Machine Learning Assignment 1

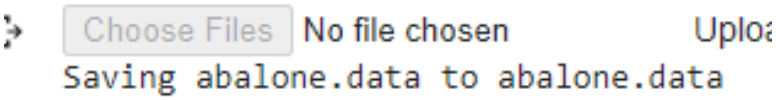
By Manvi Goel

Question 1

Working and Preprocessing

Upload the dataset at file.upload().

```
from google.colab import files
uploaded = files.upload()
```




```
Saving abalone.data to abalone.data
```

Exploratory Data Analysis.

1. Save the data in pandas dataframe.
2. Check the data.

```
dataset = pd.read_csv("abalone.data")
dataset.head()
```

	M	0.455	0.365	0.095	0.514	0.2245	0.101	0.15	15
0	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
1	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
2	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
3	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7
4	I	0.425	0.300	0.095	0.3515	0.1410	0.0775	0.120	8

Obs. The dataset does not contain column names.

3. Add column names for the dataset from readme for the dataset.

```
dataset.columns = ["Sex", "Length", "Diameter", "Height", "Whole Weight", "Shucked Weight", "Viscera Weight", "Shell Weight", "Age"]
dataset.head()
```

	Sex	Length	Diameter	Height	Whole Weight	Shucked Weight	Viscera Weight	Shell Weight	Age
0	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
1	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
2	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
3	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7
4	I	0.425	0.300	0.095	0.3515	0.1410	0.0775	0.120	8

4. Checking the data for outliers. Study means and standard deviation.

```
dataset.describe()
```

	Length	Diameter	Height	Whole Weight	Shucked Weight	Viscera Weight	Shell Weight	Age
count	4176.000000	4176.000000	4176.000000	4176.000000	4176.00000	4176.000000	4176.000000	4176.000000
mean	0.524009	0.407892	0.139527	0.828818	0.35940	0.180613	0.238852	9.932471
std	0.120103	0.099250	0.041826	0.490424	0.22198	0.109620	0.139213	3.223601
min	0.075000	0.055000	0.000000	0.002000	0.00100	0.000500	0.001500	1.000000
25%	0.450000	0.350000	0.115000	0.441500	0.18600	0.093375	0.130000	8.000000
50%	0.545000	0.425000	0.140000	0.799750	0.33600	0.171000	0.234000	9.000000
75%	0.615000	0.480000	0.165000	1.153250	0.50200	0.253000	0.329000	11.000000
max	0.815000	0.650000	1.130000	2.825500	1.48800	0.760000	1.005000	29.000000

5. The dataset includes values “M,” “F,” and “I” in the Sex column of the dataset.

```
stringData =  dataset.select_dtypes(include=[np.object]).columns
print(stringData)
```

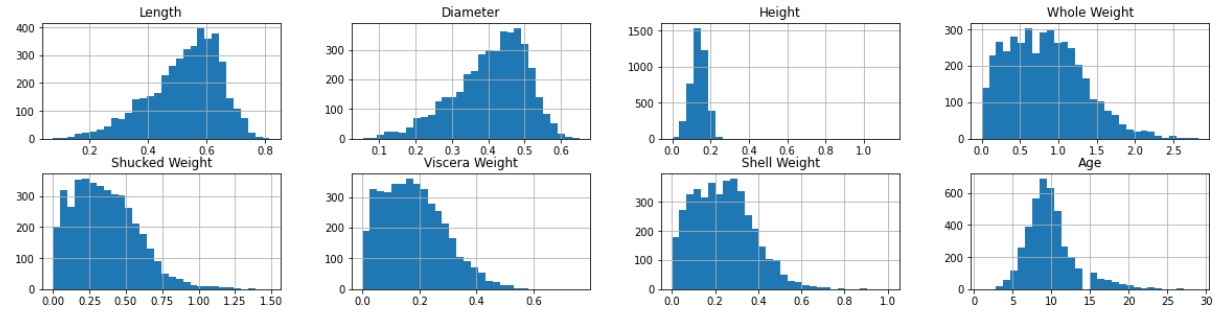
```
Index(['Sex'], dtype='object')
```

Observation. We need to drop the column.

6. I am checking the value distribution in the dataset.

```
dataset.hist(figsize=(20,10), grid = True, layout=(4, 4), bins = 30)
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f50e300bd50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e3009a50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2f7f110>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2f28690>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2edec10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2ea01d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2e55750>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2e89cd0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2e22150>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2dd77d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2db9d90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2d7c350>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2d308d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2ce7e50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2ca8410>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f50e2c5c990>]],
      dtype=object)
```



Observations.
Only height has the lowest value of 0.
The distribution is not normal.

7. Check for missing values in the dataset.

```
# check the missing points
missing = dataset.isnull().sum().sort_values(ascending = False)
percentageMissing = (missing/len(dataset))*100
pd.concat([missing, percentageMissing], axis = 1, keys= ['Number of missing values ', 'Percentage of Missing Values'])
```

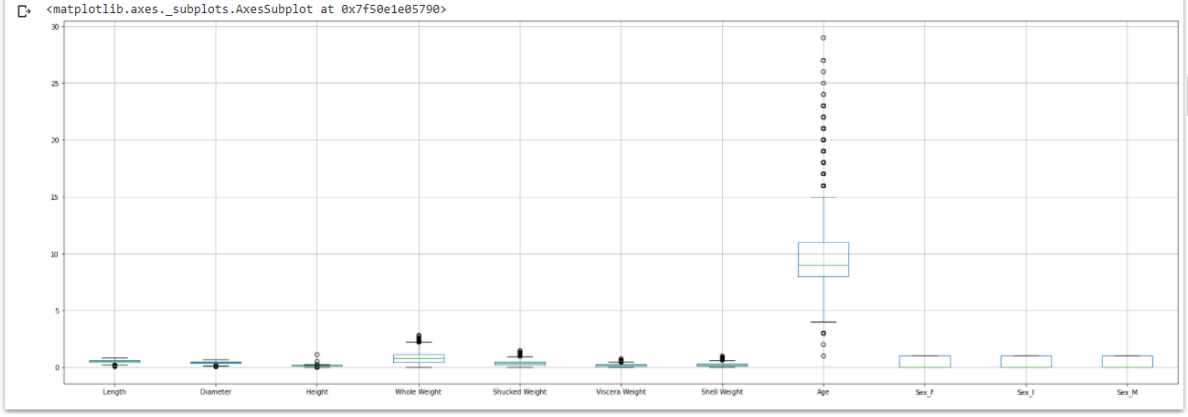
	Number of missing values	Percentage of Missing Values
Age	0	0.0
Shell Weight	0	0.0
Viscera Weight	0	0.0
Shucked Weight	0	0.0
Whole Weight	0	0.0
Height	0	0.0
Diameter	0	0.0
Length	0	0.0
Sex	0	0.0

Observation. No missing values.

8. Looking for outliers in the dataset.

```
dataset = pd.get_dummies(dataset)
dummyDataset = dataset.copy()
dataset.boxplot(figsize = (30, 10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f50e1e05790>
```



Data Preprocessing

1. Removing the “Sex” column and replacing it with original values.

```
#Preprocesing data
for label in "MFI":
    dataset[label] = (dataset["Sex"] == label)
del dataset["Sex"]

dataset.head()
```

	Length	Diameter	Height	Whole Weight	Shucked Weight	Viscera Weight	Shell Weight	Age	M	F	I
0	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7	True	False	False
1	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9	False	True	False
2	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10	True	False	False
3	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7	False	False	True
4	0.425	0.300	0.095	0.3515	0.1410	0.0775	0.120	8	False	False	True

Working on the model.

1. Splitting the data into X and y.
2. Splitting into training and testing dataset.

```
# Getting random indices.
indices = list(range(X.shape[0]))
training_instances = int(0.8 * X.shape[0])
np.random.shuffle(indices)
train = indices[:training_instances]
test = indices[training_instances:]

# Spliting the dataset.
X_train, X_test = X[train], X[test]
y_train, y_test = y[train], y[test]

X_train.shape
```

```
(3340, 10)
```

Linear Regression Model.

Working.

It calculates the loss by multiplying the coefficients to X and subtracting the values of y during each epoch.
It calculates the gradient.
Updates the theta value using the gradient.
Calculates the new cost.

```
# Gradient Descent Function
def gradientDescent(X, y, coeffMatrix, alpha, n_iters):

    # Values of all costs
    costValues = [0]*n_iters
    sizeY = len(y)

    # Iterating
    for iter in range(n_iters):

        # Calculating the loss
        loss = (X.dot(coeffMatrix.T)) - y

        # Calculating the gradient
        grad = X.T.dot(loss)/(sizeY)

        # Updating the coefficent values
        coeffMatrix = coeffMatrix - (alpha*grad.T)

        # Calculating the new cost.
        costVal = cost(X, y, coeffMatrix)
        costValues[iter] = costVal


    print("Final Model Coefficients: ")
    print(coeffMatrix)
    print("Final Model Cost: ", costValues[-1])
    return coeffMatrix, costValues
```

Predicting

Multiply the X vectors to the coefficient matrix obtained by the linear regression model.

```
def predict(X, coeffMatrix):

    # Using the new coefficients to calculate the y_pred values
    y_pred = X.dot(coeffMatrix.T)
    return y_pred
```

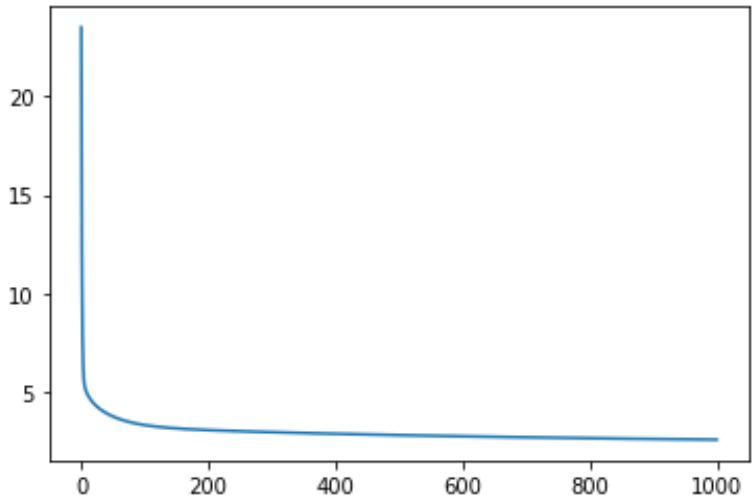
Calling the Linear Regression Model.
Train the model using the training data.
Predict the values of test data.
Calculate the RMSE error.

```
Initial Cost:  54.08922155688623
Final Model Coefficients:
[ 5.19746499  4.92417841  2.80373084  2.66756982 -8.663988   -0.96411979
  7.0411821   4.50603759  4.52457735  3.44554187]
Final Model Cost:  2.6234057925114587

Train data RMSE:  2.2905919726181962
Test data RMSE:  2.426629847301472
```

Plotting the cost values.

```
plt.plot(costValues)
```

```
[<matplotlib.lines.Line2D at 0x7f50e16d0dd0>]
```



Observation. The error decreases sharply in the beginning but slows down after reaching a value.

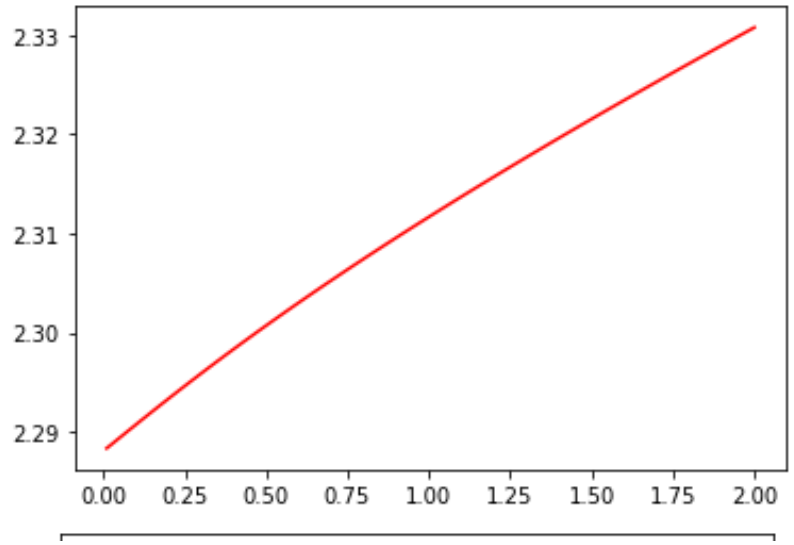
2. **Regularization Techniques**
Train the models for ridge and lasso regression.

```
ridgeModel = Ridge(alpha = a)
ridgeModel.fit(X_train, y_train)
predictedTarget = ridgeModel.predict(X_test)
ridgeRMSEs.append(RMSE(y_test, predictedTarget))

lassoModel = Lasso(alpha = a)
lassoModel.fit(X_train, y_train)
predictedTarget = lassoModel.predict(X_test)
lassoRMSEs.append(RMSE(y_test, predictedTarget))
```

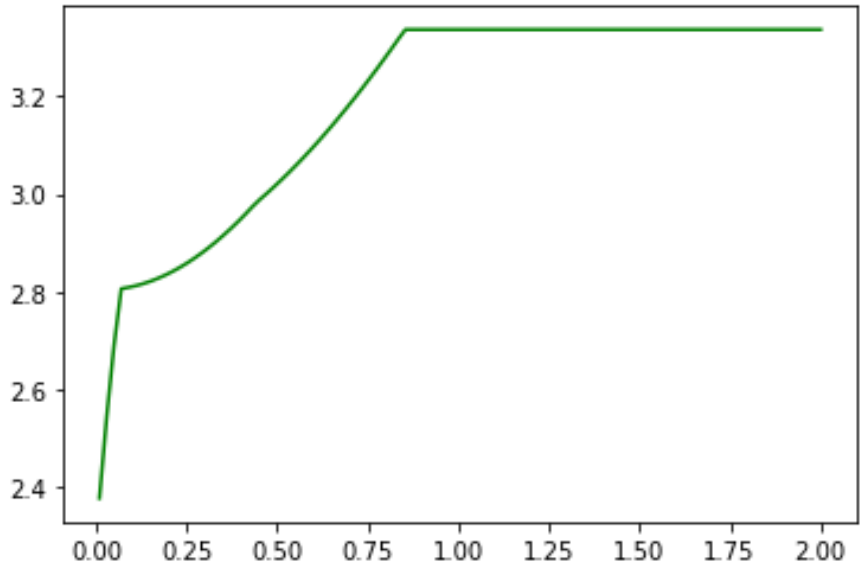
Plot the graphs with multiple values of alpha.

```
The alpha values are:  [0.01       0.03010101 0.05020202 0.07030303 0.09040404 0.11050
 0.13060606 0.15070707 0.17080808 0.19090909 0.2110101  0.23111111
 0.25121212 0.27131313 0.29141414 0.31151515 0.33161616 0.35171717
 0.37181818 0.39191919 0.4120202  0.43212121 0.45222222 0.47232323
 0.49242424 0.51252525 0.53262626 0.55272727 0.57282828 0.59292929
 0.6130303  0.63313131 0.65323232 0.67333333 0.69343434 0.71353535
 0.73363636 0.75373737 0.77383838 0.79393939 0.8140404  0.83414141
 0.85424242 0.87434343 0.89444444 0.91454545 0.93464646 0.95474747
 0.97484848 0.99494949 1.01505051 1.03515152 1.05525253 1.07535354
 1.09545455 1.11555556 1.13565657 1.15575758 1.17585859 1.1959596
 1.21606061 1.23616162 1.25626263 1.27636364 1.29646465 1.31656566
 1.33666667 1.35676768 1.37686869 1.3969697  1.41707071 1.43717172
 1.45727273 1.47737374 1.49747475 1.51757576 1.53767677 1.55777778
 1.57787879 1.5979798  1.61808081 1.63818182 1.65828283 1.67838384
 1.69848485 1.71858586 1.73868687 1.75878788 1.77888889 1.7989899
 1.81909091 1.83919192 1.85929293 1.87939394 1.89949495 1.91959596
 1.93969697 1.95979798 1.97989899 2.        ]
```

The graphs for Ridge Regression. (x axis = alpha values, y axis = RMSE error values)



The plot for Lasso Regression.



The value of alphas
Self Selected.

```
# Minimum values of alpha for ridge and lasso
alpha_ridge = alphaValues[ridgeRMSEs.index(min(ridgeRMSEs))]
alpha_lasso = alphaValues[lassoRMSEs.index(min(lassoRMSEs))]

print("The best alpha value for ridge regression is: ", alpha_ridge)
print()
print("The best alpha value for lasso regression is: ", alpha_lasso)
```

```
The best alpha value for ridge regression is:  0.01

The best alpha value for lasso regression is:  0.01
```

Using Grid Search

1. Use the same array.

For Lasso Regression.

```
The best alpha value for lasso Regression:
{'alpha': 0.01}
```

The value for alpha is the same using both approaches.

For Ridge Regression.

```
ridgeModelprime = GridSearchCV(Ridge(), param_grid = alphaValues)
ridgeModelprime.fit(X_train, y_train)

print("The best alpha value for ridge Regression: ")
ridgeModelprime.best_params_
```

```
The best alpha value for ridge Regression:
{'alpha': 0.050202020202020206}
```

The value of alpha from the manual calculation is different from the grid search implementation. But the difference is not too large.

Observation.
The difference between alpha lies in implementing Grid Search, which does not solely depend on RMSE, which we are using for our alpha selection.

Question 2.

Working and Preprocessing

1. Load the dataset into the colab notebook.

Exploratory Data Analysis.

1. Load the dataset to pandas dataframe

```
# Load the dataset
dataset = pd.read_csv("diabetes2.csv")
dataset.head()
```

	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

Obs. The data contains column names. It has eight columns.

2. Checking the dataset information.

```
dataset.info()
```

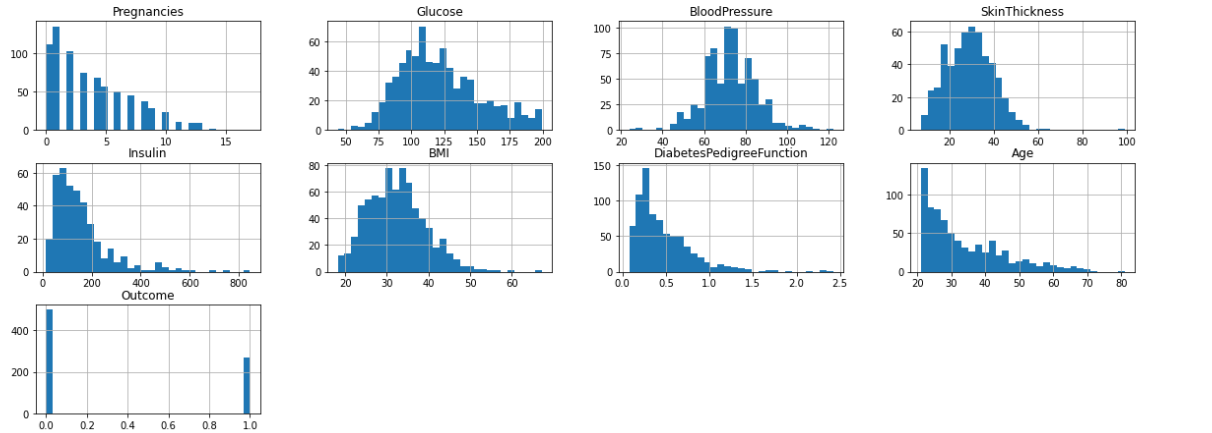
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 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
```

3. The dataset contains 0 values for Blood Pressure which should be wrong.
   Obs. We should replace them.
4. Dataset description

```
dataset.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	763.000000	733.000000	541.000000	394.000000	757.000000	768.000000	768.000000	768.000000
mean	3.845052	121.686763	72.405184	29.153420	155.548223	32.457464	0.471876	33.240885	0.348958
std	3.369578	30.535641	12.382158	10.476982	118.775855	6.924988	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	64.000000	22.000000	76.250000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	29.000000	125.000000	32.300000	0.372500	29.000000	0.000000
75%	6.000000	141.000000	80.000000	36.000000	190.000000	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

5. Histograms.



6. String data. No column has string data.

```
stringData =  dataset.select_dtypes(include=[np.object]).columns
print(stringData)
```

```
Index([], dtype='object')
```

7. It contains missing values.

```
# check the missing points
missing = dataset.isnull().sum().sort_values(ascending = False)
percentageMissing = (missing/len(dataset))*100
pd.concat([missing, percentageMissing], axis = 1, keys= ['Number of missing values ', 'Percentage of Missing Values'])
```

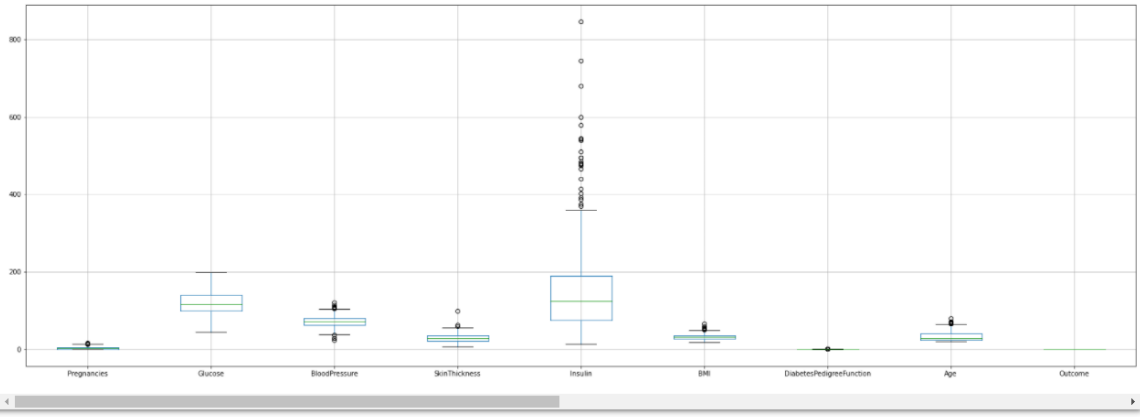
	Number of missing values	Percentage of Missing Values
Insulin	374	48.697917
SkinThickness	227	29.557292
BloodPressure	35	4.557292
BMI	11	1.432292
Glucose	5	0.651042
Outcome	0	0.000000
Age	0	0.000000
DiabetesPedigreeFunction	0	0.000000
Pregnancies	0	0.000000

Obs. We can replace them with the median.

8. Boxplot

```
dataset.boxplot(figsize = (30, 10))
```

```
/usr/local/lib/python3.7/dist-packages/numpy/core/_asarray.py:83: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-
  return array(a, dtype, copy=False, order=order)
<matplotlib.axes._subplots.AxesSubplot at 0x7fc52398af50>
```



Preprocessing
Changing the 0 values for Blood Pressure with the median values using the outcome.

```
#Replacing the zero-values for Blood Pressure
df1 = dataset.loc[dataset['Outcome'] == 1]
df2 = dataset.loc[dataset['Outcome'] == 0]
df1 = df1.replace({'BloodPressure':0}, np.median(df1['BloodPressure']))
df2 = df2.replace({'BloodPressure':0}, np.median(df2['BloodPressure']))
dataframe = [df1, df2]
dataset = pd.concat(dataframe)
```

Working on the model.
Splitting the data into X and y.
Splitting the data into training, test, and validation sets.

Normalizing the data.

```
# Noramlising the data
means = np.mean(X_train, axis = 0)
stds = np.std(X_train, axis = 0)

X_train = (X_train - means)/stds
X_validation = (X_validation - means)/stds
X_test = (X_test - means)/stds
```

Adding a column for intercept.

```
# adding an extra column for intercept
X_train = np.column_stack(([1] * X_train.shape[0], X_train))
X_test = np.column_stack(([1]*X_test.shape[0], X_test))
X_validation = np.column_stack(([1]*X_validation.shape[0], X_validation))
```

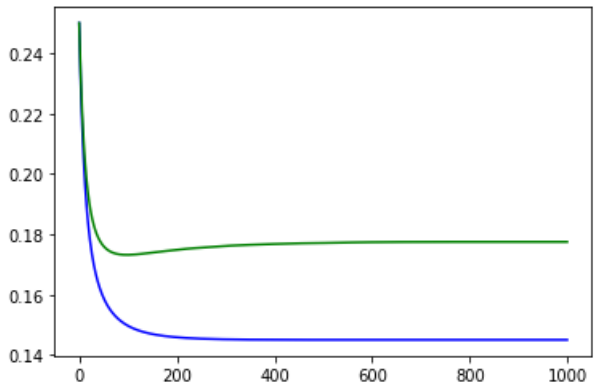
Logistic regression Model.

1. Define functions for SGD and BGD.
2. SGD = using one value for theta.
3. BGD = using all the values of X for updating the values of theta.

Predicting.
Calculate the linear regression. Calculate the sigmoid value. Assign values depending on the sigmoid values.

SGD.
Train the model.
Plot the validation and training data loss per epoch. (x = epoch, y = error)

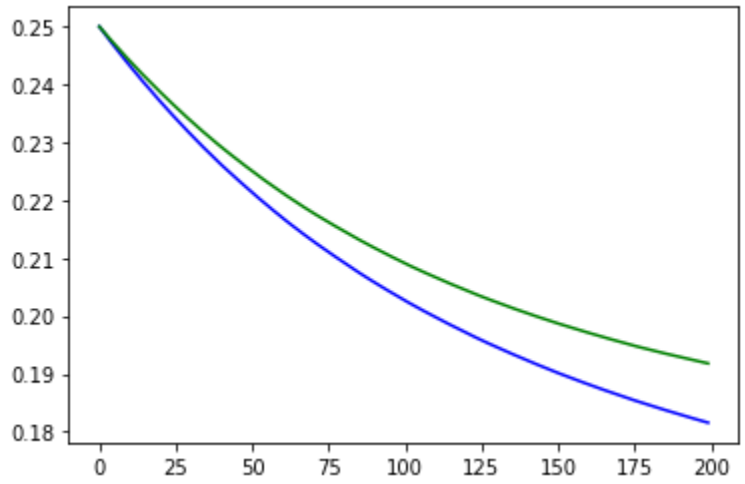
```
Initial Model Cost:  0.6931471805599468
Final Model Coefficients:
[-1.0069, 0.3807, 1.1889, -0.0839, -0.0853, -0.2263, 0.8469, 0.3063, 0.1099]
Final Model Cost:
0.1450828477980217
```



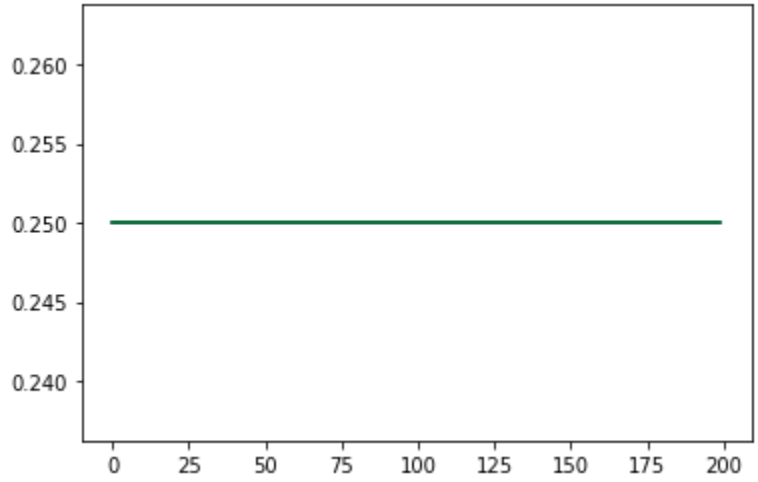
The model shows a steep dive in error initially, but the rate slows down after some time.
We see the error keeps decreasing for the training data, which shows that the model might be overfitting after 80 - 100 epochs. The error in the validation set rises after the reaching the minimum. We should set our number of iterations to at that point for an optimum result.

Implement using three learning rates given

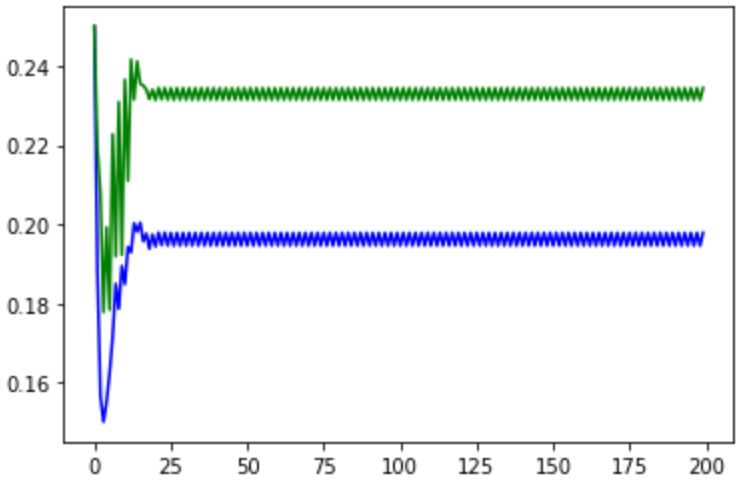
For 0.01



For 0.00001



For 10



We can see that the 3 learning rates give varied results.
0.01 gives optimum results. The error decreases steadily for both sets. We can increase the number of iterations to see this model and previous model converging.

We do not see much difference in the case of 0.0001 as the learning rate might be too low for it to make a change. That is, after multiplying with gradient it does not change the theta values by a lot. Hence the error remains almost constant.

10 is too large for learning this data, and it is possible for the algorithm to miss the minima; hence we can detect oscillations in the graph.

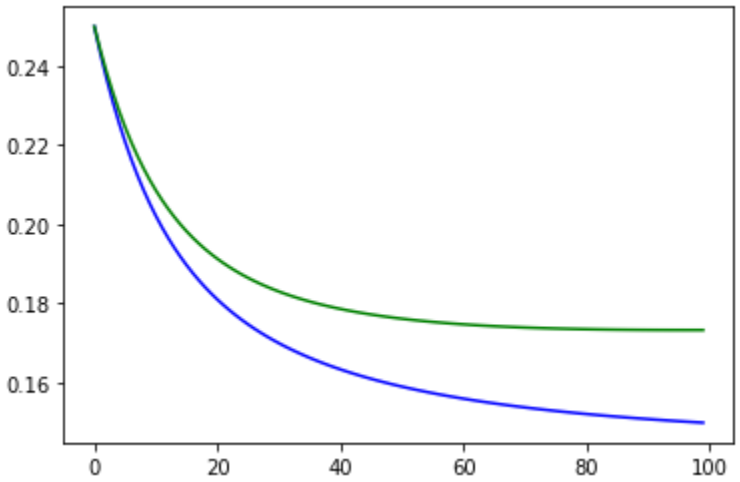
Print the confusion matrix and accuracy scores.

```
Initial Model Cost:  0.6931471805599468
Final Model Coefficients:
[-0.1495, 0.0836, 0.1893, 0.0614, 0.0174, 0.0312, 0.1211, 0.0704, 0.0767]
Final Model Cost:
0.20293888260602883
The confusion matrix:
[[38  9]
 [14 16]]

The f1 score is:  0.5818181818181818

The precision is:  0.64

The recall is:  0.5333333333333333

The accuracy is:  0.7012987012987013
```

We can see that our original cost for the zero matrix as coefficients decreases, and we achieve satisfactory results.

We might be able to achieve better results by reducing the skewness of the dataset or by sampling the data.

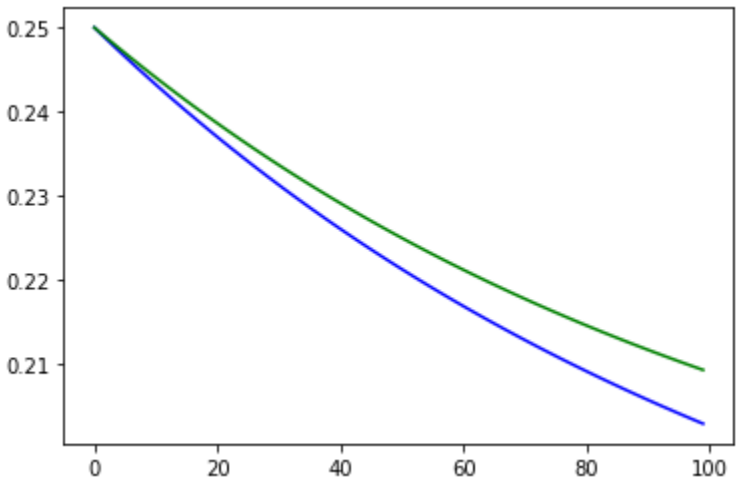
BGD. (The results are similar to SGD)
Original Plot.



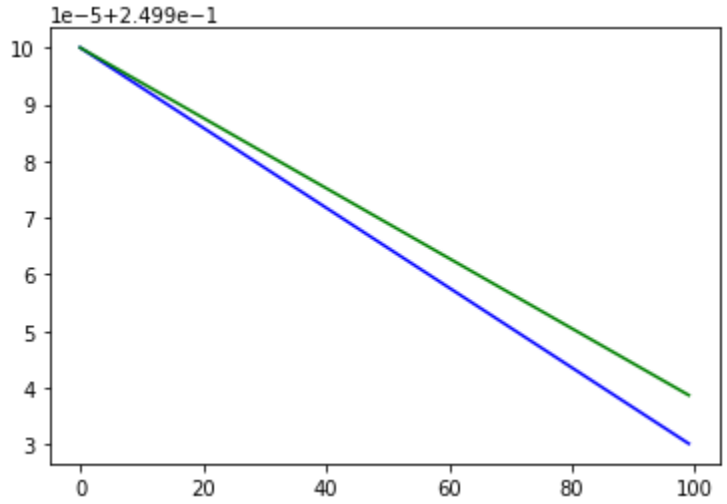
We can see here also, that the training loss keeps on decreasing while the validation loss starts increasing after reaching a minima.

Using 3 learning rates.

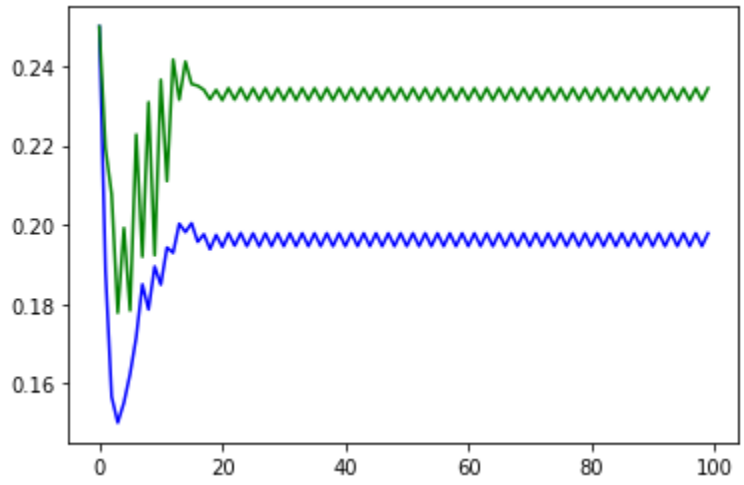
1. 0.01



2. 0.0001



3. 10

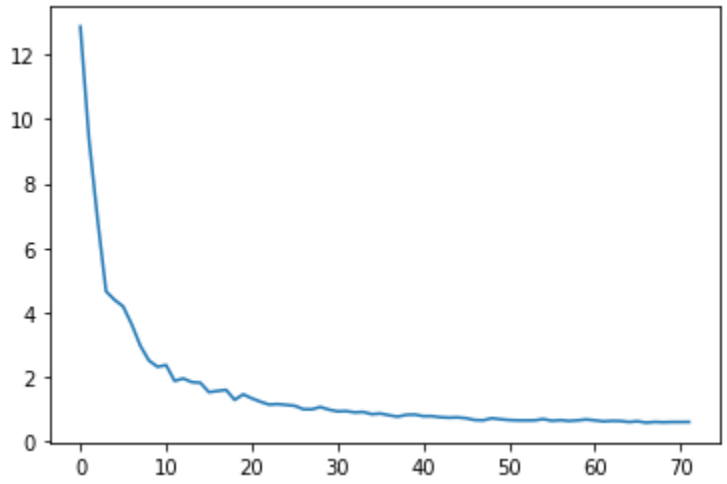


We see that 0.1 and 0.0001 both achieve significant decrease in error rates for this model. But 0.1 results in a better error in less number of epochs. We may have to increase the number of epochs for both the models to converge. The results are similar to SGD for learning rate 10 as we can see oscillations in the graph.

The accuracy scores and confusion matrix.

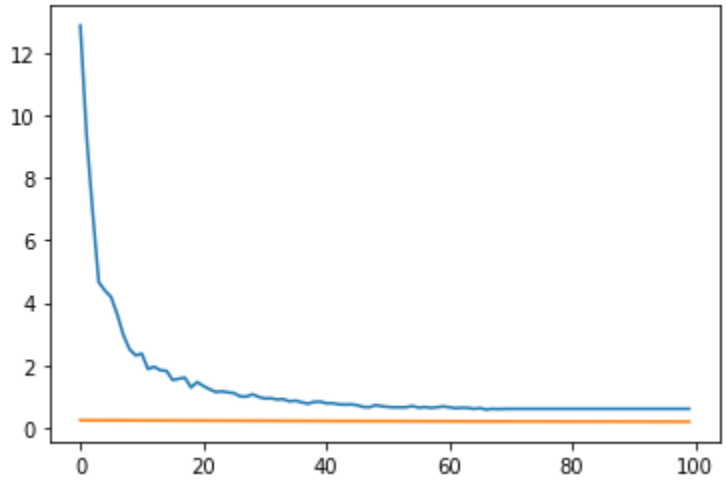
```
Initial Model Cost:  0.6931471805599468
Final Model Coefficients:
[-0.14954178  0.08344099  0.1891629   0.06153227  0.0174964   0.03128463
  0.12130003  0.0701579   0.07651399]
Final Model Cost:  0.2029457616930269
The confusion matrix:
[[38  9]
 [14 16]]

The f1 score is:  0.5818181818181818

The precision is:  0.64

The recall is:  0.5333333333333333

The accuracy is:  0.7012987012987013
```

2. Using the sklearn implementation. (SGDOptimizer using log error)
We train the model using X_train.
We report the plot for loss per epoch. The model converges in only 70 epochs.



This is similar to the original model, which took 80 - 100 epochs to converge.

We print the loss values with the original model and plot the both to compare.



We can see that our original loss is less in the beginning compared to sklearn implementation and hence shows a lower difference between the loss drop for each epoch.

The confusion matrix and score for the sklearn implementation

```
The confusion matrix:
[[34 13]
 [13 17]]

The f1 score is:  0.5666666666666667

The precision is:  0.5666666666666667

The recall is:  0.5666666666666667

The accuracy is:  0.6623376623376623
```

The values are comparable to the original model. Hence, we can say that our model working correctly.

Question 3.
Working.

Mount google drive

```
# Mounting google drive
from google.colab import drive
drive.mount('/content/drive')
```

Upload the four files to google drive.

```
# import the files from google drive
import gzip
file1 = gzip.open('/content/drive/MyDrive/train-images-idx3-ubyte.gz','r')
file2 = gzip.open('/content/drive/MyDrive/train-labels-idx1-ubyte.gz','r')
file3 = gzip.open('/content/drive/MyDrive/t10k-images-idx3-ubyte.gz','r')
file4 = gzip.open('/content/drive/MyDrive/t10k-labels-idx1-ubyte.gz','r')

image_size = 28
train_images = 60000
test_images = 10000

# Read image
file1.read(16)
buf = file1.read(image_size * image_size * train_images)
data = np.frombuffer(buf, dtype=np.uint8).astype(np.float32)
train_data1 = data.reshape(train_images, image_size, image_size)
```

Update the address in file1, file2, file3, and file4.
Read the files and add to train data and test data.

Data Preprocessing
Delete the samples that are not Trouser or pullover.

```
# Data preprocessing
def preProcess(num, X, y):
  train_data = []
  y_train = []
  for i in range(num):
    # keep if trouser or pullover
    if (y[i][0] == 1) or (y[i][0] == 2):
      train_data.append(X[i])

      # append the values of target
      y_train.append(y[i][0] - 1)

  return train_data, y_train

# Preprocess the data
X_train, y_train = preProcess(train_images, train_data1, array1)
X_test, y_test = preProcess(test_images, test_data1, array2)

print("The number of samples left in training data is: " , len(X_train))
```

```
The number of samples left in training data is:  12000
```

Working on the model.
Binarize the values for training data and testing data.
Convert all values to 0 or 1.

```
[11] # Binarize the data
     def binarize(data):
       n = len(data)
       new_data = np.zeros((n, 784))

       for i in range(n):
         new_data[i] = data[i].ravel()
         new_data[i] = [int(round(i/255)) for i in new_data[i]]
       return new_data

     X_train = binarize(X_train)
     X_test = binarize(X_test)
```

Naive Bayes Classification.
Working.
Calculate the prior probabilities using the training data.

Assign the number classcs values.

```
def fit(self, X, y):
  n_samples, n_features = X.shape
  self._classes = np.unique(y)
  n_classes = len(self._classes)

  self._features = np.zeros((n_classes, n_features), dtype = np.float64)
  self._classcs = np.zeros((n_classes, 1), dtype = np.int64)
  self._priors = np.zeros(n_classes, dtype = np.float64)

  # calculate the priors
  for c in self._classes:
    Xc = X[c == y]
    self._features[c, :] = Xc.sum(axis = 0)
    self._priors[c] = Xc.shape[0]/float(n_samples)

    if c == 0:
      self._classcs[0] = Xc.shape[0]

    if c == 1:
      self._classcs[1] = Xc.shape[0]
```

Predicting.
Calculate the priors and class conditionals. Get the values using the features stored in the model. Calculate the posterior probabilities for all classes. Assign the class with highest probability.

```
def predict(self, X):
  y_pred = [self._predx(x) for x in X]
  return y_pred

def _predx(self, x):
  posts = []
  for id, c in enumerate(self._classes):

    # calculate the prior probability
    prior = np.log(self._priors[id])
    classConditionals = np.sum(np.log(self._pdf(id, x)))

    # calculate the posterior probability
    posterior = prior + classConditionals
    posts.append(posterior)

  # argmax to get the class
  return self._classes[np.argmax(posts)]
```

Calling the K fold Models.
K chosen = 3.
The training data contains 12000 samples.
Firstly k = 3 divides the data into 3 parts that are easy to train and do not take very long.
The dataset is too large to use leave one out cross validation. But k = 3 gives us the opportunity to try different models without taking too long to train and the computatoion is also not very heavy.

Since the training data is not skewed towards one class, boht classes have lamost equal samples. Hence a small K will also give appropriate results.

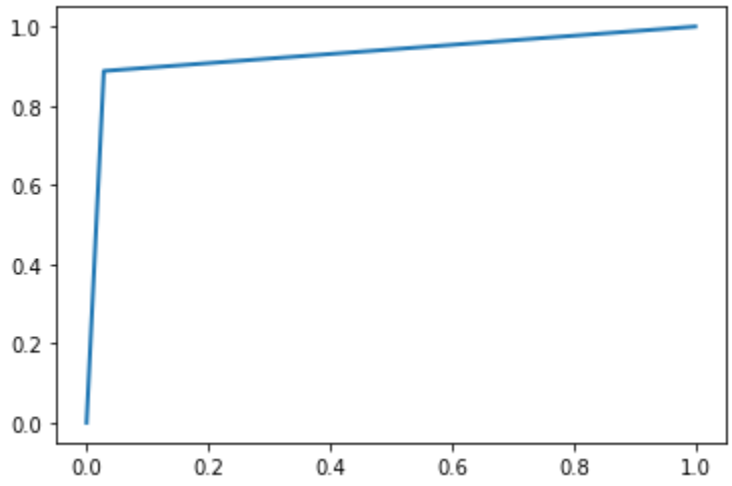
Calling the naive bayes classiefier.
Train the models three times, one for each fold.

```
model = naiveBayesClass()

# First training data
X_train1 = np.concatenate((train1, train2))
y_train1 = np.concatenate((y1, y2))

# First testing data
X_test1 = train3
y_test1 = y3

# 1st model
model.fit(X_train1, y_train1)
y_pred1 = model.predict(X_test1)
```

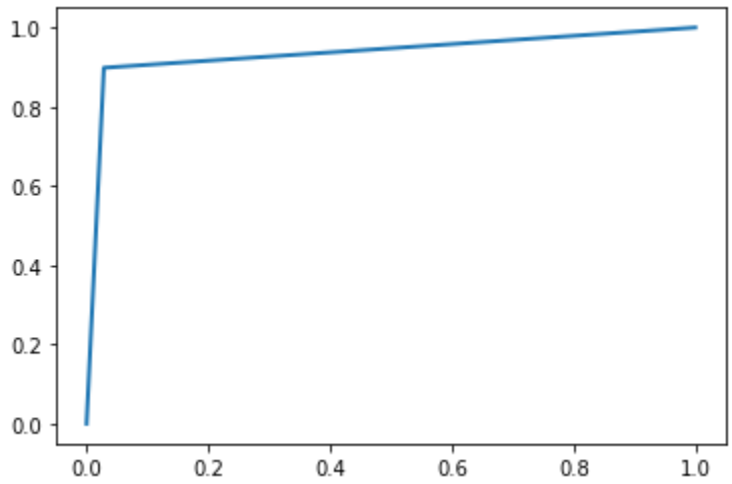
Making the confusion matrix
Use the sklraen implementation to calculate the confusion matrix.

```
The confusion matrix 1:
[[1921   56]
 [ 227 1796]]

The confusion matrix 2:
[[1982   58]
 [ 199 1761]]

The confusion matrix 3:
[[1904   79]
 [ 213 1804]]
```

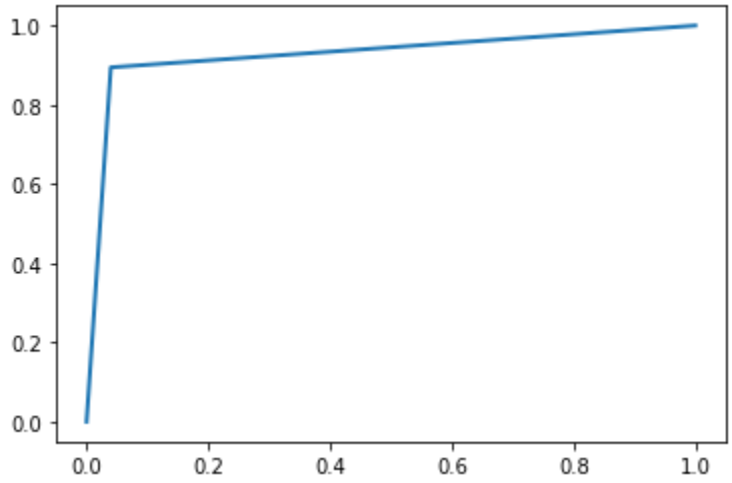
Plot the ROC curves.
1st model



2nd model



3rd model

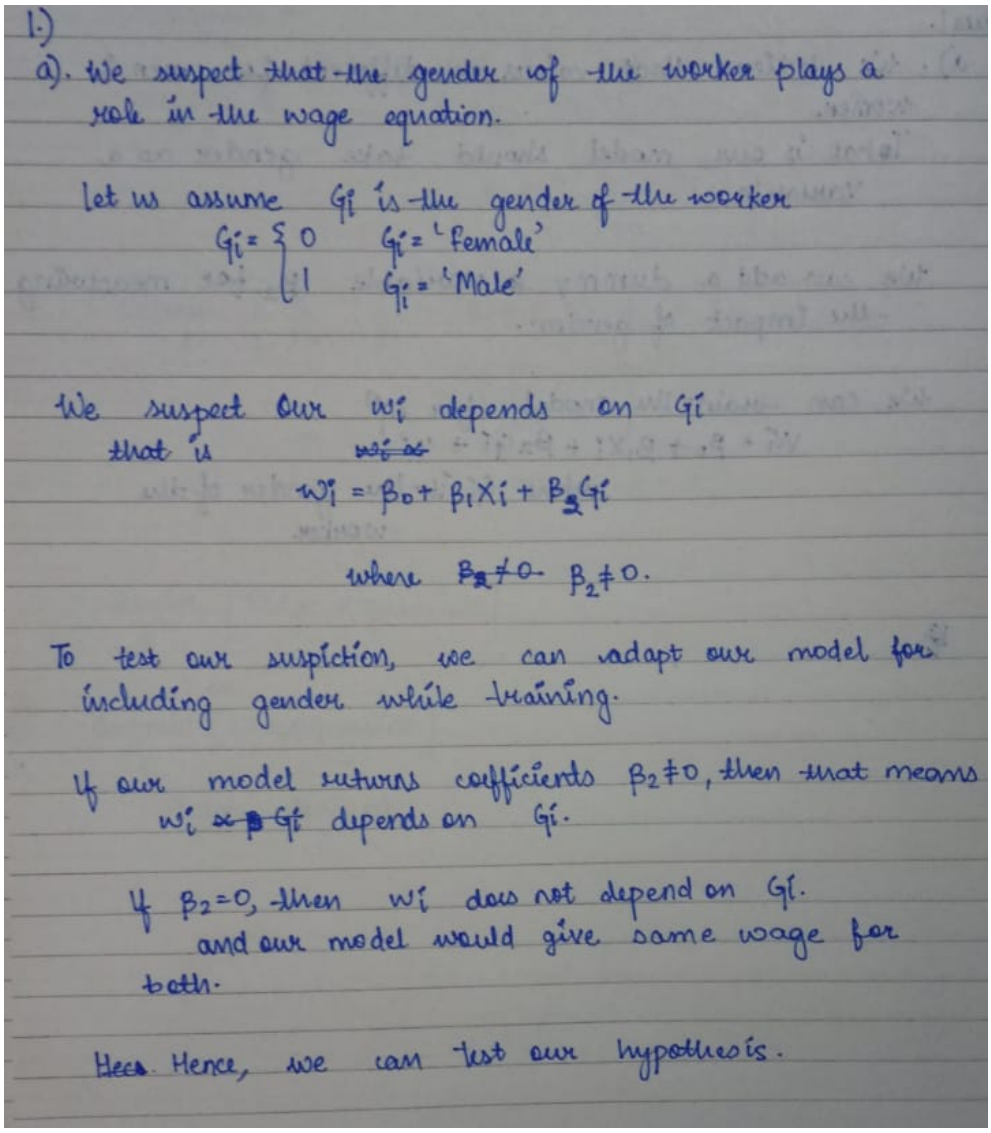


Calculate the precision, accuracy and recall.

```
First model
Precision score:  0.9697624190064795
Accuracy scores 0.93575
Recall score:  0.8877904102817598

Second Model
Precision score:  0.9681143485431556
Accuracy scores 0.92925
Recall score:  0.898469387755102

Third Model
Precision score:  0.9580456718003186
Accuracy scores 0.927
Recall score:  0.8943976202280615
```

Question 4.

1.



1.)

a). We suspect that the gender of the worker plays a role in the wage equation.

let us assume G_i is the gender of the worker

$$G_i = \begin{cases} 0 & G_i = \text{'Female'} \\ 1 & G_i = \text{'Male'} \end{cases}$$

We suspect our w_i depends on G_i that is

$$w_i = \beta_0 + \beta_1 X_i + \beta_2 G_i$$

where $\beta_2 \neq 0$.

To test our suspicion, we can adapt our model for including gender while training.

If our model returns coefficients $\beta_2 \neq 0$, then that means w_i depends on G_i.

If $\beta_2 = 0$, then w_i does not depend on G_i. and our model would give same wage for both.

Hence, we can test our hypothesis.

1. b

b) We suspect that the value of β_1 depends on the gender of the worker.

let us assume G_i is the gender of the worker

$$G_i = \begin{cases} 0 & G_i = \text{'female'} \\ 1 & G_i = \text{'male'} \end{cases}$$

We suspect β_1 depends on G_i

that is $\beta_1 = \beta_1' + \beta_2 G_i$

where $\beta_2 \neq 0$.

To test our hypothesis, we can adapt our model exe equation to

$$W_i = \beta_0 + (\beta_1' + \beta_2 G_i) X_i + e_i$$

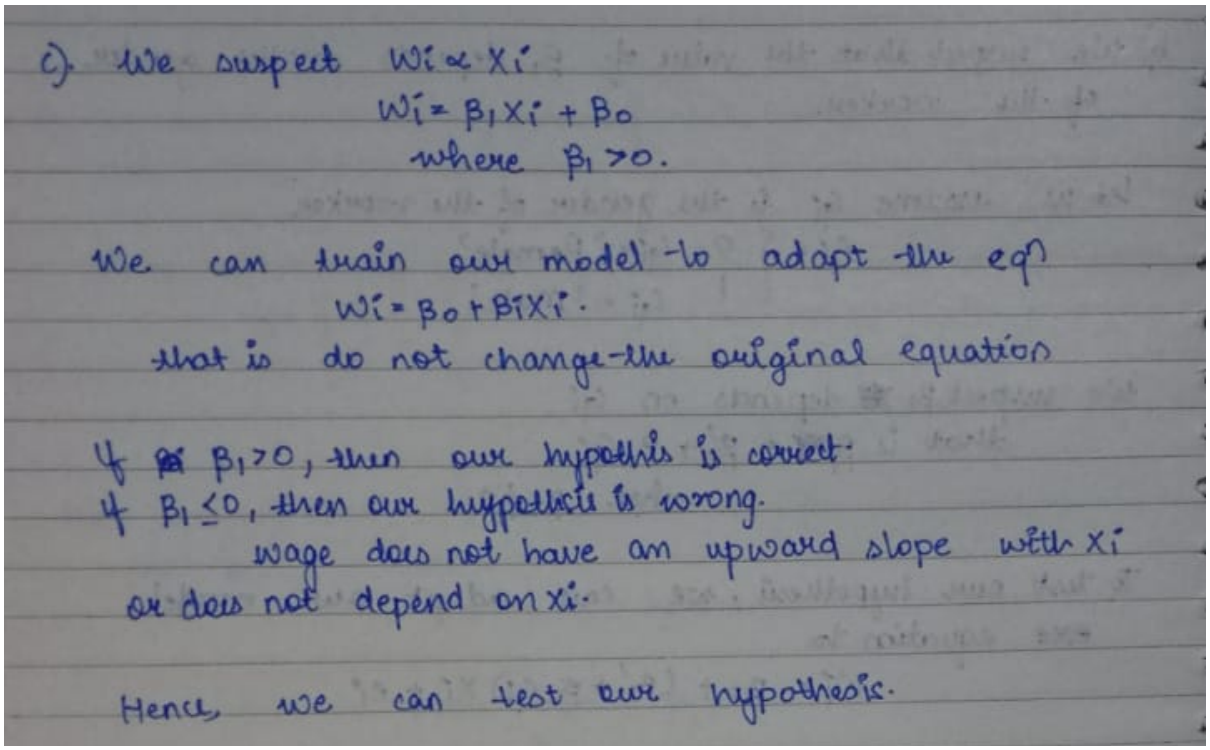
$$= \beta_0 + \beta_1' X_i + \beta_2 G_i X_i$$

We can train our model to adapt to this eqn.

If $\beta_2 = 0$, then gender does not change the slope of experience

If $\beta_2 \neq 0$, then our ~~sex~~ hypothesis is correct.

1. c


c). We suspect $W_i \propto X_i$

$W_i = \beta_1 X_i + \beta_0$

where $\beta_1 > 0$.

We can train our model to adapt the eqn

$W_i = \beta_0 + \beta_1 X_i$.

that is do not change the original equation

If $\beta_1 > 0$, then our hypothis is correct.

If $\beta_1 \leq 0$, then our hypothesis is wrong.

wage does not have an upward slope with X_i or does not depend on X_i.

Hence, we can test our hypothesis.

2 and 3

Ques 4.

2).

Regularisation is useful to prevent the overfitting of data with respect to the training data. Regularization lowers the bias. If we lower the value of the coefficients, the model will not be able to adapt to the data too much. That is, lower the coefficients, lower the ability to mug up the data.

We also introduce a penalty factor

$$pf = \alpha \sum_{i=1} \theta_i^2$$ where α = alpha or regularization factor.

We add it the cost function of the model. This factor will ensure that the values of θ remains small. Which in turn will ensure that our model does not overfit

3.)

To find the eqn for L2 regularization we need to find the θ_{MAP} that is the parameter with maximum posterior

We have

$$\theta_{MAP} = \underset{\theta}{\operatorname{argmax}}\ P(\theta|y)$$

$$= \underset{\theta}{\operatorname{argmax}}\ \frac{P(y|\theta) \cdot P(\theta)}{P(y)}$$

3 contd.

If $P(y)$ is constant, then

$$= \underset{\theta}{\operatorname{argmax}}\ P(y|\theta)\, P(\theta)$$

Taking log,

$$= \underset{\theta}{\operatorname{argmax}}\ \log\left(P(y|\theta).P(\theta)\right)$$

$$= \underset{\theta}{\operatorname{argmax}}\ \left[\log(P(y|\theta)) + \log(P(\theta))\right] \quad \text{—(i)}$$

[Using $\ln ab = \ln a + \ln b$]

We also take likelihood function:

$$\alpha(\theta|y) = P(y|\theta)$$

$$= \prod_{i=1}^{n} P_y\,(y_i|\theta, \sigma^2)$$

$$- (y_i - (\theta_0 + \theta_1 x_{i,1} + \dots + \theta_p x_{i,p}))^2$$

$$= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}}\, e$$

Putting $P(\theta)$ and $P(y|\theta)$ in (i)

$$\hat{\theta}_{MAP} = \underset{\theta}{\operatorname{argmax}}\left[\log \prod_{i=0}^{n} \frac{1}{\sigma\sqrt{2\pi}}\, e^{-(y_i - (\theta_0 + \theta_1 x_{i,1} + \dots + \theta_p x_{i,p}))^2} + \log \prod_{i=0}^{p} \frac{1}{\tau\sqrt{2\pi}}\, e^{-\frac{\theta_i^2}{2\tau^2}}\right]$$

$$= \underset{\theta}{\operatorname{argmax}}\left[-\sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_{i,1} + \dots + \theta_p x_{i,p}))^2 + \sum_{i=0}^{p} \frac{\theta_i^2}{2\tau^2}\right]$$

3rd contd.

$= \underset{\theta}{\text{argmin}} \; *$

taking out the negative sign

$= \underset{\theta}{\text{argmin}} \left(\frac{1}{2\sigma^2} \left[\sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_{i,1} + \ldots + \theta_p x_{i,p}))^2 + \frac{\sigma^2}{\tau^2} \sum_{i=0}^{p} \theta_i^2 \right] \right)$

$= \underset{\theta}{\text{argmin}} \left[\sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_{i,1} + \ldots + \theta_p x_{i,p}))^2 + \lambda \sum_{i=0}^{p} \theta_i^2 \right]$

where $\lambda = \frac{\sigma^2}{\tau^2}$

$= \underset{\theta}{\text{argmin}} \; [\text{Ridge regression}]$

$= \underset{\theta}{\text{argmin}} \; [\text{L2 regression}]$ or.

Hence, derived.

Thanks