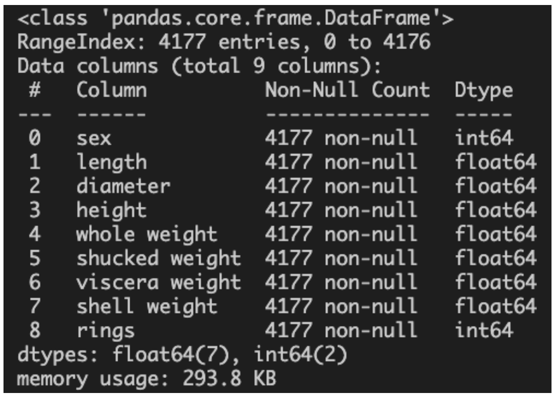
CSE 343: Machine Learning

Assignment 1 | Report

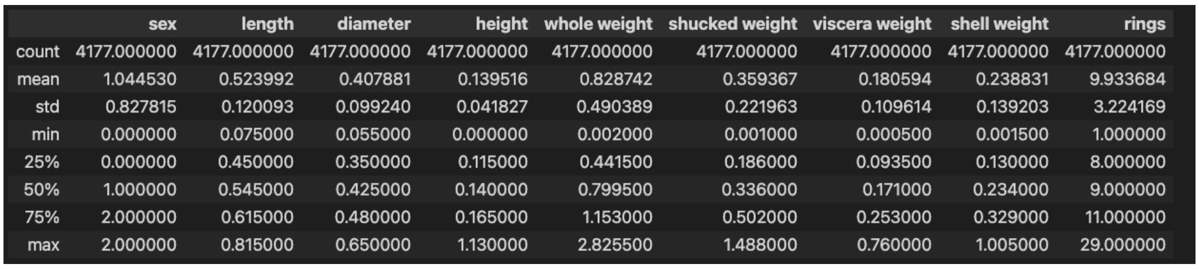
Question 1. Linear Regression

1. Loading the Dataset
   - The dataset was loaded and converted to DataFrame using the pandas library.
   - The Columns header was added and the following names were given to all the columns:
     `abaloneDF.columns = ['sex', 'length', 'diameter', 'height', 'whole weight', 'shucked weight', 'viscera weight', 'shell weight', 'rings']`
2. Preprocessing the Data
   - Firstly the unique values in the 'sex' column were found. Those were:

     `array(['F', 'I', 'M'], dtype=object)`
   - The above categorical data was converted to numeric data as per the following mapping:
     - I = 0
     - F = 1
     - M = 2
   - Basic info about the data was obtained:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4177 entries, 0 to 4176
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   sex             4177 non-null   int64  
 1   length          4177 non-null   float64
 2   diameter        4177 non-null   float64
 3   height          4177 non-null   float64
 4   whole weight    4177 non-null   float64
 5   shucked weight  4177 non-null   float64
 6   viscera weight  4177 non-null   float64
 7   shell weight    4177 non-null   float64
 8   rings           4177 non-null   int64  
dtypes: float64(7), int64(2)
memory usage: 293.8 KB
```



	sex	length	diameter	height	whole weight	shucked weight	viscera weight	shell weight	rings
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000
mean	1.044530	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.238831	9.933684
std	0.827815	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.139203	3.224169
min	0.000000	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.001500	1.000000
25%	0.000000	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.130000	8.000000
50%	1.000000	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.234000	9.000000
75%	2.000000	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.329000	11.000000
max	2.000000	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.005000	29.000000

   - The data was randomly divided into train and test sets in the ratio of 8:2

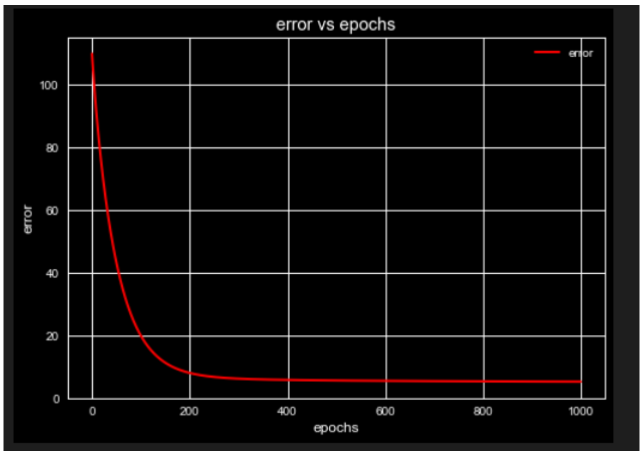
- The train and test sets were divided into X_train, y_train and X_test, y_test respectively such that first 8 columns were part of X and last column was part of y
- The data was then normalized such that each column had mean 0 and standard deviation 1
    - It was made sure that the test set was normalized using the mean and standard deviation of training data itself.

```
X_mean = np.mean(X_train, axis=0)
X_std = np.std(X_train, axis=0)
```

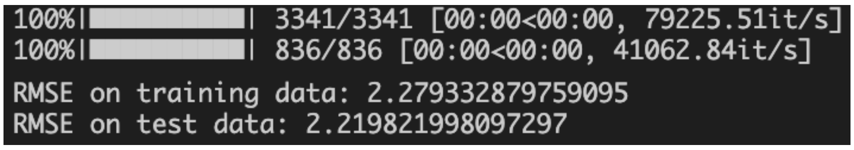
```
X_train = (X_train - X_mean) / X_std
X_test = (X_test - X_mean) / X_std
```

- A column of 1's was added in both the training and the test sets so as to denote X0

3. Performing Linear Regression
    - A set of functions were written to calculate:
        - hypothesis
        - rmse
        - gradient
        - gradient descent
    - Linear regression was performed using Batch Gradient Descent and theta vector and errorlist corresponding to each epoch was obtained
    - The error list was plotted against the number of epochs:



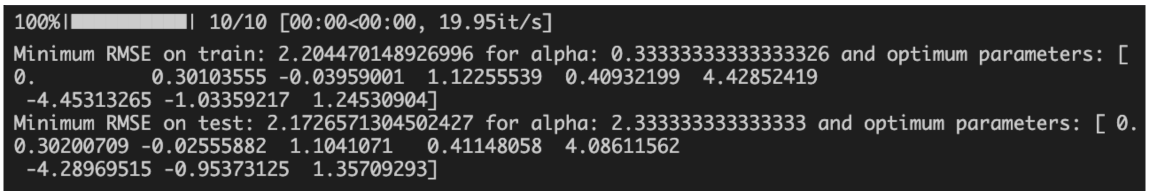
- To get the predictions following functions were written:
    - getPredictionList
    - rmse
- Using the above functions, RMSE on training and testing data was calculated



```
100%|██████████| 3341/3341 [00:00<00:00, 79225.51it/s]
100%|██████████| 836/836 [00:00<00:00, 41062.84it/s]
RMSE on training data: 2.279332879759095
RMSE on test data: 2.219821998097297
```

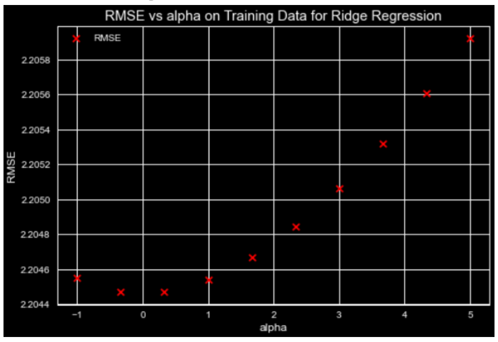
4. Ridge Regression
    - To use Ridge Regression, sklearn library was used
    - Function to calculate RMSE error for ridge regression was written
    - Minimum RMSE and its corresponding alpha and theta values were stored. They came

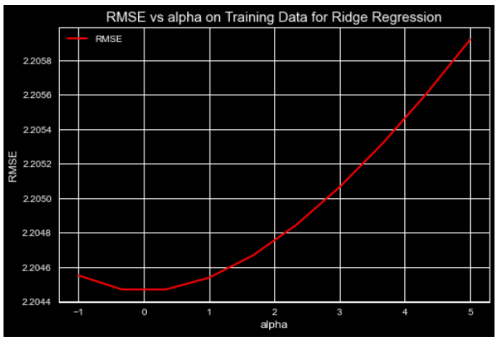
out to be:



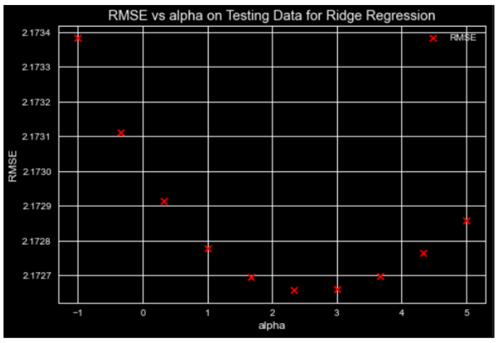
```
100%|██████████| 10/10 [00:00<00:00, 19.95it/s]
Minimum RMSE on train: 2.204470148926996 for alpha: 0.33333333333333326 and optimum parameters: [
0.          0.30103555 -0.03959001  1.12255539  0.40932199  4.42852419
 -4.45313265 -1.03359217  1.24530904]
Minimum RMSE on test: 2.1726571304502427 for alpha: 2.333333333333333 and optimum parameters: [ 0.
0.30200709 -0.02555882  1.1041071   0.41148058  4.08611562
 -4.28969515 -0.95373125  1.35709293]
```

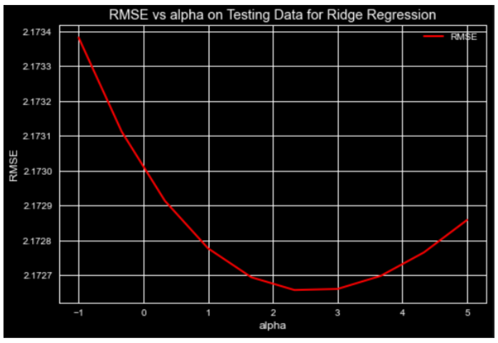
- Corresponding scatter and line plots were also plotted.
    - Training Data:



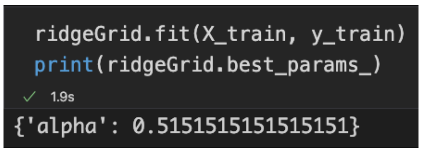


    - Test Data:





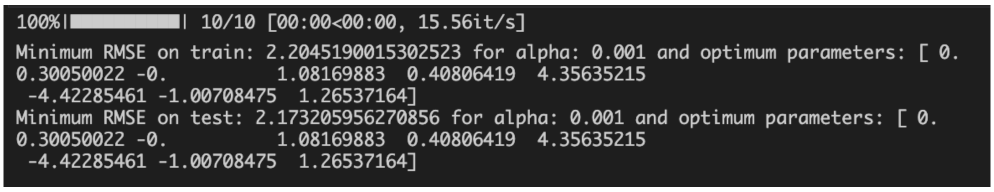
- For the above data, 10 different alpha values were takes which were equally distributed in the range of -1 to 5.
- The most optimum alpha was also calculated using sklearn's GridSearchCV function.
- For the same, the alphas used were equally distributed over 100 different values between -1 and 4, and the training data was used.
- The optimum received was:



```
ridgeGrid.fit(X_train, y_train)
print(ridgeGrid.best_params_)
✓ 1.9s
{'alpha': 0.5151515151515151}
```

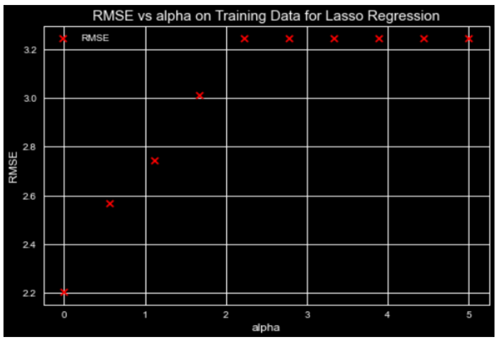
- The best model coefficients in the above are close but not the same as the values of alpha used in Ridge regression were more than those done manually.

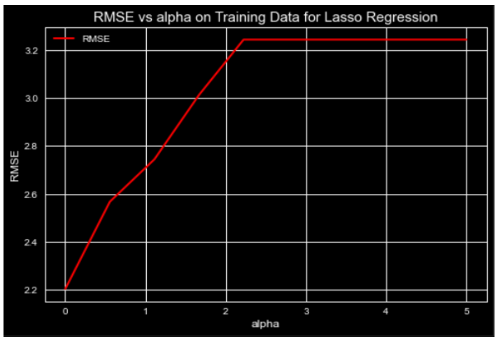
5. Lasso Regression
    - To use Lasso Regression, sklearn library was used
    - Function to calculate RMSE error for ridge regression was written
    - Minimum RMSE and its corresponding alpha and theta values were stored. They came out to be:



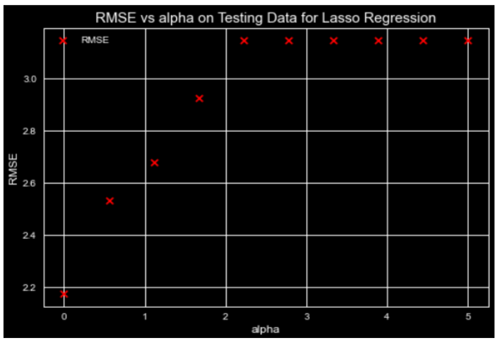
```
100%|██████████| 10/10 [00:00<00:00, 15.56it/s]
Minimum RMSE on train: 2.2045190015302523 for alpha: 0.001 and optimum parameters: [ 0.
0.30050022 -0.          1.08169883  0.40806419  4.35635215
 -4.42285461 -1.00708475  1.26537164]
Minimum RMSE on test: 2.173205956270856 for alpha: 0.001 and optimum parameters: [ 0.
0.30050022 -0.          1.08169883  0.40806419  4.35635215
 -4.42285461 -1.00708475  1.26537164]
```

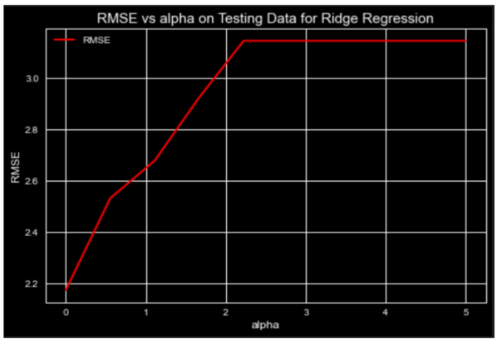
- Corresponding scatter and line plots were also plotted
  - Training Data:



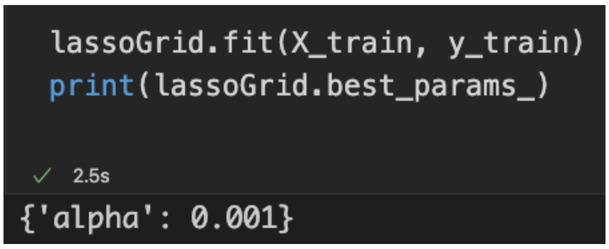


  - Testing Data:





- For the above data, 10 different alpha values were takes which were equally distributed in the range of 0.01 to 5.
- The most optimum alpha was also calculated using sklearn’s GridSearchCV function.
- For the same, the alphas used were equally distributed over 100 different values between -1 and 4, and the training data was used.
- The optimum received was:



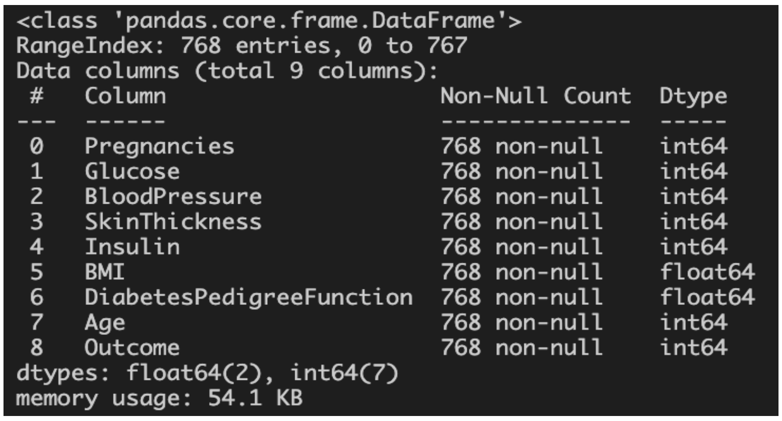
```
lassoGrid.fit(X_train, y_train)
print(lassoGrid.best_params_)

✓ 2.5s
{'alpha': 0.001}
```

6. Analysis:
   - Ridge Regression:
     - The best model coefficients varied a little as the number of alpha values being fed in the loop and the number in the gridsearch function were different.
     - The values in the gridsearch were more, hence getting a more accurate value.
     - Some other factors like some changes in the gridsearch function to look for the final values might have also played a major role.
     - Moreover, the values obtained for test data in manual alpha selection, and the one

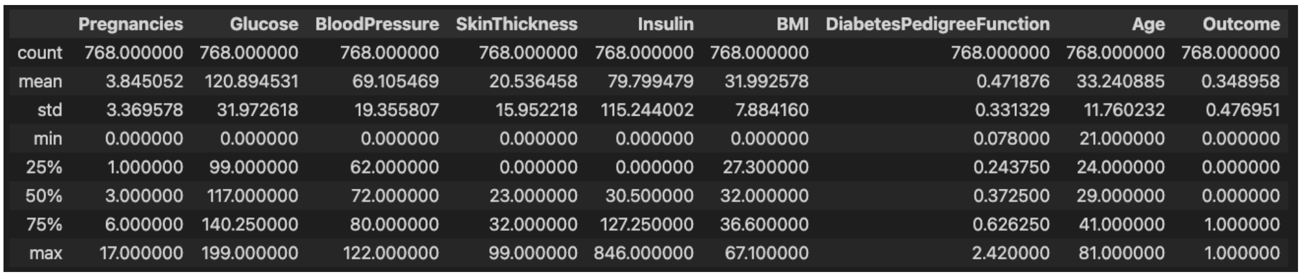
    given by the gridsearchcv function are different as for the gridsearch function, it had never seen the test data.
    - Also the errors calculated for the test data also were in respect of the alpha values for training data. This might have been a huge factor while showing the final alpha values.
- Lasso Regression
    - All three alpha values obtained (manual testing, manual training and the one from gridsearchcv) are 0.001
    - This might indicate that the function has not converged for the given alpha values.
    - However, when it was tried to take the parameters on the other side of y axis, more errors popped up as the lasso regression could not porcess -ve values effectively.

Question 2. Logistic Regression

1. Loading the dataset
   - The dataset was loaded and converted to DataFrame using the pandas library.
2. Preprocessing the data
   - Basic info about the data was obtained.



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

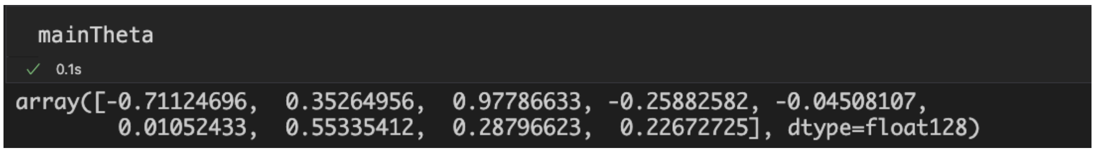
   - The data was randomly divided into training, validation and testing datasets in the ratio of 7:2:1
   - All the parameters were converted to np.float128 datatype
   - The train and test sets were divided into X_train, y_train, X_validation, Y_validation and X_test, y_test respectively such that first 8 columns were part of X and last column was part of y
   - The data was then normalized such that each column had mean 0 and standard deviation 1
     - It was made sure that the test and validation sets were normalized using the mean and standard deviation of training data itself.

```
X_mean = np.mean(X_train, axis=0)
X_std = np.std(X_train, axis=0)
```

```
X_train = (X_train - X_mean)/X_std
X_validation = (X_validation - X_mean)/X_st
X_test = (X_test - X_mean)/X_std
```

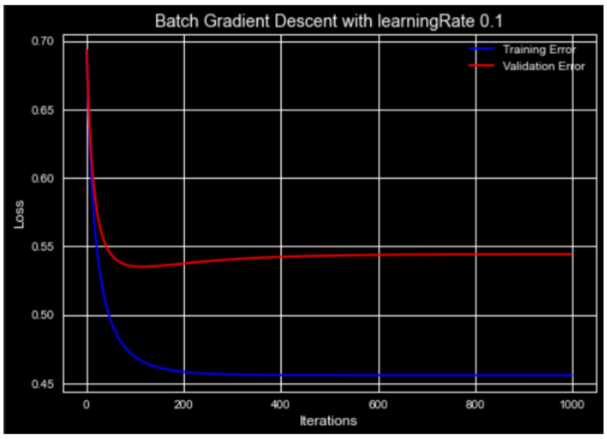
- A column of 1's was added in both the training and the test sets so as to denote X0

3. Performing Logistic Regression
   - Certain functions were made to calculate:
     - sigmoid
     - hypothesis
     - error
     - gradient
4. Batch Gradient Descent(BGD)
   - Batch Gradient Descent was performed
   - The final theta parameters as reported by the model are:

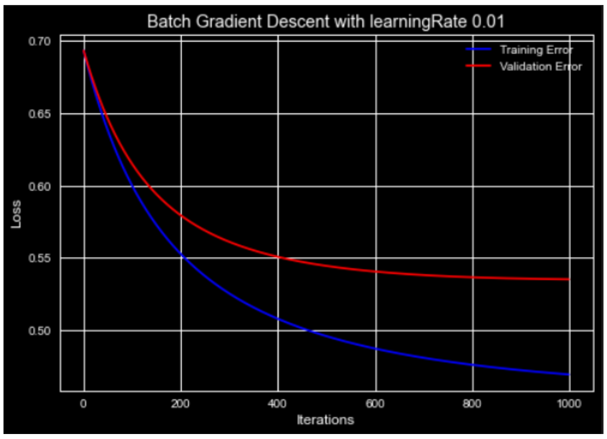


```
mainTheta
✓ 0.1s
array([-0.71124696,  0.35264956,  0.97786633, -0.25882582, -0.04508107,
        0.01052433,  0.55335412,  0.28796623,  0.22672725], dtype=float128)
```

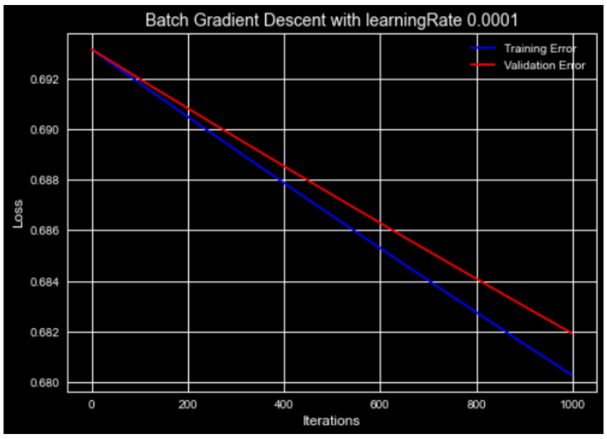
   - A function to draw plots for both training and validation data for the BGD performed was written
   - Plots for various learning rates were as follows:
     - Learning Rate = 0.1



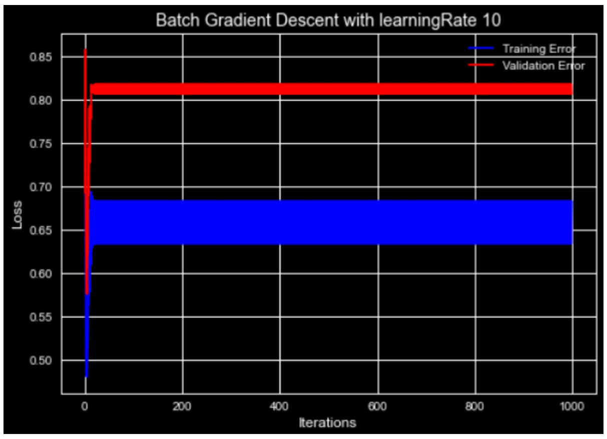
     - Learning Rate = 0.01



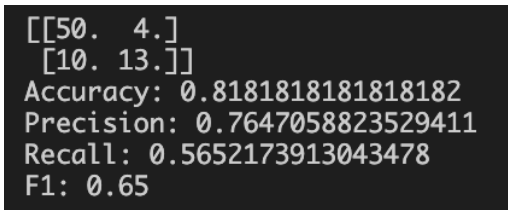
     - Learning Rate = 0.001



- Learning Rate = 10

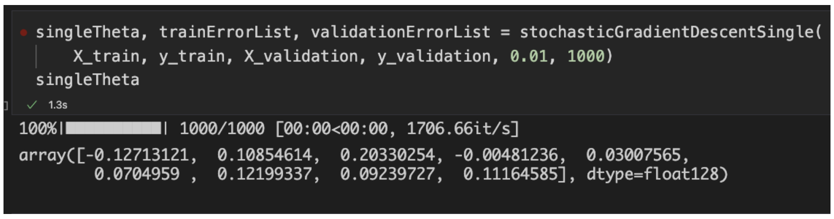


- Functions to get the following were written:
  - prediction list
  - Confusion matrix, accuracy, precision, recall and F1 scroe
- Confusion matrix, accuracy, precision, recall and F1 score for the BGD were:



```
[[50.  4.]
 [10. 13.]]
Accuracy: 0.8181818181818182
Precision: 0.7647058823529411
Recall: 0.5652173913043478
F1: 0.65
```

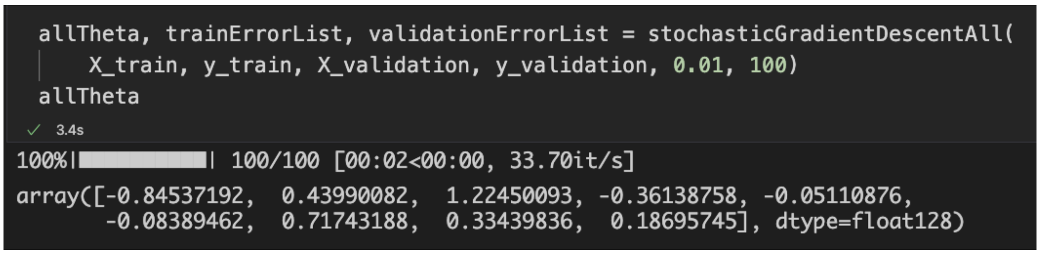
5. Stochastic Gradient Descent
   - For performing stochastic gradient descent, 2 functions were created.
     - StochasticGradientDescentSingle(): It randomly picked 1 value from the entire training dataset for each epoch and updated the theta correspondingly
     - StochasticGradientDescentAll(): For each epoch, it iterated over all datapoints once and updated theta value corresponding to each datapoint
   - The final theta values for both the implementations are as follows:
     - For StochasticGradientDescentSingle():



```
singleTheta, trainErrorList, validationErrorList = stochasticGradientDescentSingle(
    X_train, y_train, X_validation, y_validation, 0.01, 1000)
singleTheta
```

```
1.3s
100%|██████████| 1000/1000 [00:00<00:00, 1706.66it/s]
array([-0.12713121,  0.10854614,  0.20330254, -0.00481236,  0.03007565,
        0.0704959 ,  0.12199337,  0.09239727,  0.11164585], dtype=float128)
```

- For StochasticGradientDescentAll():

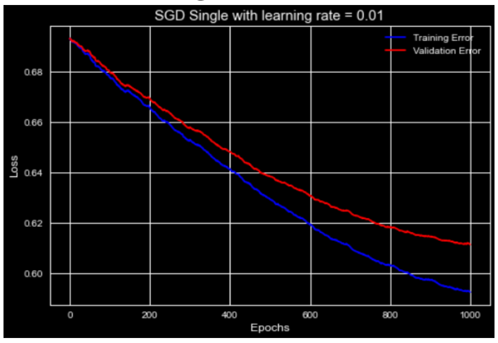


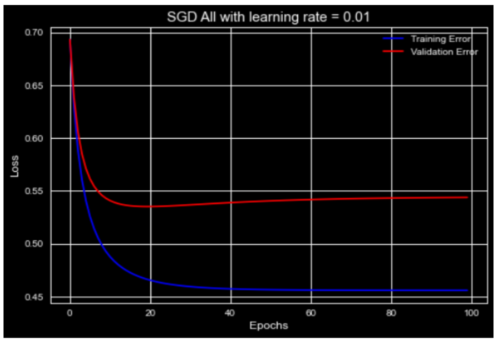
```
allTheta, trainErrorList, validationErrorList = stochasticGradientDescentAll(
    X_train, y_train, X_validation, y_validation, 0.01, 100)
allTheta
```

```
✓ 3.4s
```

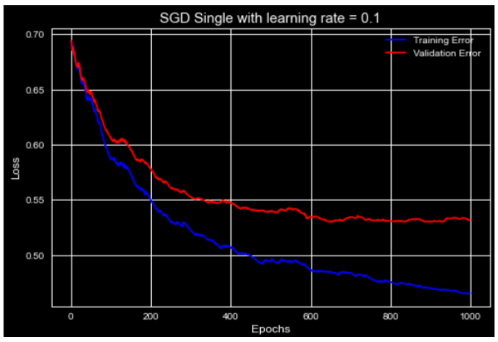
```
100%|██████████| 100/100 [00:02<00:00, 33.70it/s]
array([-0.84537192,  0.43990082,  1.22450093, -0.36138758, -0.05110876,
       -0.08389462,  0.71743188,  0.33439836,  0.18695745], dtype=float128)
```

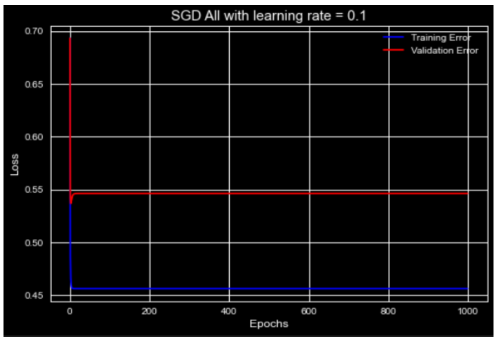
- Functions to draw plots for both training and validation data for both the implementations of the SGD was written
- Plots for various learning rates were as follows:
  - Learning Rate = 0.01



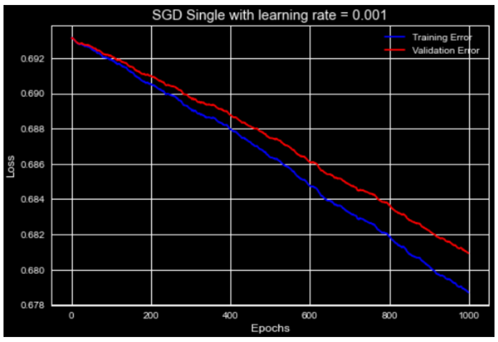


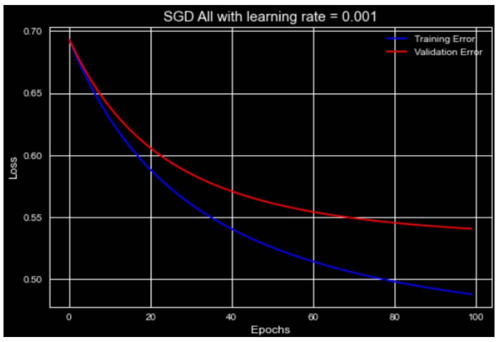
  - Learning Rate = 0.1



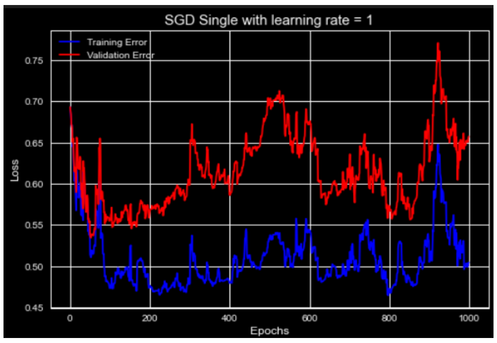


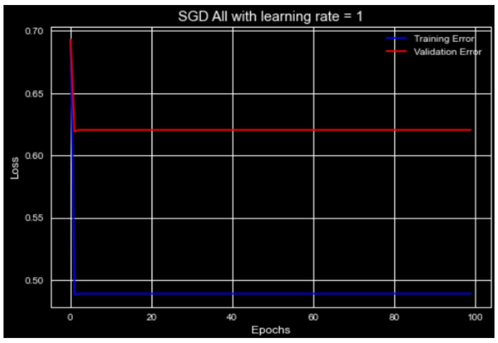
  - Learning Rate = 0.001



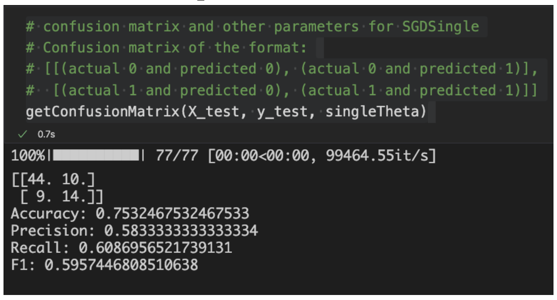


  - Learning Rate = 1

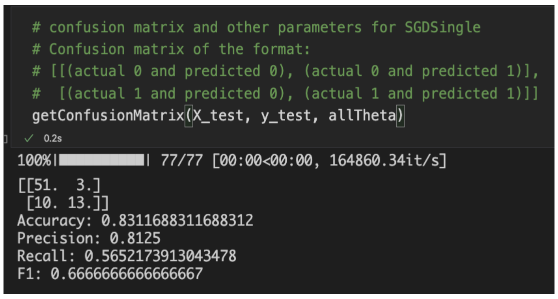




- confusion matrix for both the implementations were as follows:

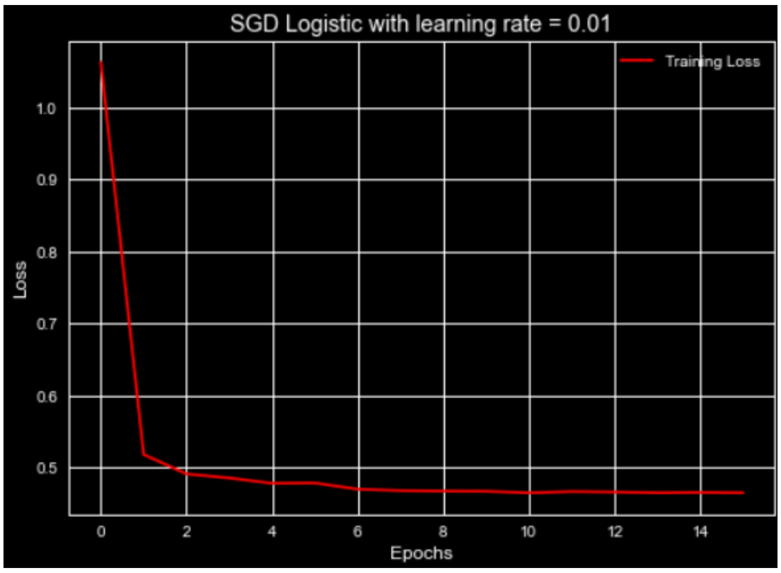


```
# confusion matrix and other parameters for SGDSingle
# Confusion matrix of the format:
# [[(actual 0 and predicted 0), (actual 0 and predicted 1)],
#  [(actual 1 and predicted 0), (actual 1 and predicted 1)]]
getConfusionMatrix(X_test, y_test, singleTheta)
✓ 0.7s
100%|██████████| 77/77 [00:00<00:00, 99464.55it/s]
[[44. 10.]
 [ 9. 14.]]
Accuracy: 0.7532467532467533
Precision: 0.5833333333333334
Recall: 0.6086956521739131
F1: 0.5957446808510638
```

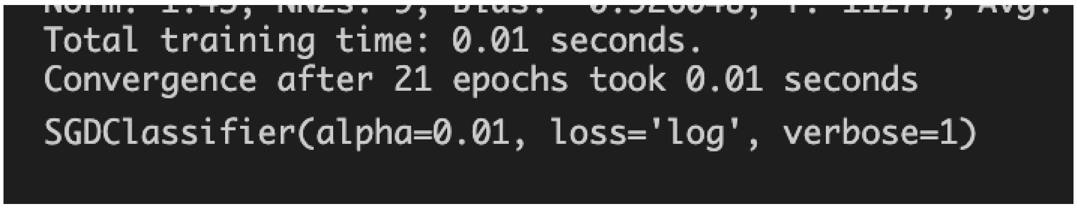


```
# confusion matrix and other parameters for SGDSingle
# Confusion matrix of the format:
# [[(actual 0 and predicted 0), (actual 0 and predicted 1)],
#  [(actual 1 and predicted 0), (actual 1 and predicted 1)]]
getConfusionMatrix(X_test, y_test, allTheta)
✓ 0.2s
100%|██████████| 77/77 [00:00<00:00, 164860.34it/s]
[[51.  3.]
 [10. 13.]]
Accuracy: 0.8311688311688312
Precision: 0.8125
Recall: 0.5652173913043478
F1: 0.6666666666666667
```

6. Sklearn's Implementation
   - Sklearn's implementation for creating SGD classifier was used
   - The loss values were stored in history of the classifier.
   - Those were extracted using StringIO, reference for the same was taken from the following: https://stackoverflow.com/questions/54388648/sgdclassifier-save-loss-from-every-iteration-to-array%22%22
   - The plot was:

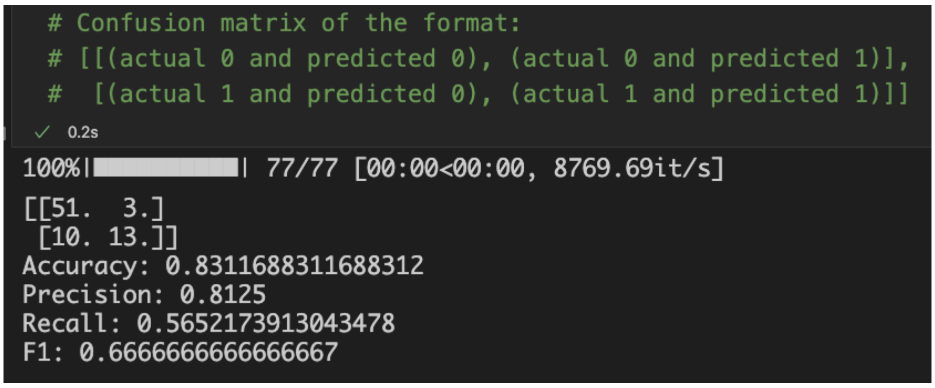


- The sklearn classifier took only about 21 epochs to reach convergence



```
Total training time: 0.01 seconds.
Convergence after 21 epochs took 0.01 seconds
SGDClassifier(alpha=0.01, loss='log', verbose=1)
```

- The confusion matrix for the same was obtained to be as follows:



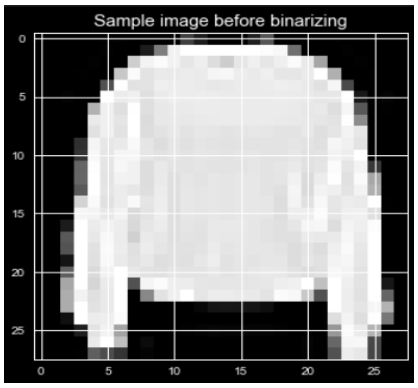
```
# Confusion matrix of the format:
# [[(actual 0 and predicted 0), (actual 0 and predicted 1)],
#  [(actual 1 and predicted 0), (actual 1 and predicted 1)]]
✓ 0.2s
100%|██████████| 77/77 [00:00<00:00, 8769.69it/s]
[[51.  3.]
 [10. 13.]]
Accuracy: 0.8311688311688312
Precision: 0.8125
Recall: 0.5652173913043478
F1: 0.6666666666666667
```

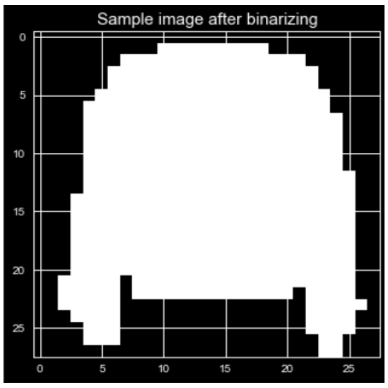
7. Analysis
    - For the training plots in both SGD and BGD, a difference between the test and the train data lines ca be observed. This was expected as the model has seen the training data but not the test data.
    - BGD
        - A smaller learning rate (0.001) shows that the speed for learning is too slow, thus giving something like a straight line.
        - Learning rates like 0.1 and 0.01 work fine
        - Learning rates like 10 are so big that the model overshoots the minima thus giving the loss on the higher side.
        - A more larger learning rate was not possible as it made the sigmoid function of the intdeterminate form 1/0.
    - SGD
        - Two implementations for SGD are presented.

    - Certain literatures talk about SGD changing its theta value once per epoch, while the others talk about it changing the theta values as many times as the number of training points in the data for each epoch.
    - Hence both are implemented by the name of `stochasticGradientDescentAll()` and `stochasticGradientDescentAll() respectively.`
    - For the first one, the values like 0.01 and 0.001 are too slow, making 0.1 as the perfect one among the lot.
    - For values like 1, the loss plots again start go haywire because of the issue discussed above.
    - For the latter, 0.01 and 0.001 works perfectly, while value like 0.1 and 1 gives the result almost immediately, but with higher difference between training and testing loss.
  - Sklearn's implementations
    - SGD classifier from sklearn is used.
    - We see that the classifier takes no more than 21 epochs to converge at alpha value of 0.01.
    - Our second SGD also worked in similar manner, while the first one almost took 10-50 times the epochs to converge for similar alpha values.

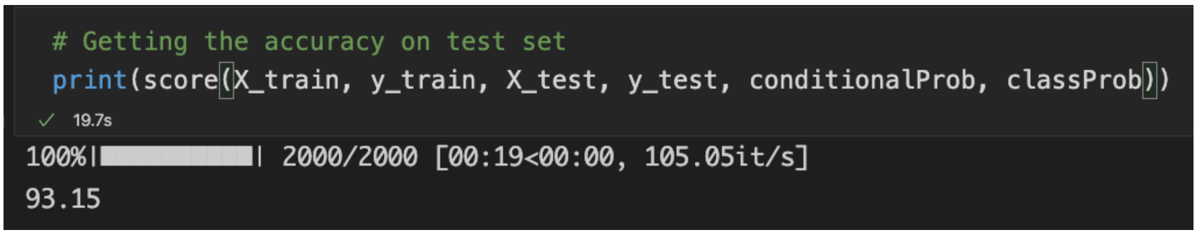
Question 3. Naïve Bayes

1. Loading the dataset
    - To load the dataset, the function available in the github repo provided to us was used.
    - The data was directly loaded in X and y, training and testing divisions
    - Copy of the above loaded dataset was created with the same variable names so as to make the data editable
2. Preprocessing the Dataset
    - From the github repo, it was found that the labels for trousers and pullovers were 1 and 2
    - Using this information at hand, the complete dataset was pruned to remove the data having any other label other than 1 and 2
    - For ease in the future tasks, the labels 1, and 2 were converted to 0 and 1 respectively.
    - The data had to be binarized as given in the problem statement
    - To do this, the threshold was fixed at 128.
    - All the values above 128 were made 255 and all those under it were made 0
    - Sample images before and after binarizing:



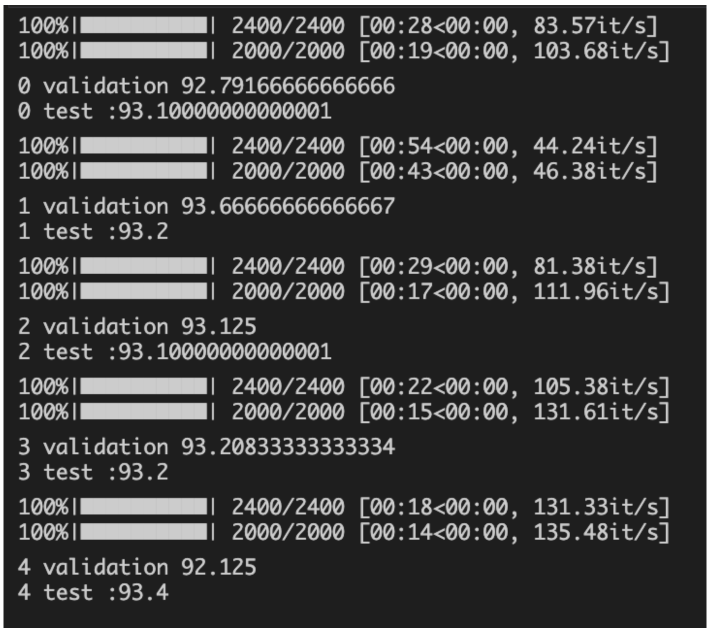


3. Implementing Naïve Bayes
    - Certain functions were made to compute the following:
        - Prior Probability
        - Conditional Probability
        - List of all prior probabilities and conditional probabilities
        - Prediction Labels
        - Accuracy of the model
    - Naïve Bayes was performed on the training data and the class probabilities and conditional probabilities were obtained
    - Using the above, the model was run on testing set and the following accuracy was obtained



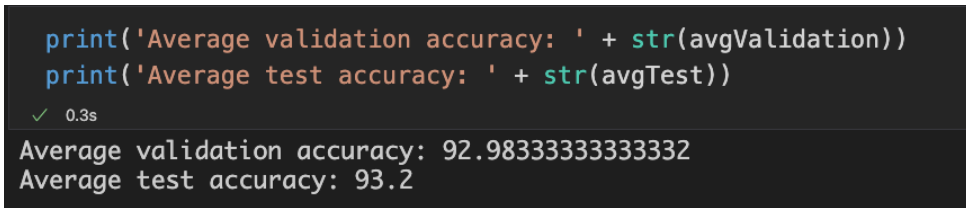
```
# Getting the accuracy on test set
print(score(X_train, y_train, X_test, y_test, conditionalProb, classProb))
✓ 19.7s
100%|██████████| 2000/2000 [00:19<00:00, 105.05it/s]
93.15
```

4. K-Fold Cross-Validation
   - the value of K was taken to be 5
   - the data was shuffled and then broken down into 5 sets of equal length
   - the above made model was trained again every time taking 4 out of 5 sets and taking the remaining as validation data
   - every time the model was retrained and accuracy for both validation and testing data was calculated



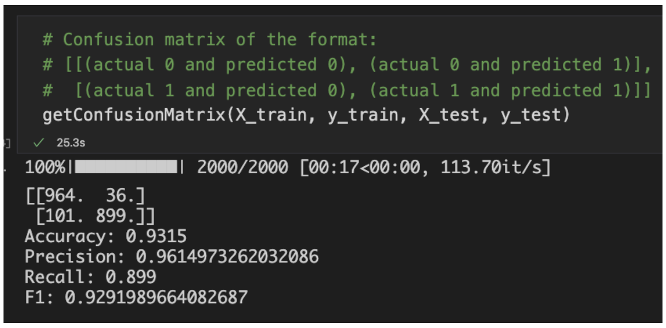
```
100%|██████████| 2400/2400 [00:28<00:00, 83.57it/s]
100%|██████████| 2000/2000 [00:19<00:00, 103.68it/s]
0 validation 92.79166666666666
0 test :93.10000000000001
100%|██████████| 2400/2400 [00:54<00:00, 44.24it/s]
100%|██████████| 2000/2000 [00:43<00:00, 46.38it/s]
1 validation 93.66666666666667
1 test :93.2
100%|██████████| 2400/2400 [00:29<00:00, 81.38it/s]
100%|██████████| 2000/2000 [00:17<00:00, 111.96it/s]
2 validation 93.125
2 test :93.10000000000001
100%|██████████| 2400/2400 [00:22<00:00, 105.38it/s]
100%|██████████| 2000/2000 [00:15<00:00, 131.61it/s]
3 validation 93.20833333333334
3 test :93.2
100%|██████████| 2400/2400 [00:18<00:00, 131.33it/s]
100%|██████████| 2000/2000 [00:14<00:00, 135.48it/s]
4 validation 92.125
4 test :93.4
```

   - The average test and validation accuracies over all K validation sets were also calculated



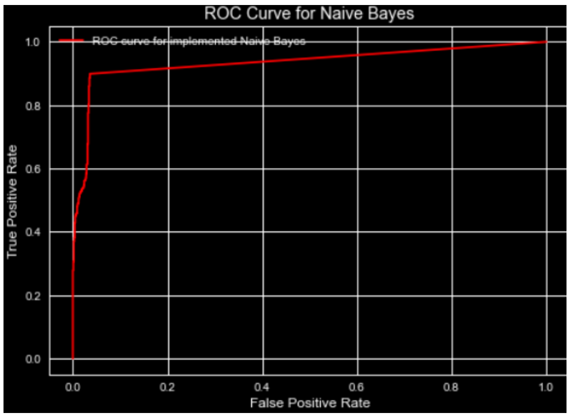
```
print('Average validation accuracy: ' + str(avgValidation))
print('Average test accuracy: ' + str(avgTest))
✓ 0.3s
Average validation accuracy: 92.98333333333332
Average test accuracy: 93.2
```

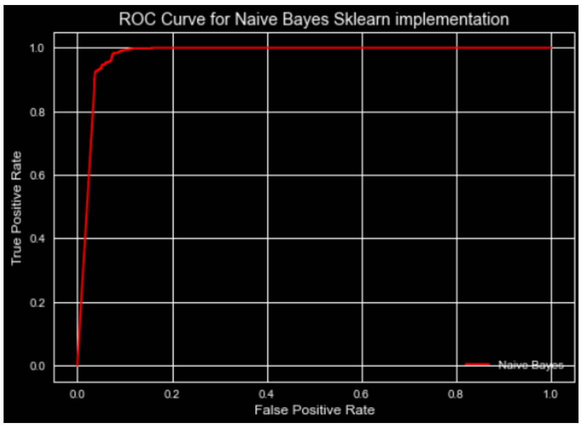
5. Confusion Matrix:
   - Confusion matrix was calculated over the entire training data and it was used to calculate precision, recall, accuracy and F1 score



```
# Confusion matrix of the format:
# [[(actual 0 and predicted 0), (actual 0 and predicted 1)],
#  [(actual 1 and predicted 0), (actual 1 and predicted 1)]]
getConfusionMatrix(X_train, y_train, X_test, y_test)
✓ 25.3s
100%|██████████| 2000/2000 [00:17<00:00, 113.70it/s]
[[964.  36.]
 [101. 899.]]
Accuracy: 0.9315
Precision: 0.9614973262032086
Recall: 0.899
F1: 0.9291989664082687
```

- For the ROC, curve for both the self implemented and the sklearn generated model are generated.





6. Analysis
   - Selection of k for k-fold cross validation:
     - value of k was chosen to be 5 to have booth diversity in results and also to keep the pace of the work up.
     - Smaller values of k might not have given us the diversity we desired, while higher values of k would have taken more time.
     - Higher value of k would also have affected the training size by reducing it further.
     - That again might not have been the best thing for our model.
   - ROC curve
     - ROC curve has been plotted on 2 different models.
     - One is the model implemented my me.
     - For that, corresponding probabilities instead of the exact labels were stored and passed onto for plotting the curve.

        - The latter one is the sklearn’s Bernoulli Naive Bayes model.
        - For that, I have directly obtained the corresponding probability values and used them to generate the plot.

Please Note:

1. All the plots and figures that were generated are stored in the plots folder while the datasets used are stored in the weights folder.
2. Since the theory questions could easily be done in the report itself, separate python file has not been created for it.

Question 4. Theory

1.

a) We would introduce dummy variable.

> $W(i) = B(0) + B(1)X(i) + u(i)$
>
> we would add dummy variabe Z with the intercept coefficient.
>
> Z can be given value 0 when it is for men and the value 1 when it is for women.
>
> So the equantion would be: $W(i) = Z.B(0) + B(1)X(i) + u(i)$
>
> so the independant eqaution for men would be: $W(i) = B(1)X(i) + u(i)$
>
> and for women it would be: $W(i) = B(0) + B(1)X(i) + u(i)$
>
> In the end, we can subtract the 2 equations to look for any kind of varitaion if it exists.

b) We would introduce dummy variable.

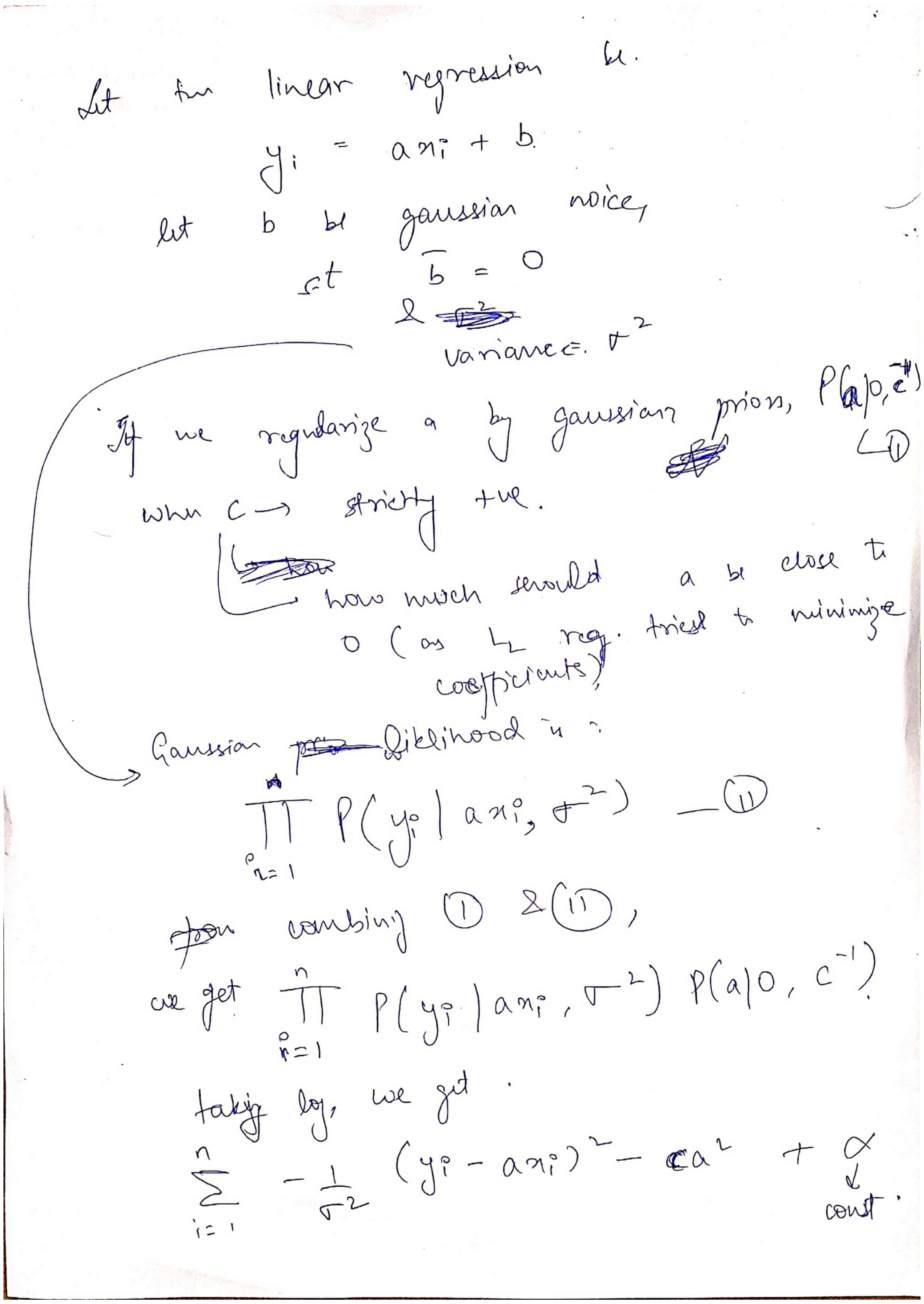
> $W(i) = B(0) + B(1)X(i) + u(i)$
>
> we would add dummy variable Z with the slope coefficient.
>
> Z can be given value 0 when it is for men and the value 1 when it is for women.
>
> So the equantion would be: $W(i) = B(0) + Z.B(1)X(i) + u(i)$
>
> so the independant eqaution for men would be: $W(i) = B(0) + u(i)$
>
> and for women it would be: $W(i) = B(0) + B(1)X(i) + u(i)$
>
> In the end, we can subtract the 2 equations to look for any kind of varitaion if it exists.

c) The model can be changed by changing the sign of B(1), and testing it on iid data. If the relation is now inversed, then the suspicion is confirmed.

2.

L2 Regularization or Ridge Regression promotes smaller coefficients. The intuition behind the same is that it aims to make sure that no one coefficient is too large. It also prevents any coefficient to become completely 0. Since in the regularization, sum of squares of slopes is taken. L2 tries to reduce the variance of estimates, which eventually tries to counteracts the effect of codependence between features. If the features are codependent, their variance would naturally be on a higher size, but this is counteracted by L2, therefore promoting smaller coefficients.

3.



Let the linear regression be.

$$y_i = a x_i + b$$

let b be gaussian noice,

st $\bar{b} = 0$

& ~~[illegible]~~

variance = σ^2

If we regularize a by gaussian prior, $P(a|0, c^{-1})$ — $\textcircled{I}$

when c → strictly +ve.

~~[illegible]~~ → how much should a be close to 0 (as L_2 reg. tries to minimize coefficients)

Gaussian ~~pr~~ likelihood is :

$$\prod_{i=1} P(y_i \mid a x_i, \sigma^2) \quad — \textcircled{II}$$

~~from~~ combining $\textcircled{I}$ & $\textcircled{II}$,

we get $\prod_{i=1}^{n} P(y_i \mid a x_i, \sigma^2)\, P(a \mid 0, c^{-1})$

taking log, we get .

$$\sum_{i=1}^{n} -\frac{1}{\sigma^2}(y_i - a x_i)^2 - ca^2 + \underset{\substack{\downarrow \\ \text{const}}}{\alpha}$$

It can be seen that we are going in the direction of calculating squares of slopes, thus resembling L2 regression.

If the final equation is maximized in terms of a, the MAP estimate for a would be found.

Reference taken from: https://stats.stackexchange.com/questions/163388/why-is-the-l2-regularization-equivalent-to-gaussian-prior