473
For CV number  7 the train loss =  1.5750455710534508  and the val loss =  1.5486485293038714
For CV number  8 the train loss =  1.5704778436554456  and the val loss =  1.5821056761087104
For CV number  9 the train loss =  1.5533079332565543  and the val loss =  1.7405564842853942
Stats for LR RMSE Loss
For CV number  0 the train loss =  2.2547985464586877  and the val loss =  2.0412690011059484
For CV number  1 the train loss =  2.249698767346536  and the val loss =  2.07532426614657
For CV number  2 the train loss =  2.233280663288711  and the val loss =  2.233114347387998
For CV number  3 the train loss =  2.2307004213868926  and the val loss =  2.2794417841527426
For CV number  4 the train loss =  2.2151739359965603  and the val loss =  2.421574336456404
For CV number  5 the train loss =  2.246405823462628  and the val loss =  2.1135795977029903
For CV number  6 the train loss =  2.2400883442676127  and the val loss =  2.1790555885215483
For CV number  7 the train loss =  2.2318012813871224  and the val loss =  2.2631278876872827
For CV number  8 the train loss =  2.22963868553345  and the val loss =  2.2689822586442823
For CV number  9 the train loss =  2.2027829202216322  and the val loss =  2.6801884124988433
```

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

Train loss = 1.5862642965124074 and the val loss = 1.4240977434729953 ← MAE

Train loss = 2.249698767346536 and the val loss = 2.07532426614657 ← 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.

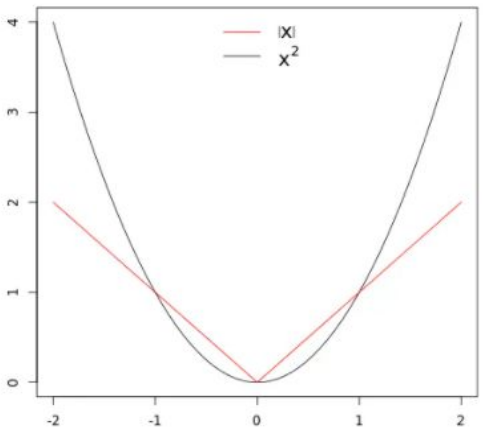
Now, SInce **Abalone dataset** is a regression problem to predict rings of the range 0-29, large errors need not be penalised to that extent therefore **MAE** is the better choice for this dataset. Even evident from part (a) and (b) the MAE received was **1.4240977434729953** which is very reasonable.

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)

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

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

```
[[-3.16573902e-01]
 [-9.35381651e-03]
 [ 1.10142143e+00]
 [ 4.35909882e-01]
 [ 4.66595525e+00]
 [-4.63952803e+00]
 [-1.09014505e+00]
 [ 1.19578816e+00]
 [ 9.95246476e+00]]
```

```
In [28]: y_pred_train = np.dot(Xtrain_i,W)
         error = (np.sum(abs(ytrain_i-y_pred_train))/y_pred_train.shape[0])
         print(error)
```

```
1.5972868051385163
```

```
In [29]: y_pred_test = np.dot(X_test,W)
         error = (np.sum(abs(y_test-y_pred_test))/y_pred_test.shape[0])
         print(error)
```

```
1.4951636472577468
```

Optimal Weights = [[-3.16573902e-01]
[-9.35381651e-03]
[1.10142143e+00]
[4.35909882e-01]
[4.66595525e+00]
[-4.63952803e+00]
[-1.09014505e+00]
[1.19578816e+00]
[9.95246476e+00]]

Training Loss = 1.5972868051385163

Validation Loss = 1.4951636472577468

Question 2

Exploratory Data Analysis for the Bank Note Dataset.

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

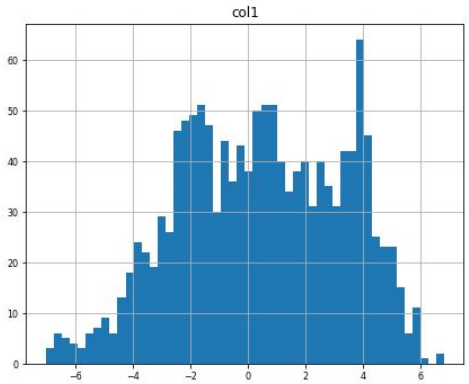
	col1	col2	col3	col4	val
col1	1.000000	0.264026	-0.380850	0.276817	-0.724843
col2	0.264026	1.000000	-0.786895	-0.526321	-0.444688
col3	-0.380850	-0.786895	1.000000	0.318841	0.155883
col4	0.276817	-0.526321	0.318841	1.000000	-0.023424
val	-0.724843	-0.444688	0.155883	-0.023424	1.000000

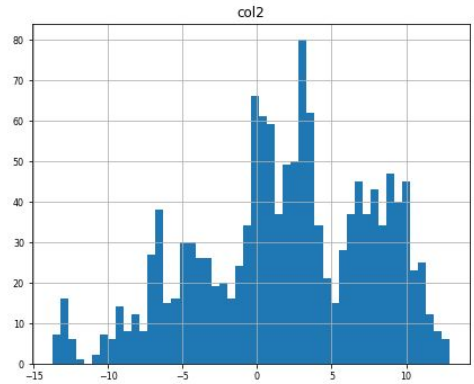
Correlation values signifies how the two variables are associated. The values of correlation range from -1.0 to 1.0.

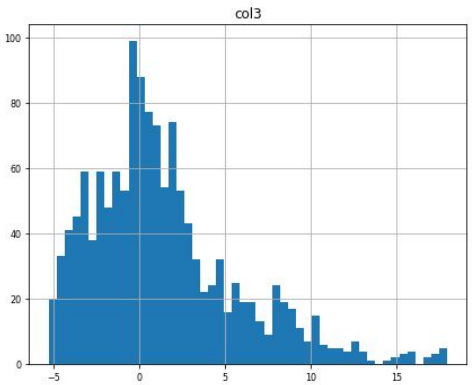
   a. Correlation **value 0** signifies that there is **no relation** between the two variables
   b. Correlation **value greater than 0** is called a Positive Correlation. Two variables having a positive correlation signifies that if one variable moves in a direction then the other variable also moves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

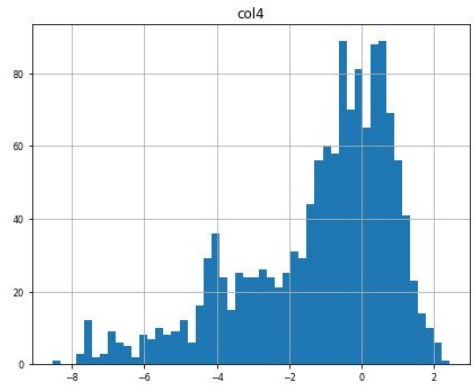
   Correlation matrix provides us with the overall picture of the dataset and how our features and labels are related.

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





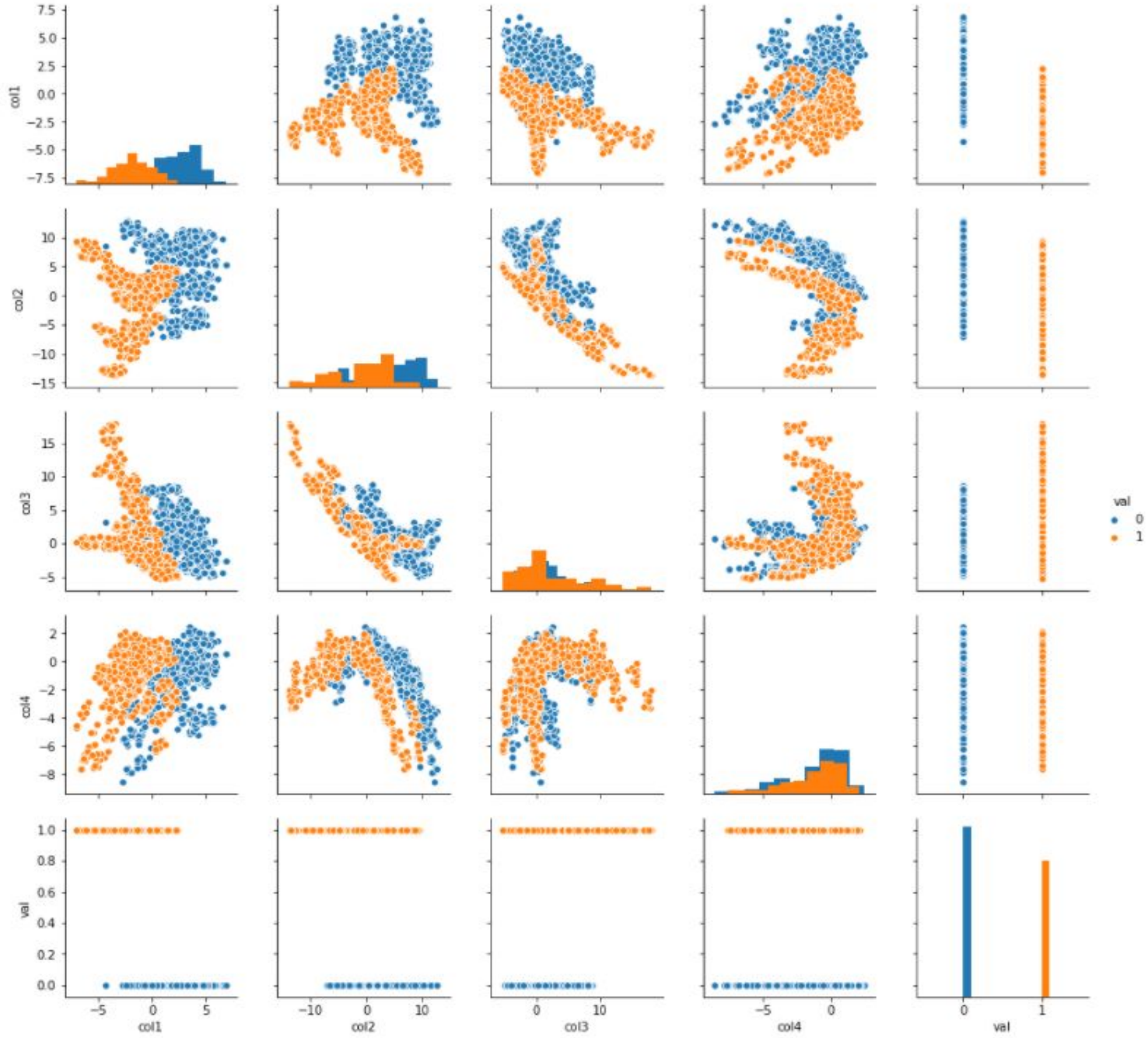




3. **Pair Plot Visualization** of the Data

   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.

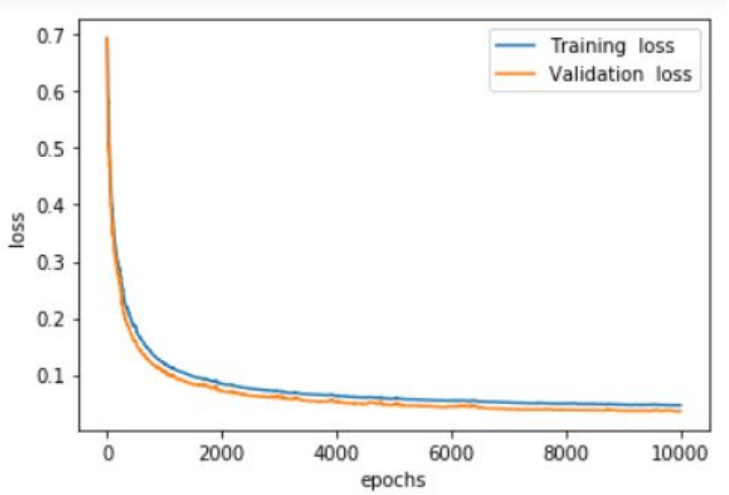


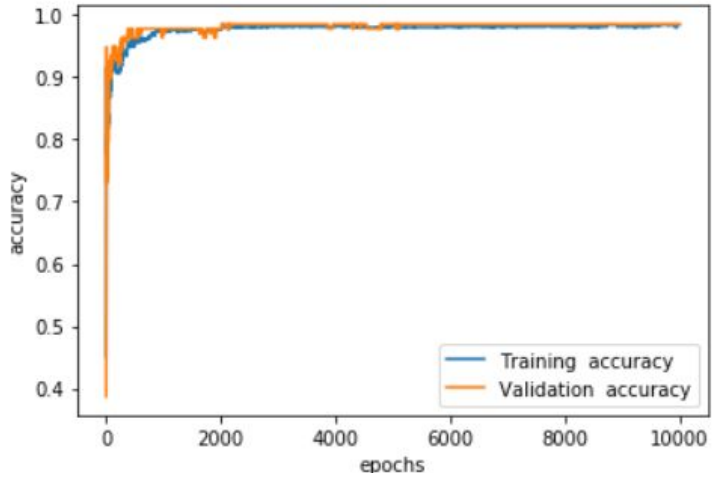
The data was further **normalised and shuffled** and then the logistic model was trained.

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**





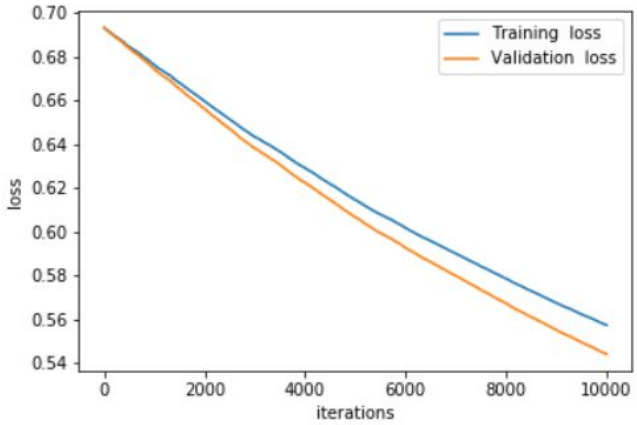
After experimenting from multiple learning rates and epochs I discovered that the losses stabilize after reaching a
training loss = 0.046893793198063305
validation loss = 0.03646555561191206

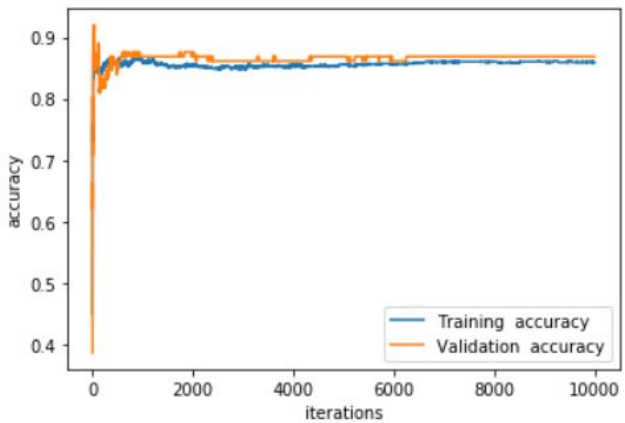
```
Training loss after  9500  sgd steps is :  0.046893793198063305  | validation loss is :  0.03646555561191206
Training accuracy after  9500  sgd steps is :  98.33159541188738 %  | validation accuracy is :  98.54014598540147 %
```

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**

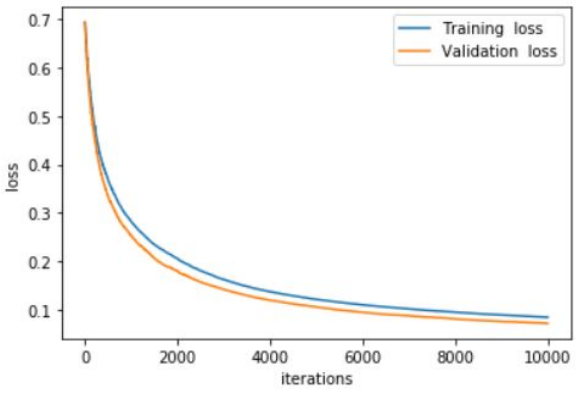


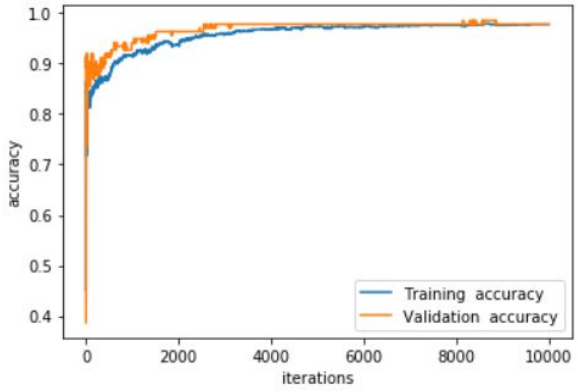


I ran 10k epochs for 0.0001 and it is clear that the loss didn't stabilize.

```
Training loss after  9500  sgd steps is :  0.562229660464482  | validation loss is :  0.5494383982239276
Training accuracy after  9500  sgd steps is :  86.02711157455683 %  | validation accuracy is :  86.86131386861314 %
```

2. **Learning rate = 0.01**

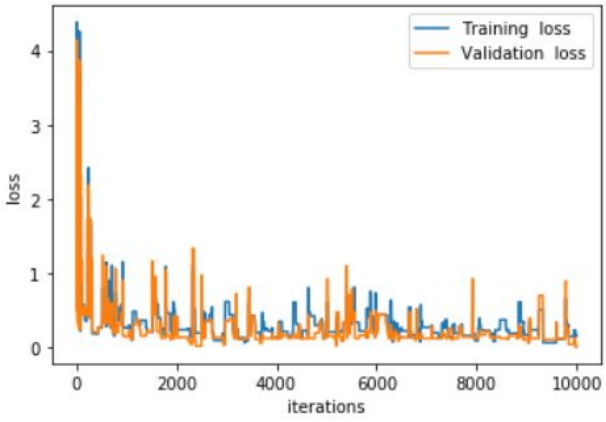


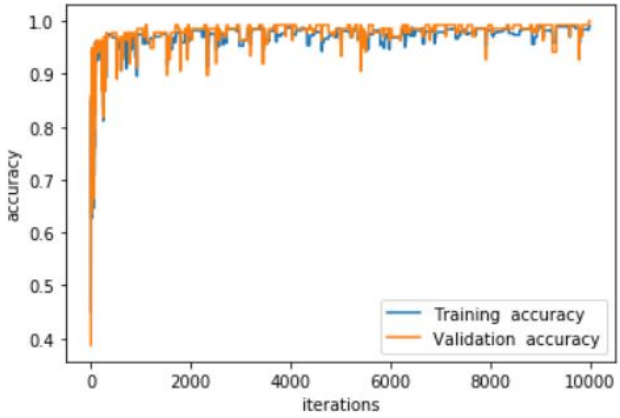


I ran 10k epochs and the final loss and accuracy is :

```
Training loss after  9500  sgd steps is :  0.08680050263742053  | validation loss is :  0.07386607737128432
Training accuracy after  9500  sgd steps is :  97.70594369134515 %  | validation accuracy is :  97.8102189781022 %
```

3. **Learning rate = 10**





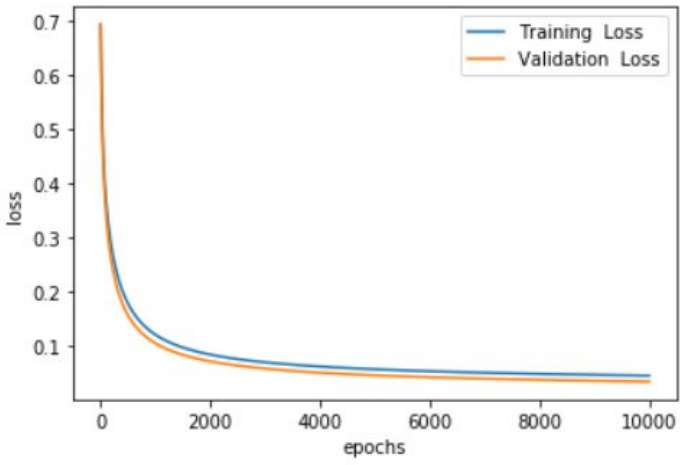
On running 10k epochs for 10 learning rate the loss never stabilised. The loss was fluctuating a lot which indicates that the gradient was jumping around the minimum due to the very high learning rate.

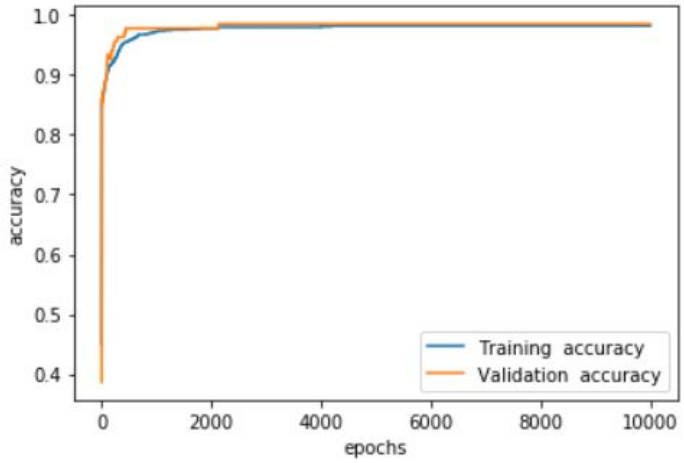
```
Training loss after  6500  sgd steps is :  0.21807244703308168  | validation loss is :  0.13039741072755137
Training accuracy after  6500  sgd steps is :  98.22732012513035 %  | validation accuracy is :  98.54014598540147 %

Training loss after  7000  sgd steps is :  0.26749417260031005  | validation loss is :  0.11767040970881494
Training accuracy after  7000  sgd steps is :  97.91449426485923 %  | validation accuracy is :  99.27007299270073 %

Training loss after  7500  sgd steps is :  0.12356692447439672  | validation loss is :  0.22257924678912
Training accuracy after  7500  sgd steps is :  98.1230448383733 %  | validation accuracy is :  97.8102189781022 %

Training loss after  8000  sgd steps is :  0.0875574000145561  | validation loss is :  0.11765038194847457
Training accuracy after  8000  sgd steps is :  98.22732012513035 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8500  sgd steps is :  0.36767549816311174  | validation loss is :  0.11765793508396706
Training accuracy after  8500  sgd steps is :  97.39311783107404 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9000  sgd steps is :  0.16163652397404019  | validation loss is :  0.1010983084256144
Training accuracy after  9000  sgd steps is :  98.85297184567257 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9500  sgd steps is :  0.059215721052834534  | validation loss is :  0.10056632201514765
Training accuracy after  9500  sgd steps is :  98.95724713242961 %  | validation accuracy is :  99.27007299270073 %
```

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**





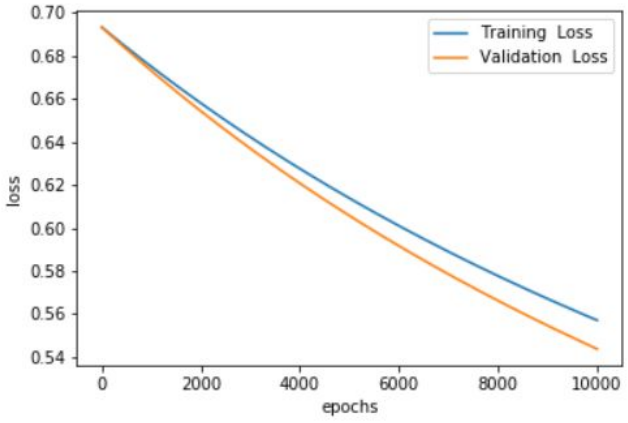
After experimenting from multiple learning rates and epochs I discovered that the losses stabilize after reaching a
training loss = 0.04612043040052663
validation loss = 0.03517665334231373

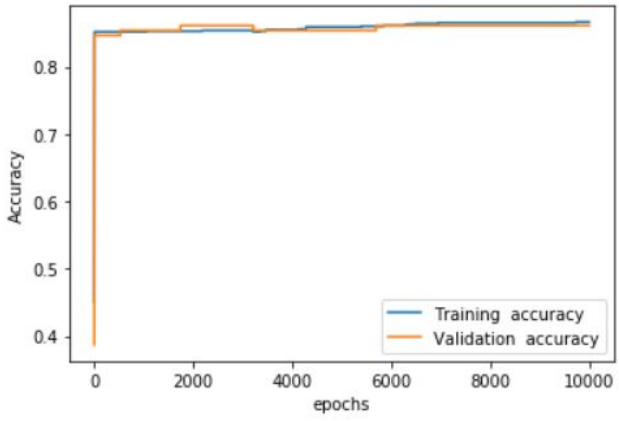
```
Training loss after  9500  bgd steps is :  0.04612043040052663  | validation loss is :  0.03517665334231373
Training accuracy after  9500  bgd steps is :  98.22732012513035 %  | validation accuracy is :  98.54014598540147 %
```

The Final accuracy achieved by the model :
Training Accuracy = 98.22732012513035 %
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**

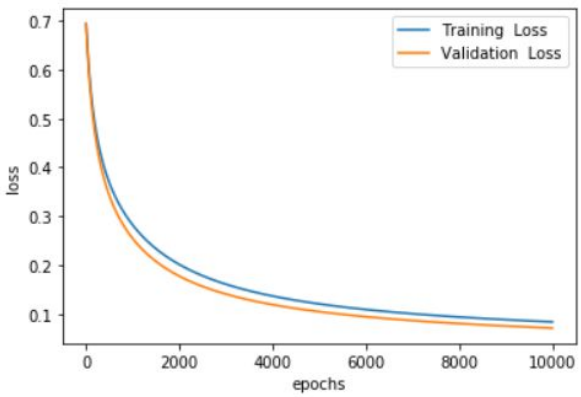


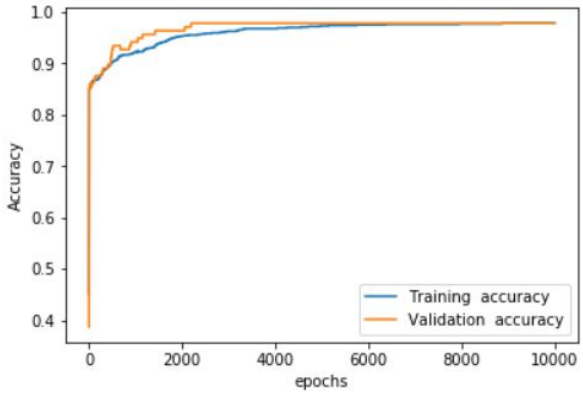


I ran 10k epochs for 0.0001 and it is clear that the loss didn't stabilize.

```
Training loss after  9500  bgd steps is :  0.5620198862304883  | validation loss is :  0.549117870198338
Training accuracy after  9500  bgd steps is :  86.54848800834203 %  | validation accuracy is :  86.13138686131387 %
```

2. **Learning rate = 0.01**

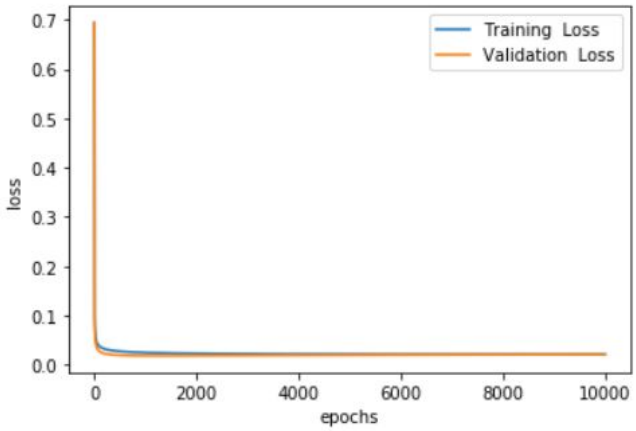


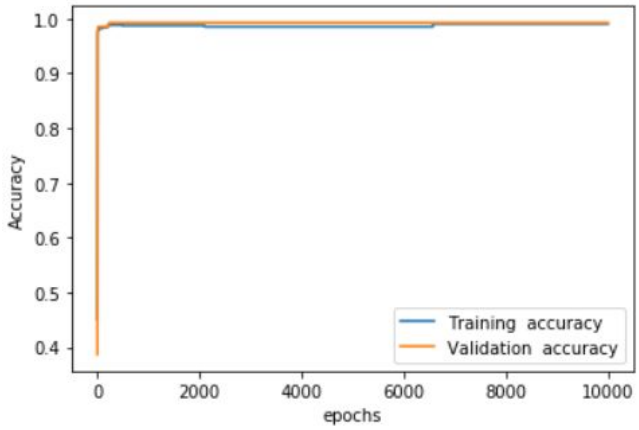


I ran 10k epochs and the final loss and accuracy is :

```
Training loss after  9500  bgd steps is :  0.08636137128425853  | validation loss is :  0.07362462408609635
Training accuracy after  9500  bgd steps is :  97.8102189781022 %  | validation accuracy is :  97.8102189781022 %
```

4. **Learning rate = 10**





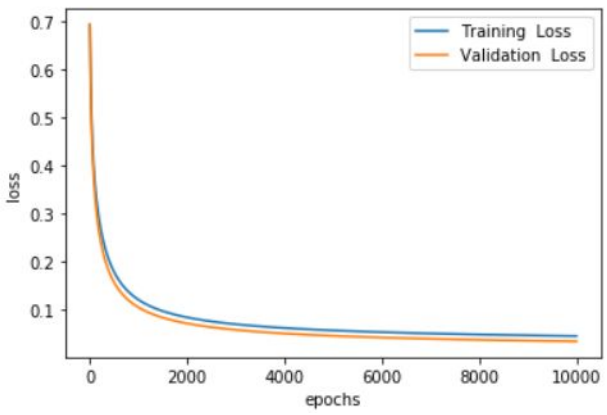
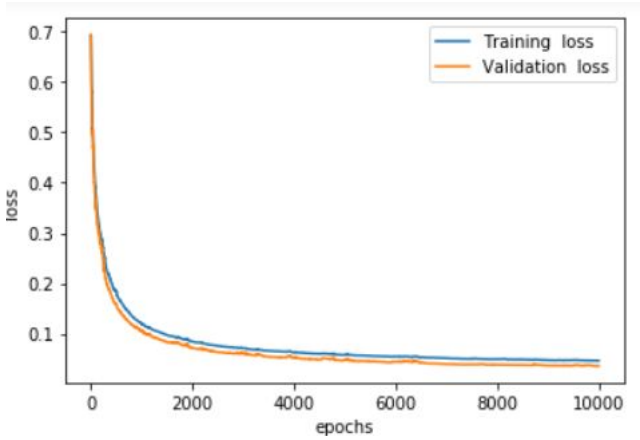
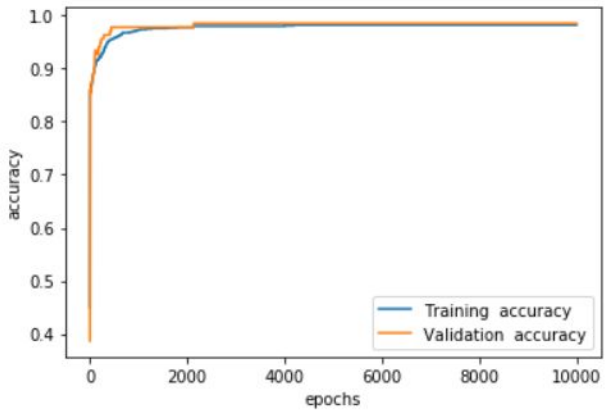
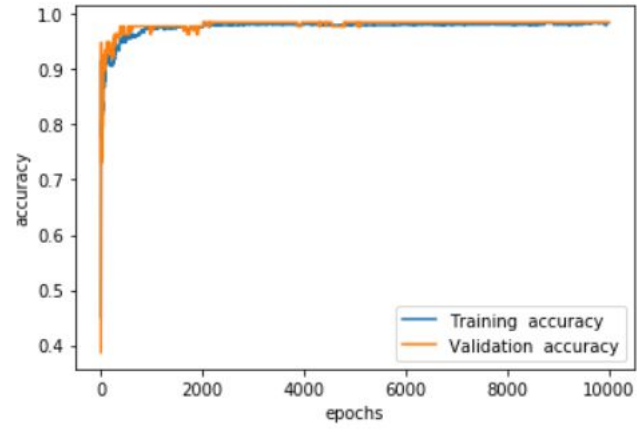
On running 10k epochs for 10 learning rate the loss stabilised extremely quickly.

```
Training loss after  7500  bgd steps is :  0.020981736207239702  | validation loss is :  0.02044147741646399
Training accuracy after  7500  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8000  bgd steps is :  0.02093985288356669  | validation loss is :  0.02055906193333234
Training accuracy after  8000  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  8500  bgd steps is :  0.020903804608373936  | validation loss is :  0.02066953502828161
Training accuracy after  8500  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9000  bgd steps is :  0.02087262567024995  | validation loss is :  0.02077341194206686
Training accuracy after  9000  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %

Training loss after  9500  bgd steps is :  0.02084554051772798  | validation loss is :  0.02087116728288216
Training accuracy after  9500  bgd steps is :  99.06152241918666 %  | validation accuracy is :  99.27007299270073 %
```

Comparison between SGD and BGD

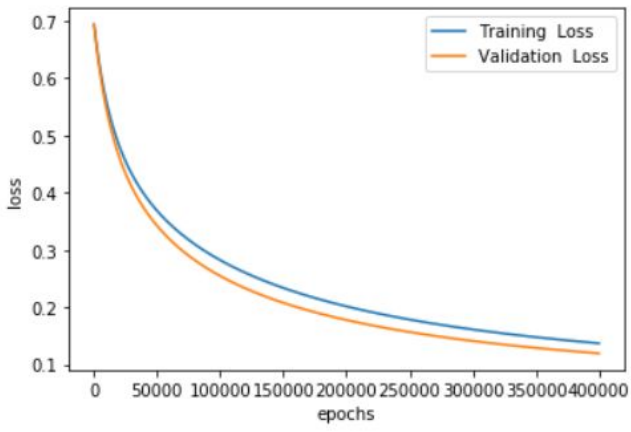
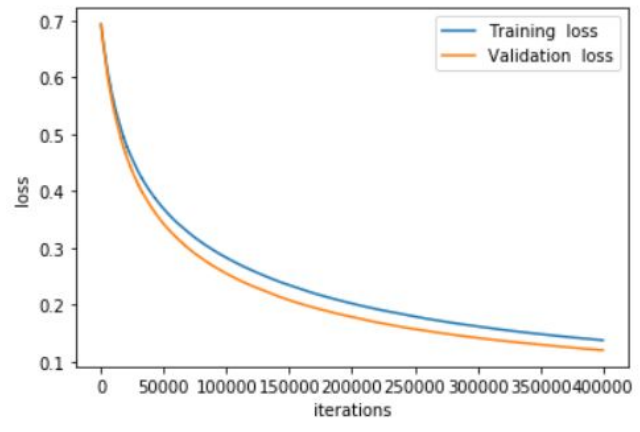
Part (a) and (b) : Loss plots and number of epochs to converge.

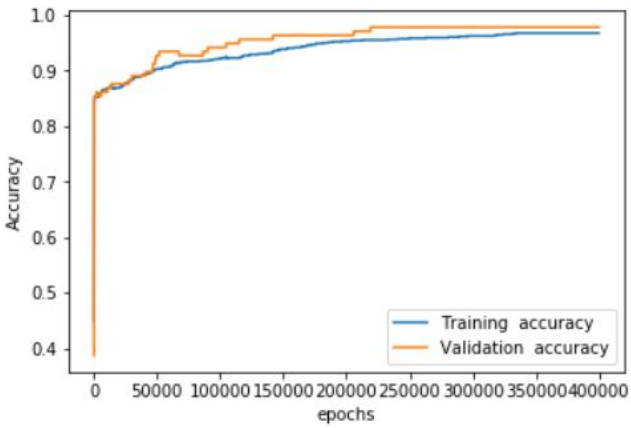
1. **Learning rate = 0.05**

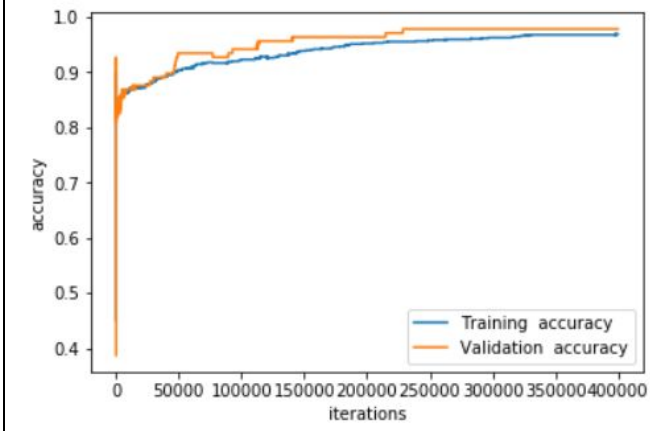
BGD	SGD
	
	

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

2. **Learning rate = 0.0001**

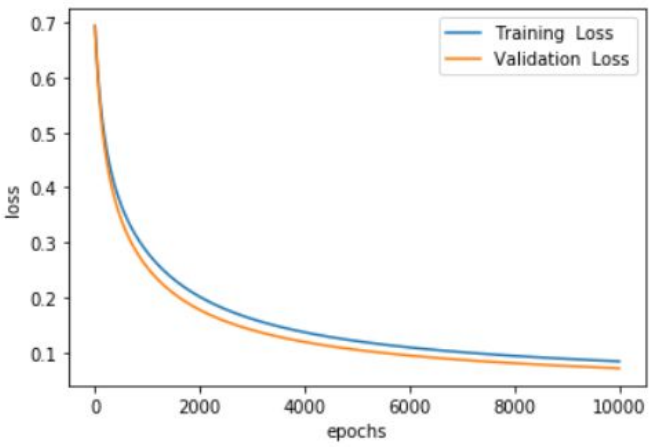
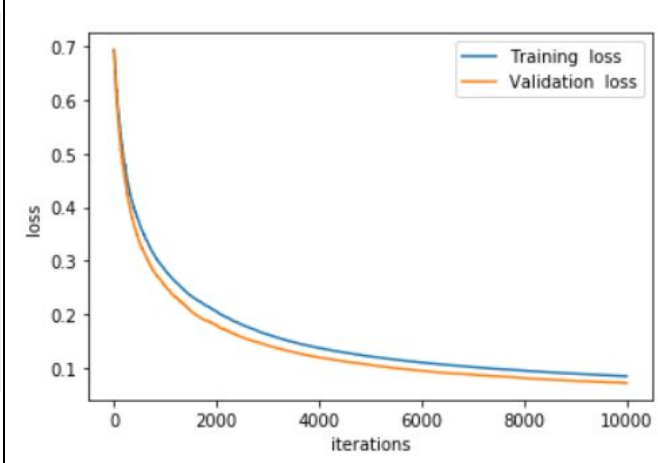
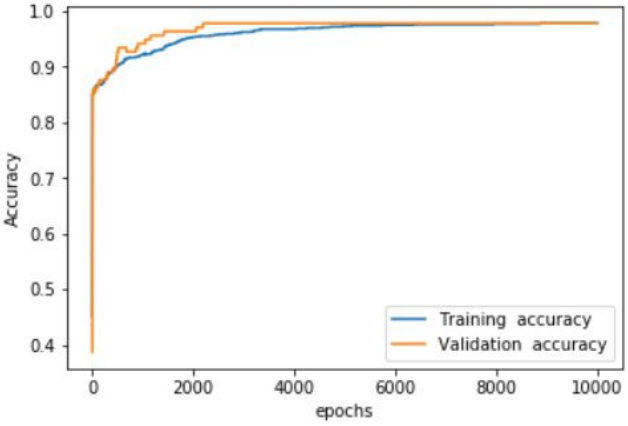
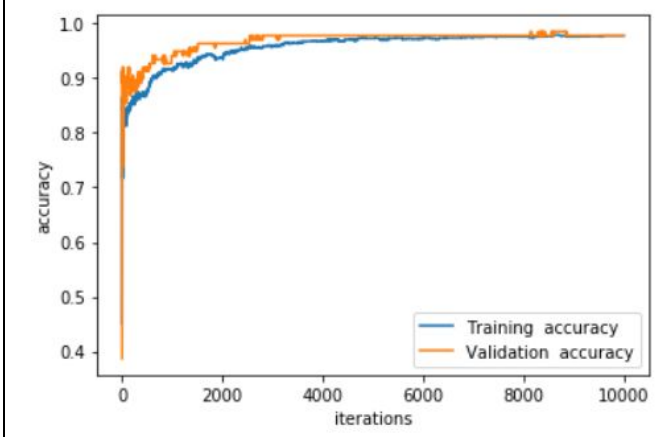
BGD	SGD
	





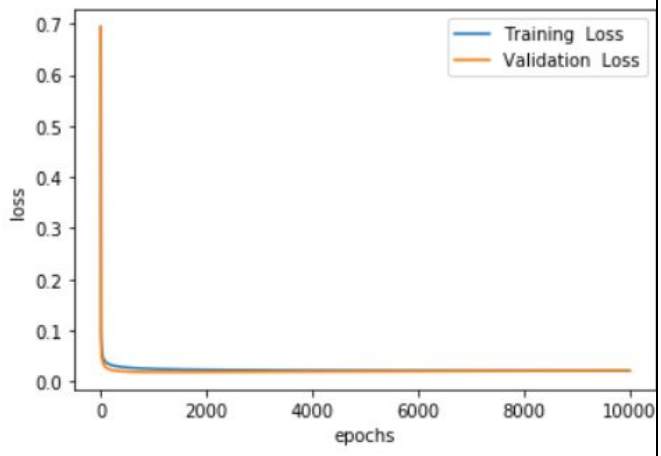
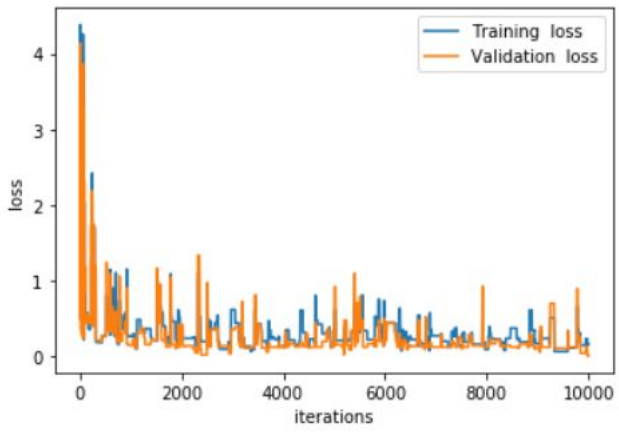
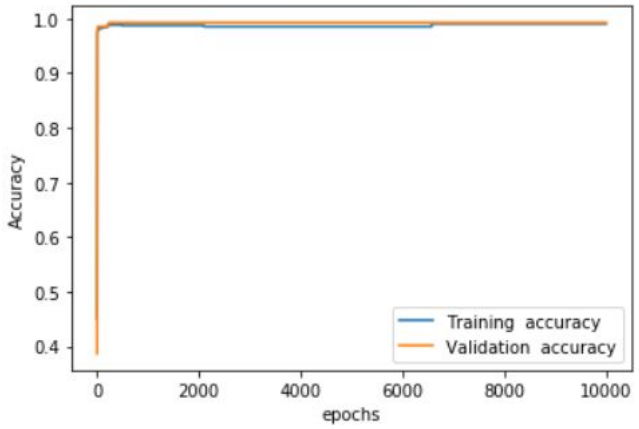
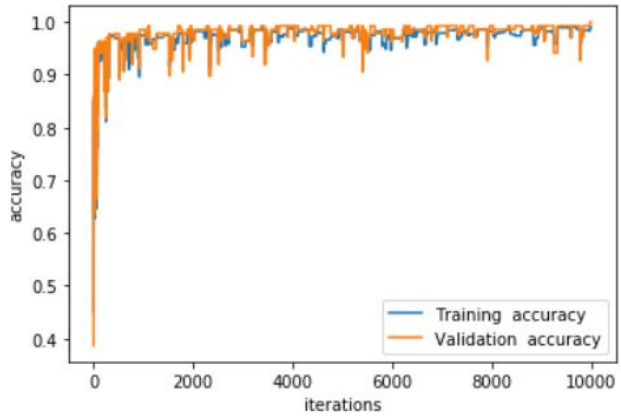
Both SGD and BGD converge > 400k epochs as seen from the above plots.

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 $y\_true = 1$ $y\_pred = 0$ or $y\_true = 0$ $y\_pred = 1$ $\therefore \hat{y} = \sigma(x\theta)$

Using MSE, we get an error of $\sum_{i=1}^{m} \frac{|y-\hat{y}|^2}{m}$

where m is the number of training ex.

$$\text{Loss} = (0-1)^2 = 1$$

where as when using binary cross entropy loss

$$\text{Loss} = -(1 * \log(0) + 0 * \log(1)) \longrightarrow \infty$$

— (3)

(I)

Now

Gradient when using MSE.

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

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

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

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

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 upward updation.

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

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

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

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} 1 & x_{21} & \cdots & x_{k1} \\ \vdots & & & \\ 1 & x_{2n} & \cdots & x_{kn} \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_k \end{pmatrix}, \quad \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

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

Sum of square errors $= \varepsilon_1^2 + \varepsilon_2^2 + \cdots \varepsilon_n^2$

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

Our goal is to minimize this loss.

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

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

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

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

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

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

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 (1), equation (2) becomes.

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

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

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

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

$$\Rightarrow 2X^T X\beta - 2X^T Y = 0$$

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

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.

QUESTION 4

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

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

↳ (Similar done in lectures)

Thus the likelihood of all data is

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

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

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

$$\log(L) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\sigma(\theta^T x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - \sigma(\theta^T x^{(i)})\right) \right]$$

↓

Loss fn for logistic regression (cross-entropy)

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

```
In [43]: log_regressor = MyLogisticRegression()
         log_regressor.fit(X,y, epochs = 10000 ,learning_rate = 0.005)

         Training loss after  7000  iterations is  0.5564252509189604
         Training accuracy after  7000  iterations is :  75.75757575757575 %

         Training loss after  7500  iterations is  0.556282101308868
         Training accuracy after  7500  iterations is :  75.75757575757575 %

         Training loss after  8000  iterations is  0.5561787962708594
         Training accuracy after  8000  iterations is :  75.75757575757575 %

         Training loss after  8500  iterations is  0.5561040241089631
         Training accuracy after  8500  iterations is :  75.75757575757575 %

         Training loss after  9000  iterations is  0.556049768388717
         Training accuracy after  9000  iterations is :  75.75757575757575 %

         Training loss after  9500  iterations is  0.5560103161634421
         Training accuracy after  9500  iterations is :  75.75757575757575 %

Out[43]: <scratch.MyLogisticRegression at 0x170250fec88>

In [44]: print(log_regressor.W)
         print(log_regressor.b)

         [[1.38835658]
          [0.73511168]]
         -0.3497166832331577

In [45]: print(np.exp(log_regressor.W))
         print(np.exp(log_regressor.b))

         [[4.00825738]
          [2.08571492]]
         0.7048877679546853
```

1. **Beta_2 = 0.73511168**
   Beta_1 = 1.38835658
   Beta_0 = -0.3497166832331577

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 : **[4.00825738] [2.08571492]]**
   a. **E1 = exp(beta_1)** 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 **4.00825738**
   b. times**.**
   c. **E2 = exp(beta_2)** signifies that with a unit increase in theage of the child the odds of disease occurring (y=1) to disease not occuring (y=0) increases by **2.08571492** 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 **0.8169816**

```
In [49]: X_test = np.array([[75,2]])
         print(X_test.shape)
         print(mean.shape)
         X_test = (X_test-mean)/std
         print(X_test)

         (1, 2)
         (2,)
         [[ 1.74336321 -0.78173596]]

In [51]: y_test = 1/(1 + np.exp(-(np.dot(X_test,log_regressor.W) + log_regressor.b)))
         print(y_test)

         [[0.8169816]]
```