Harsh Tripathi
MT2022048

Machine Learning

Assignment 1
Part A : Regression Task

Dataset:
The regression dataset "*Life Expectancy Data.csv*" has 2938 rows × 22 columns. Which contains columns which affects the '*Life expectancy*' in many Countries and it is also our target column.

Analysis of Dataset:

- First need to make a data frame from the given csv file to work on python and rem_life is our data frame. rem_life.head() gives the following output, just to look at the data.

	Country	Year	Status	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	...	Polio	Total expenditure	Diphtheria	HIV/AIDS	GI
0	Afghanistan	2015	Developing	65.0	263.0	62	0.01	71.279624	65.0	1154	...	6.0	8.16	65.0	0.1	584.2592
1	Afghanistan	2014	Developing	59.9	271.0	64	0.01	73.523582	62.0	492	...	58.0	8.18	62.0	0.1	612.6965
2	Afghanistan	2013	Developing	59.9	268.0	66	0.01	73.219243	64.0	430	...	62.0	8.13	64.0	0.1	631.7449
3	Afghanistan	2012	Developing	59.5	272.0	69	0.01	78.184215	67.0	2787	...	67.0	8.52	67.0	0.1	669.9590
4	Afghanistan	2011	Developing	59.2	275.0	71	0.01	7.097109	68.0	3013	...	68.0	7.87	68.0	0.1	63.5372

5 rows × 22 columns

- After this we can check how the metrics in given data.

```
In [248]: rem_life.describe() #for looking at diff. metrics at data
```

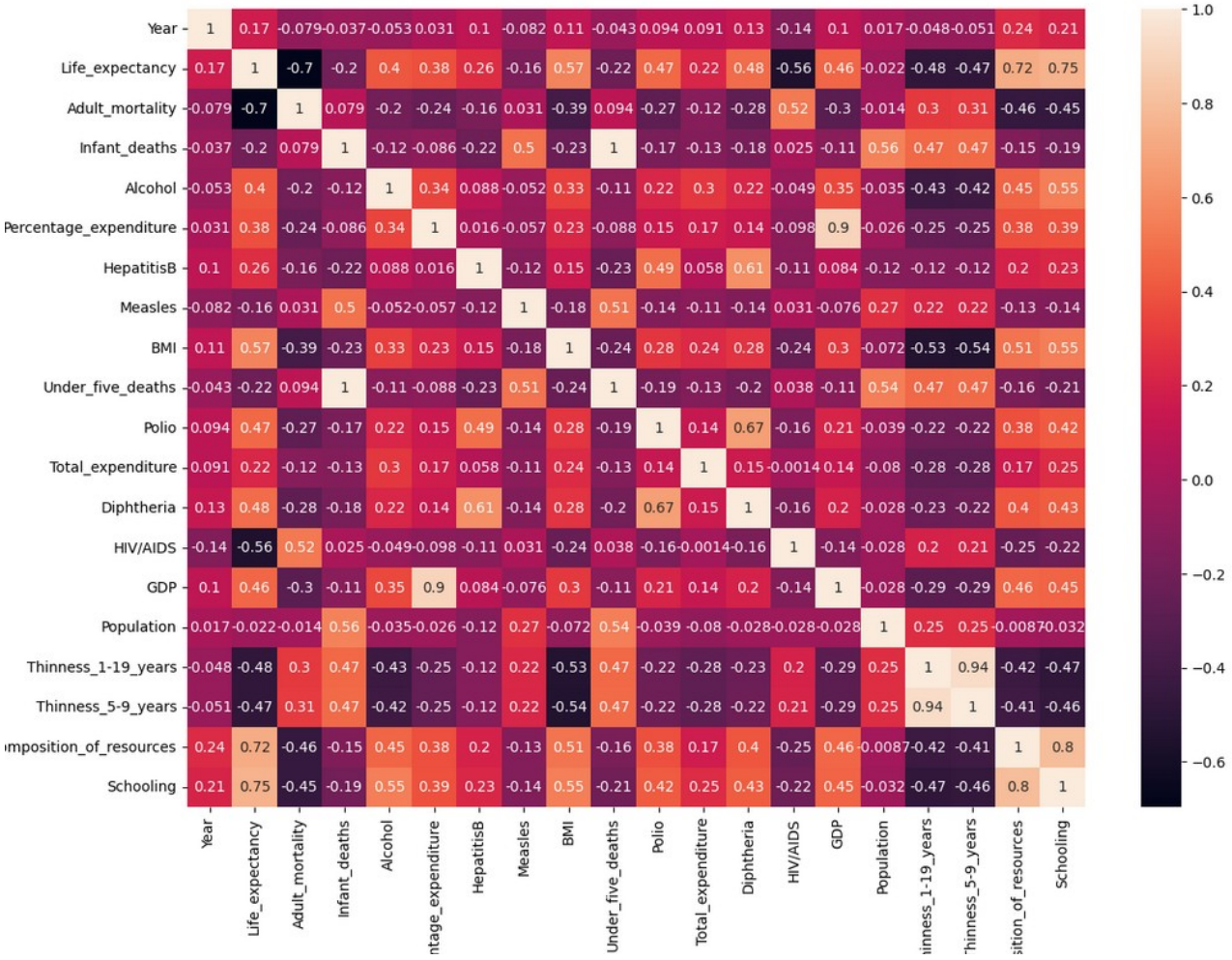
Out[248]:

	Year	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polio	ex
count	2938.000000	2928.000000	2928.000000	2938.000000	2744.000000	2938.000000	2385.000000	2938.000000	2904.000000	2938.000000	2919.000000	2
mean	2007.518720	69.224932	164.796448	30.303948	4.602861	738.251295	80.940461	2419.592240	38.321247	42.035739	82.550188	
std	4.613841	9.523867	124.292079	117.926501	4.052413	1987.914858	25.070016	11467.272489	20.044034	160.445548	23.428046	
min	2000.000000	36.300000	1.000000	0.000000	0.010000	0.000000	1.000000	0.000000	1.000000	0.000000	3.000000	
25%	2004.000000	63.100000	74.000000	0.000000	0.877500	4.685343	77.000000	0.000000	19.300000	0.000000	78.000000	
50%	2008.000000	72.100000	144.000000	3.000000	3.755000	64.912906	92.000000	17.000000	43.500000	4.000000	93.000000	
75%	2012.000000	75.700000	228.000000	22.000000	7.702500	441.534144	97.000000	360.250000	56.200000	28.000000	97.000000	
max	2015.000000	89.000000	723.000000	1800.000000	17.870000	19479.911610	99.000000	212183.000000	87.300000	2500.000000	99.000000	

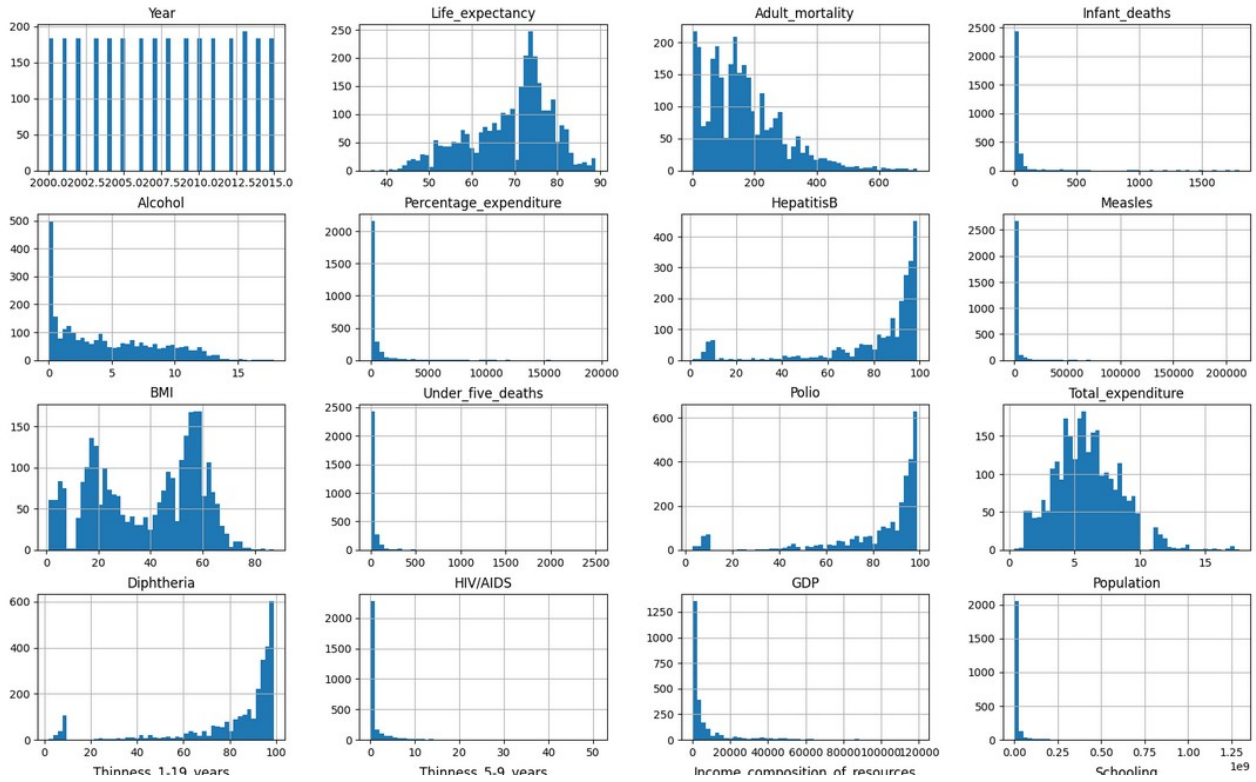
- Here we can check mean, std or 25%tile of data which gives insights and metrics. Also we can check that there are many colums in data which contains untidy names which needs to correct.
  Like "thinness 1-19 years" should be written as "Thinness_1-19_years and " BMI" should be as "BMI".
  Which can be done by using *rem_life.rename.*

```
rem_life.rename(columns = {" BMI " :"BMI",
                           "Life expectancy ": "Life_expectancy",
                           "Adult Mortality":"Adult_mortality",
                           "infant deaths":"Infant_deaths",
                           "percentage expenditure":"Percentage_expenditure",
                           "Hepatitis B":"HepatitisB",
```

- After correcting the columns we need to look at the co-relation between the data, which tells us how one columns affect the other.
  Given below is the heatmap co-related matrix which gives how columns are related and we can see that Income and Schooling are the one's which affects our outcome.



- Now we can look at the histograms to look for the data distributions, like in “Life expectancy” histo we can line after 70 is vey high which tells us most of the people dies in this age.

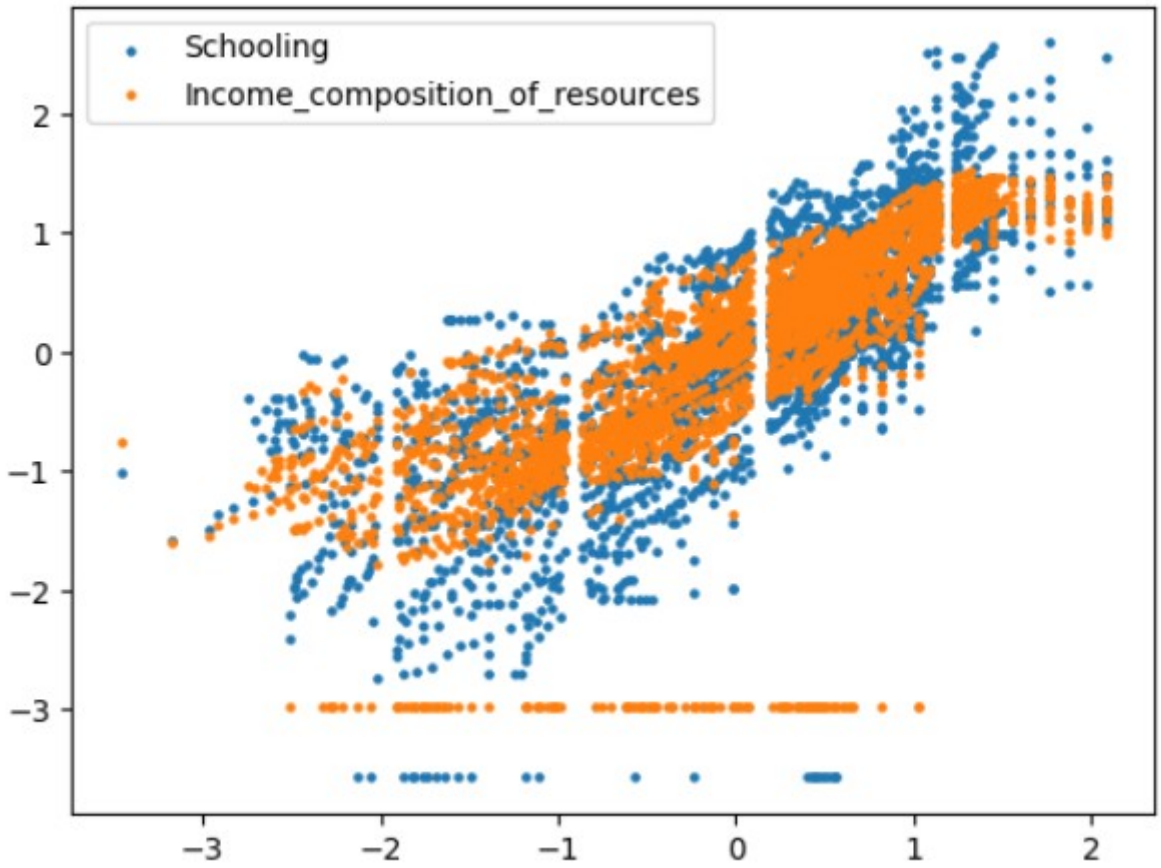


- Since our data contains lots of unrelated and unscalable values like values in “Schooling” is like 10 or 12 and in “GDP” it was like 584.259210 so we need to normalize our data such that all of our would come in some range and “Gradient Descent” would work fine else it will not give nice solution and the formula we used is *rem_life=(rem_life-rem_life.mean())/rem_life.std()*. We also need to remember that there should be no object column else this would not work or we can change these objects to numerical. In my case i did remove those columns which contains object values.

	Year	Life_expectancy	Adult_mortality	Infant_deaths	Alcohol	Percentage_expenditure	HepatitisB	Measles	BMI	Under_five_deaths	Polio
0	1.621486	-0.443615	0.790103	0.268778	-1.133365	-0.335513	-0.635838	-0.110366	-0.958951	0.255316	-3.267459
1	1.404747	-0.979112	0.854468	0.285738	-1.133365	-0.334384	-0.755503	-0.168095	-0.983896	0.274014	-1.047897
2	1.188008	-0.979112	0.830331	0.302697	-1.133365	-0.334537	-0.675726	-0.173502	-1.008841	0.292712	-0.877162
3	0.971269	-1.021112	0.862513	0.328137	-1.133365	-0.332040	-0.556061	0.032040	-1.033786	0.317642	-0.663742
4	0.754530	-1.052611	0.886650	0.345097	-1.133365	-0.367800	-0.516173	0.051748	-1.053742	0.342573	-0.621059

- Now we need to remove our target column from given dataframe and also need to convert to array.

- We can also how our main co-related columns affects our target column. Both x and y are normalized. It also contains some outlier which we will deal later.



- Since most of our data is ready now we can start buiding our model and we can also remove outlier as we will find them.

Gradient Descent(Multivariate) :

- In multivariate we need to pick many columns we take all columns which affects our target.
- For calculating the loss function we need to set out dependent and independent columns .

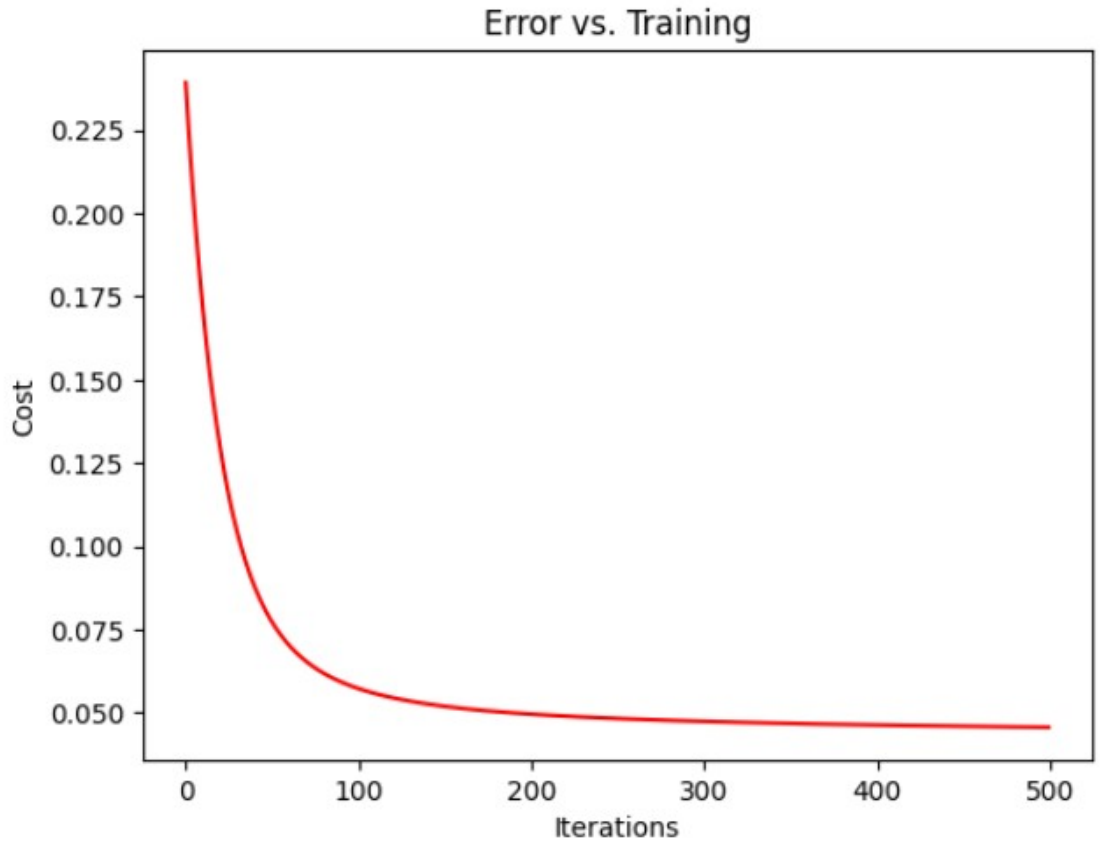
$$Least\ Squared\ Error = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$Cost\ Function = J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2, \quad m = number\ of\ sample\ data$$

- Now we need to add X0 which is all 1 to out existing X and theta must be set according to that also we need to split test and training data for checking our model. After setting all the variable we run our *Gradient Descent*. This is our theta and final cost after running the algo.

```
[[ 0.02187156 -0.03906824 -0.23658444 -0.0042707   0.00270799  0.06036286
  -0.01053728  0.00126636  0.09423534 -0.02758646  0.02307812  0.00863737
   0.04521939 -0.24483023  0.05483524  0.01845977 -0.02059248 -0.0236932
   0.22013714  0.22323431]]
0.04566807788887698
```

- We can also see how our cost decreases with the Iterations. After 150th iterations error becomes constant.



- Now we can predict our output since we got final theta. Our output will be containing some ‘nan’ values due to irregularity in data so we need to find those index and remove those ‘nan’ values in output as well as out test set.

- Finall we use MSE and MAE method to calculate our predictions, 0 values means our model is perfect. This is our model output.

```
def MAE(y_pred,y):
    res=np.sum(abs(y_pred-y))/len(y)
    return res
MAE(y_predUp, y_testUp)
```

```
0.273631946010005
```

```
MSE = np.square(np.subtract(y_testUp,y_predUp)).mean()
```

```
MSE
```

```
0.1278186179113694
```

Gradient Descent(Univariate) :

- In univariate we need a single most co-related columns as our independent data and feed to model. So picked up "*Schooling*" .
- Now we need to add 1 columns which contains all one to our independent variable and set theta acc. to that which is [1,2] in this case and split the data in test and train.
- By making few changes we can feed our model with the above dataset.

```
def MAE(y_pred,y):
    res=np.sum(abs(y_pred-y))/len(y)
    return res
MAE(y_predUp, y_unitestUp)
```

```
0.4771109393821663
```

```
MSE = np.square(np.subtract(y_unitestUp,y_predUp)).mean()
```

```
MSE
```

```
0.4682921334986371
```

Normal/Closed Equation(Multivariate) :

- In Normal/closed form we need out independent and dependent variable to calculate theta which we can directly used to predict the output.
- Sample value of X after normalization in multivariate:

```
array([[ 1.        ,  0.10431217, -0.81096437, ..., -0.92491148,
         0.87930682,  0.77620408],
       [ 1.        , -0.5459053 ,  1.84407207, ...,  0.80500734,
        -0.62374987, -0.7719127 ],
       [ 1.        ,  0.32105133,  0.95101436, ...,  0.87154268,
        -1.28756039, -1.81391438],
       ...,
       [ 1.        , -0.11242698,  0.81423975, ..., -0.14866585,
        -0.80392701, -1.06962747],
       [ 1.        , -0.97938361, -0.74659985, ..., -0.96926837,
         1.01206893,  1.0143759 ],
       [ 1.        , -0.5459053 , -1.05233132, ...,  1.02679181,
        -2.97553573, -1.96277177]])
```

- Since in normal form if we get ‘nan’ in input then our ouput will also give ‘nan’ so we need to clean/remove all ‘nan’ from data, one thing we can do is replace all ‘nan’ values with the mean value. Lines used for this is , this need to run for all test as well as training set :

```
colmean = np.nanmean(Ytrain, axis = 0)
Indxs = np.where(np.isnan(Ytrain))
Ytrain[Indxs] = np.take(colmean, Indxs[1])
```

- Equ. Of normal form

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

- So after setting the variable we can feed the data into model and our model gives this prediction.

```
def MAE(y_pred,y):
    res=np.nansum(abs(y_pred-y))/len(y)
    return res
MAE(y_prednor, y_testnor)
```

```
0.3123583200271751
```

```
MSE = np.square(np.subtract(y_prednor,y_testnor)).mean()
MSE
```

```
0.18558624578200908
```

Normal/Closed Equation(Univariate) :

- In univariate we will follow same model with some adjustment to our code and also to our X which is our independent variable.
- After feeding to our model we can see there is ‘nan’ to our output we can remove those things by removing those columns or replacing them with mean. If we want to remove ‘nan’ from input the we can replace them with mean while in output we can remove those rows.
- Predictions by our model :

```
def MAE(y_pred,y):
    res=np.nansum(abs(y_pred-y))/len(y)
    return res
MAE(y_prednor, y_testnor)
```

```
0.4339216368195522
```

```
MSE = np.square(np.subtract(y_prednor,y_testnor)).mean()
MSE
```

```
0.4242156448049466
```

Conclusion :

Following are the MSE and MAE across all the Linear Regression models :

	MSE	**MAE**
Multivarite Linear Regression	0.14944245648643675	0.2901235658481076
Univarite Linear Regression	0.6057274268133023	0.5279621910582338
Multivariate Normal Equation	0.1946995626060679	0.32442858435422844
Univariate Normal Equation	0.5436621912658486	0.527808039915621

From the above table we can understand that our model performs much better in Multivariate systems where multiple columns are present and all affects the target columns which is '*Life expectancy*'. While in case of Univariate only 'Schooling' is present and only corelated to target column by 0.75 metrics, hence little unefficient as compared to Multivariate.
So from all model '*Multivarite Linear Regression*' is the best performing for given data.

Harsh Tripathi
MT2022048

Part B : Logistic Regression

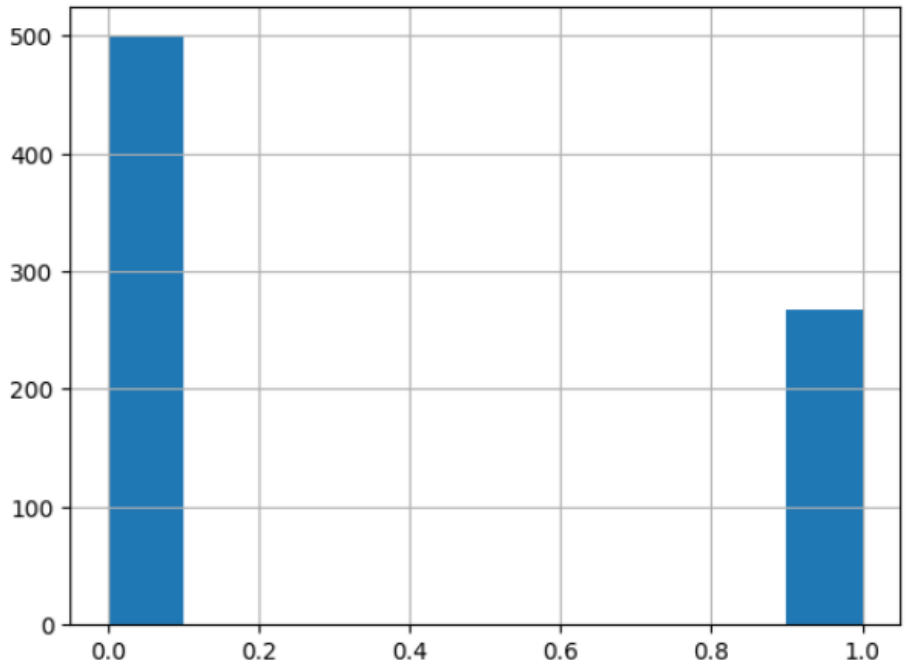
Dataset :
The Classification dataset "**diabetes.csv**" has 768 rows × 9 columns. In the data we are provided with many columns which are related to diabetes like 'Glucose', 'BMI' and 'Insiulin' and one 'Outcome' columns which shows whether person has diabetes or not and it also our target column.

Analysis of Dataset :

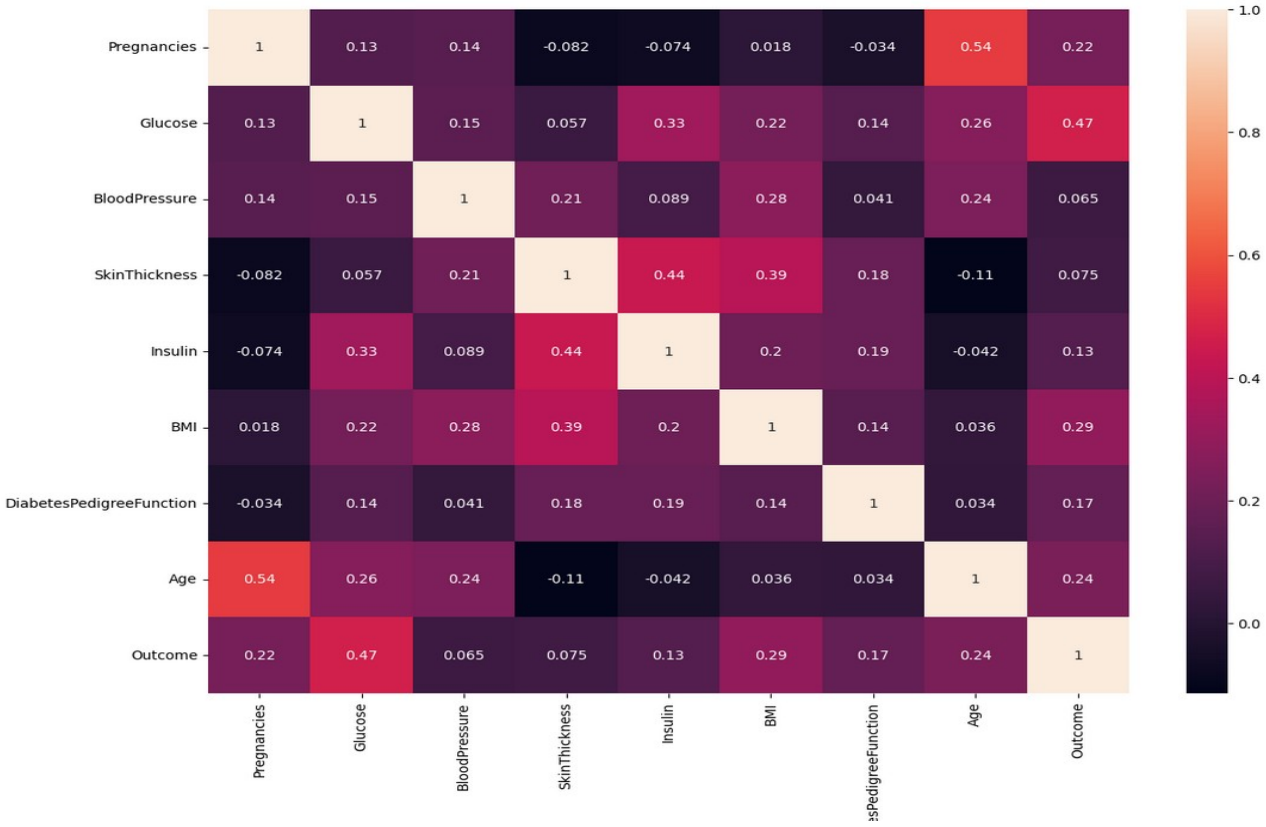
- First need to make a data frame from the given csv file to work on python and *dia* is our data frame. *dia.head()* gives the following output, just to look at the data.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

- After this we can check whether data contains some null values or not and which columns contains numerical value. Data contains some 0 value which may give bad prediction and we will clean as we find them.
- We can als see how the outcome values are biased towards the one set.



- We can check co-realted heatmap matrix for any corelations between the columns.



- Since from the above fig. we can see corealtion is not very high between the columns among all only 'Glucose' has corelation more than 0.4 for the outcome column.
- Now again we need to normalize the data but we can leave our target columns out of dataframe that needs to normalize since out target columns just contains 0 and 1.
- For the logistic functions we we need to clear the data from the outliers so we will use *quantile* method and remove 0 and 'nan' with the mean values.
- We can als see how the outcome values are biased towards the one set.
- Since after our data is normalize and target column is also split from the data we can feed the data into our model.

Gradient Descent(Multivariate) :

- In the logistic gradient descent we need to find the sigmoid function first which is

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

- So after splitting the data for training and test, computing the loss function we can feed data to our model.
- We will take columns according to correlated values.
- For the checking of the accuracy we need to find the True Positive, True Negative, False Postive and False Negative. Pre(TP/(TP+FP)) and re(TP/(TP+FN)) are precison and recall.

```
acc=(TP+TN)/(TP+TN+FP+FN)
```

```
acc
```

```
0.746268656716418
```

```

```

```
f1=(2*pre*re)/(pre+re)
f1
```

```
0.3928571428571428
```

Gradient Descent(Univariate) :

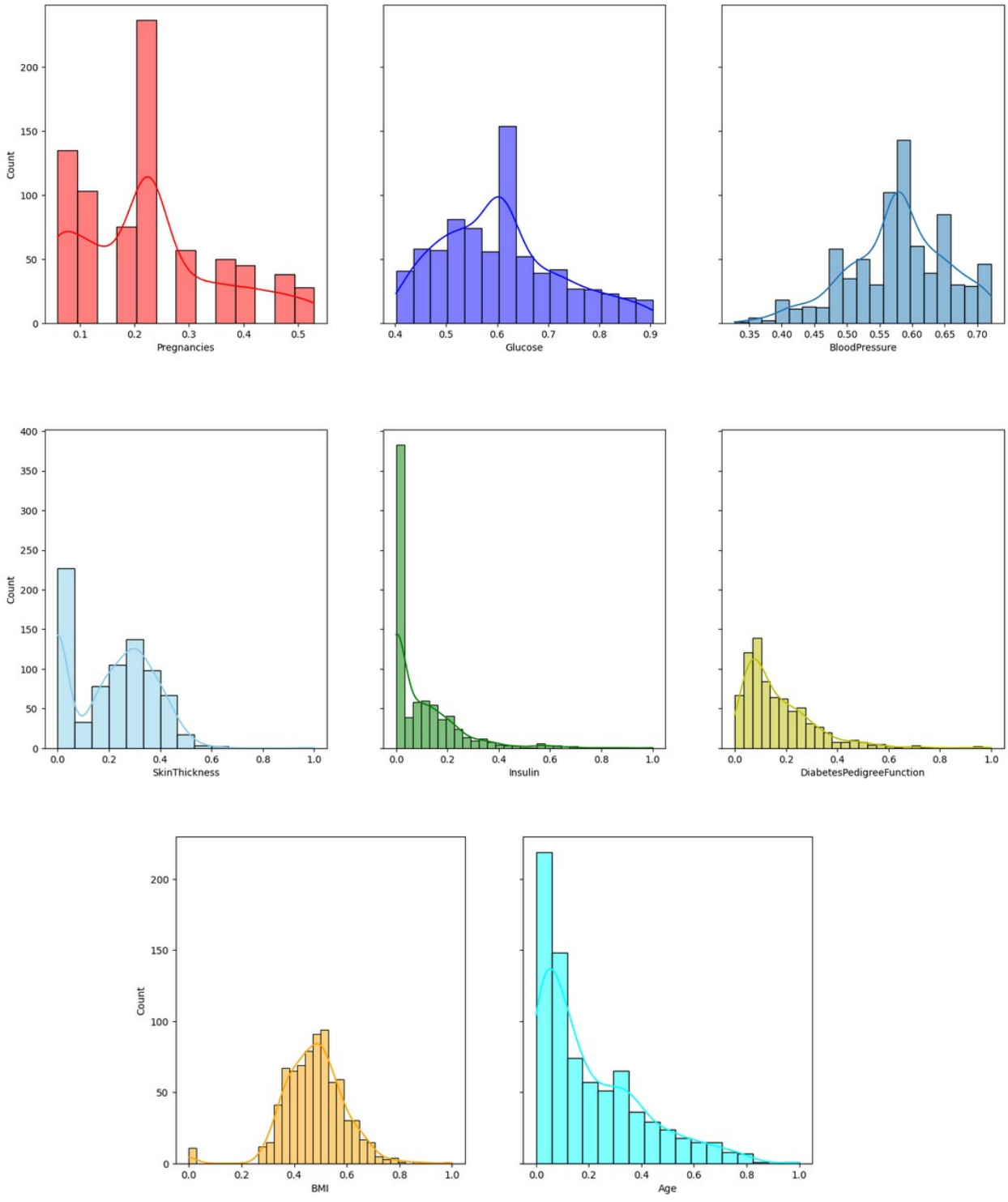
- In the Univariate we will take only 1 column i.e. glucose in our case which shows highest correlation with the outcome columns.
- Again after setting the ‘weight’ and ‘bias’ we will feed into our model.
- Accuracy of given model.

```
: acc=(TP+TN)/(TP+TN+FP+FN)
  acc
```

```
: 0.7276119402985075
```

Naive Baye's(Multivariate) :

- For the Naive Baye's fisrt we can use make histograms to plot a charts to check for the normal distributions for all the columns.



- From the histo we can see that mostly columns follows the normal distribution except few like '*skin thickness*' so we can drop these columns.
- In the naive bayes's we need to calculate prior, Gaussian likelihood and then we can calculate posterior probability .
-
- After feeding the data into model, following are the accuracy.

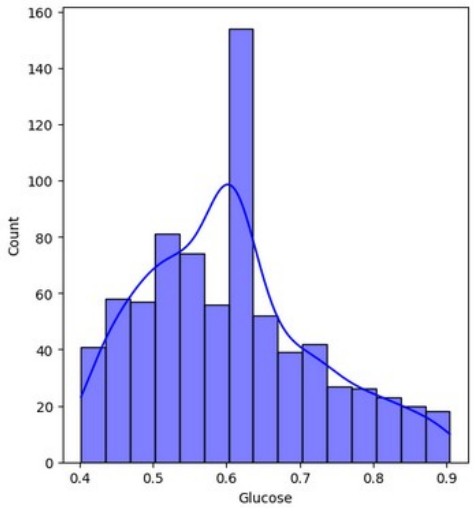
```
acc=(TP+TN)/(TP+TN+FP+FN)
```

```
acc
```

```
0.7272727272727273
```

Naive Baye's(Univariate) :

- In univariate Naive Bayes we simply take glucose column remove all others and the new data into same model after spilliting into test and train.
- Since has the highest corelation and good distribution among all the columns.



- After feeding the data into model we can see accuracy of the model which is.

```
acc=(TP+TN)/(TP+TN+FP+FN)
```

```
acc
```

```
0.6948051948051948
```

Conclusion :

Following are the F1_Score and Accuracy across all the Linear Regression models :

	F1_Score	**Accuracy**
Gradient Descent(Multivarite)	0.3928571428571428	0.746268656716418
Gradient Descent(Univariate)	0.31775700934579443	0.7276119402985075
Naive Baye's(Multivarite)	0.5858585858585857	0.7337662337662337
Naive Baye's(Univariate)	0.528735632183908	0.7337662337662337

From the above table we can understand that accuracy is over 70% for all model while F1 score variate as it shows how True Positive and other values like this varites over model. Also we can see that Naive Baye's performs beter for the given data since it has high F1_score.
So from all model '*Naive Baye's(Multivarite)*' is the best performing for given data.