NUST
NATIONAL UNIVERSITY
OF SCIENCES & TECHNOLOGY

MACHINE LEARNING

Project # 1

CREDIT CARD FRAUD DETECTION

Submitted by:

Muhammad Faizan Faiz

Student ID # 362940

Submitted to:

Dr. Hassan Sajid

Submission date:

26 April, 2022

Department of Robotics and Artificial Intelligence
School of Mechanical and Manufacturing Engineering (SMME)

Introduction

Dataset:

Dataset containing 284, 807 transactions was provided to me out of which only 492 were fraudulent transactions. The data have 29 features from V1 to V28 and amount of transaction. Data have been labelled as ‘0’ or ‘1’. The transactions labelled as ‘0’ are legit transactions while transactions labelled as ‘1’ are fraudulent transactions.

Data Pre-Processing:

Separating the data:

The data was first separated on the basis of the labels. The data facing labels ‘1’ was separated from the data facing labels ‘0’. Two excel files (.csv) were formed out of which one had data with ‘0’ labels while other had data with labels ‘1’.

Then from both the datasets, the column with labels were separated to prepare the data to be scaled.

Scaling / Normalizing the data:

Normalization is done to scale data within the range of 0 and 1. The data with labels ‘0’ was scaled using min-max scaling / normalization technique. A value is normalized as follows:

$$y = (x - \text{min}) / (\text{max} - \text{min})$$

Where the minimum and maximum values are for the normalized value x.

Splitting the data:

For an unbiased evaluation of prediction performance, data must be split. The dataset is divided into either two sets i.e. Training set and Test set or three sets i.e. Training set, Cross-Validation set and Test set.

The Training set is used to train the model i.e. to find the optimal weights, or coefficients, for linear regression, logistic regression or neural networks.

The Cross-Validation set is used to evaluate model during hyperparameter tuning. After fitting the model with the training set CV set evaluate its performance with the validation set for each considered setting of hyperparameters.

The test set is required for a correct evaluation of the final model. It should not be used for fitting or validation purposes.

In this case the dataset was split as 60% training, 20% validation and 20% testing. So out of 284, 807 samples, training set have 170884 samples while cross-validation and test set have 56961 samples.

Both the datasets i.e. with labels ‘0’ and labels ‘1’ were split separately.

Concatenating the data:

The datasets were then concatenated. Both the training sets with the labels ‘0’ and ‘1’ were concatenated. Similarly, cross-validation and test sets were also concatenated.

Mathematical Model

<u>Mathematical Model Details:</u>

Hypothesis:

A hypothesis is a proposed explanation based on insufficient evidence or assumptions. It's only an educated assumption based on certain known facts that hasn't been verified yet. A good hypothesis is one that can be tested and found to be true or false.

Let's look at an example to better grasp the hypothesis. According to some scientists, ultraviolet (UV) light can harm the eyes and induce blindness. In this case, a scientist just states that UV rays are hazardous to the eyes, but people infer that they can lead to blindness. It may or may not be doable, however. As a result, these assumptions are referred to as hypotheses.

In Machine Learning, one of the most widely utilized statistical notions is the hypothesis. It's utilized in Supervised Machine Learning, where an ML model uses an available dataset to learn a function that best maps the input to the associated outputs.

A basic linear function was chosen as the hypothesis, and non-linearity (sigmoid function) was applied to it. Simple linear function was chosen since it produced the least amount of error when compared to others. The hypothesis used is as follows:

$$h_\theta(x) = g(\theta^T x)$$

Here,

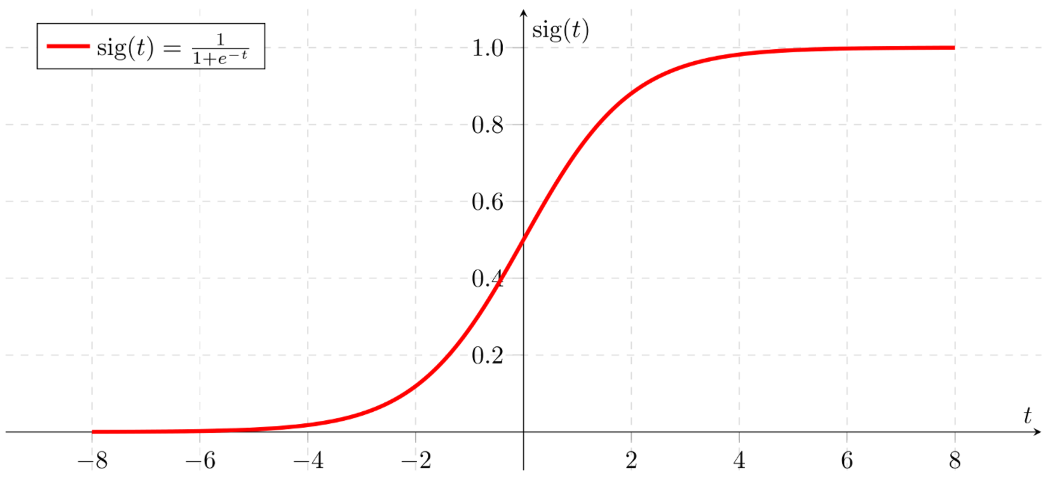
$$z = \theta^T x = \theta_0^T x_0 + \theta_1^T x_1 + \theta_2^T x_2 + \ldots\ldots\ldots + \theta_{29}^T x_{29}$$

Sigmoid Function:

We utilize the sigmoid function since it exists between two points (0 to 1). As a result, it is particularly useful in models where the probability must be predicted as an output. Because the likelihood of anything only occurs between 0 and 1, sigmoid is the best option.

It is possible to differentiate the function. That is, the slope of the sigmoid curve may be found at any two places. Although the function is monotonic, the derivative is not.

The graph of sigmoid function is as under:



As you can see, the sigmoid is a function that asymptotes both values and only occupies the range 0 to 1. This makes it ideal for binary classification with possible output values of 0 and 1. When a linear regression model produces a continuous result, such as -2.5, -5, or 10, the sigmoid function converts it to a number between 0 and 1.

$$\begin{cases} \sigma(z) < 0.5 & if\ z < 0 \\ \sigma(z) \geq 0.5 & if\ z \geq 0 \end{cases}$$

This can be interpreted as a probability that indicates whether the output should be sorted into class 1 or class 0. If the sigmoid function returns a value less than 0.5, you sort it into class 0. You classify it into class 1 if it doesn't fit into any of the other categories. As a result, the logistic regression model predicts the following in practice:

$$\hat{y} = \begin{cases} 0 & if\ \sigma(z) < 0.5 \\ 1 & if\ \sigma(z) \geq 0.5 \end{cases}$$

Cost Function:

We've been able to calculate the least value for the sum of squared residuals analytically using linear regression. This isn't achievable with logistic regression because we're working with a convex function rather than a linear one. The gradient descent approach is often used in real machine learning applications to iteratively identify the global minimum. To get gradient descent to discover the global minimum, we must first create a loss function (also known as a cost function) for the parameters of our logistic regression model that gradient descent is attempting to minimize. For classifications that differ from the actual results, the cost function imposes a penalty. We utilize the logarithmic loss of the probability returned by the model for the logistic regression cost function.

$$cost(\beta) = \begin{cases} -log(\sigma(z)) & if\ y = 1 \\ -log(1 - \sigma(z)) & if\ y = 0 \end{cases}$$

The full cost function with ‘m’ representing the number of samples is represented as:

$$Cost(\beta) = (-1/m)\sum_{i=1}^{m} [y_i log(\sigma(\beta_t x_i)) + (1 - y_i) * log(1 - \sigma(\beta_t x_i))]$$

Gradient Descent Algorithm:

The gradient is the vector that directs us to the highest ascent. To obtain the least, subtract our gradient from the original cost and go in the opposite direction of where the gradient is pointing.

By taking the first partial derivative of the cost with respect to each parameter in β, we can calculate the gradient. β is a j-dimensional vector.
The initial cost function is:

$$Cost(\beta) = (-1/m)\sum_{i=1}^{m} [y_i log(\sigma(\beta_t x_i)) + (1 - y_i) * log(1-\sigma(\beta_t x_i))]$$

With regard to each β_j, we take the partial derivative of the cost. The gradient vector of j entries points us in the direction of the steepest climb on each dimension j in β.

$$\partial Cost(\beta) / \partial \beta j = (-1/m)\sum_{i=1}^{m}[\sigma(\beta_t x_i) - y_i) x_{ij}$$

We iteratively reduce the gradient multiplied by a little value from our cost because we don't know how far we have to go and don't want to overstep the minimum.

$$Cost(\beta) = Cost(\beta) - \alpha(\partial Cost(\beta) / \partial \beta j)$$

Regularization:

Regularization is a term that refers to any change to a learning system that improves performance on unknown datasets. If we add too many features, the model will fit the data very well such that it will fail to generalize new data thus high variance. Hence, regularization is required to inject bias into the model and reduce variance. This can be accomplished by adding a penalty term to the loss function, which essentially decreases the coefficient estimates. To ensure shrinkage we add the penalty term $\lambda\sum\beta_{j2}$ to the loss function. The new loss function is as follows:

$$Cost(\beta) = (-1/m)\sum_{i=1}^{m} [y_i log(\sigma(\beta_t x_i)) + (1 - y_i) * log(1-\sigma(\beta_t x_i))])] + \lambda\sum\beta_j^2$$

Regularization is also applied on the gradient descent as follows:

$$Cost(\beta) = Cost(\beta) - \alpha(\partial Cost(\beta) / \partial \beta j + \lambda\sum\beta_j^2)$$

Model Training Details

Iterations:
The number of iterations chosen for the gradient descent algorithm were 2000.

Leaning Rate:
The learning rate i.e. alpha was chosen to be 0.5

Lambda:
Lambda was chosen to be 10.

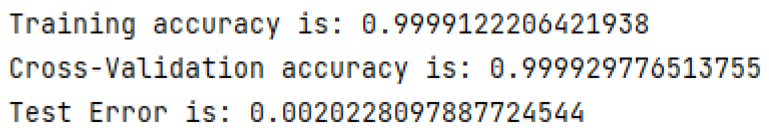
Model Output

Training Accuracy:
Training Accuracy attained after running 1000 iterations on gradient descent with lambda is equal to 99.9912 % while with 2000 iterations, it is equal to 99.9953%

Cross-Validation Accuracy:
Cross-Validation Accuracy attained is equal to 99.9929 % on 1000 iterations. With 2000 iterations accuracy goes to 99.9912%

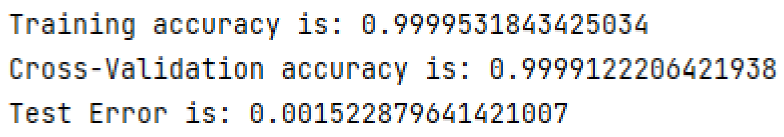
Test Error:
Test Error attained is equal to 0.002022 on 1000 iterations. With 2000 iterations test error is 0.001522.

<u>On 1000 iterations</u>



```
Training accuracy is: 0.9999122206421938
Cross-Validation accuracy is: 0.999929776513755
Test Error is: 0.0020228097887724544
```

On 2000 iterations

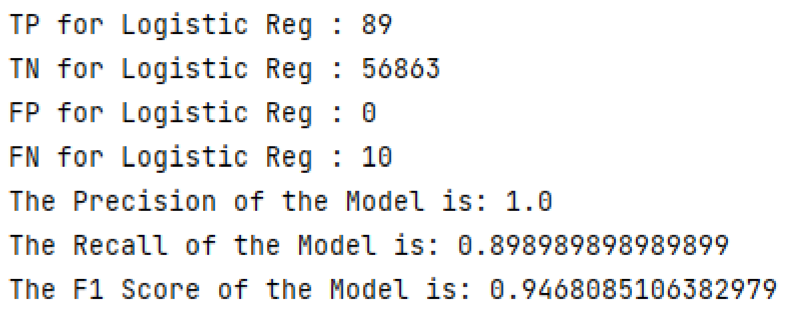


```
Training accuracy is: 0.9999531843425034
Cross-Validation accuracy is: 0.9999122206421938
Test Error is: 0.001522879641421007
```

Error Metrics:

Prediction model is run with trained data by 2000 iteration as it has better accuracy.

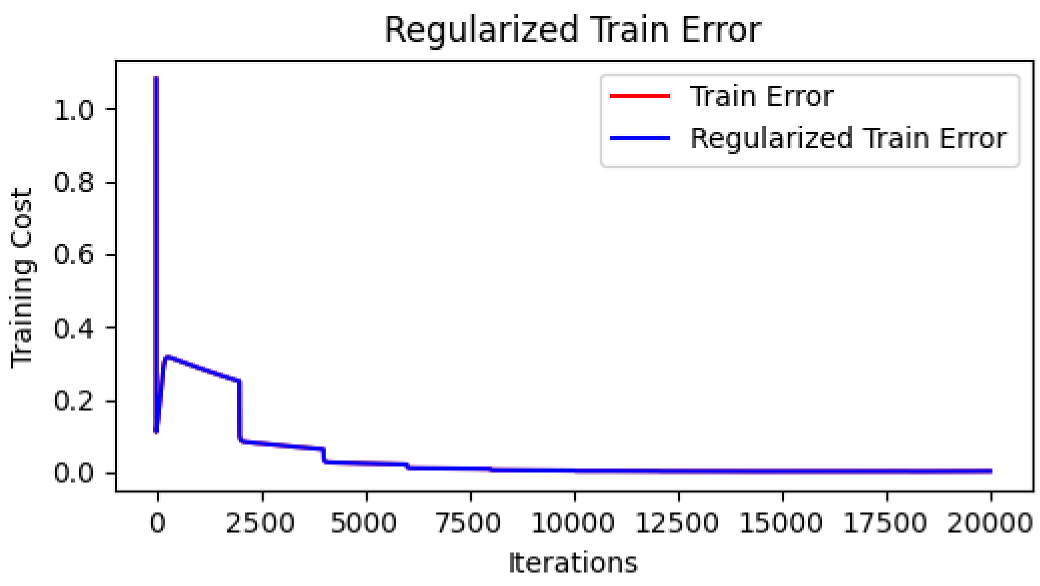
1. Number of True Positives came out to be 89
2. Number of False Positives came out to be 56863
3. Number of True Negatives came out to be 0
4. Number of False Negatives came out to be 10
5. The Precision of the Model is 1.0
6. The Recall of the Model is 0.898989898989899
7. The F1 Score of the Model is 0.9468085106382979



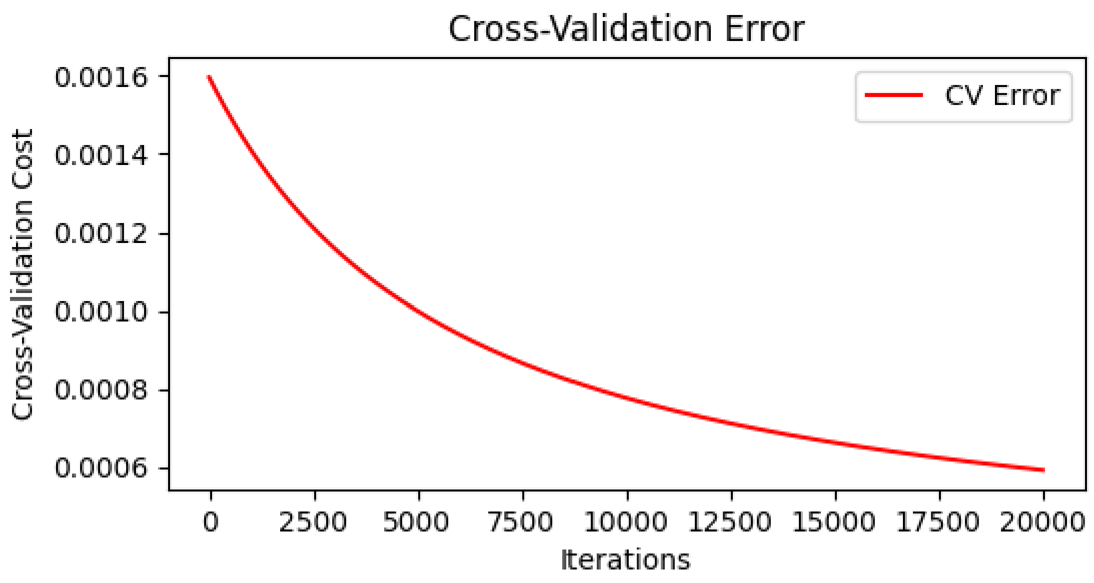
```
TP for Logistic Reg : 89
TN for Logistic Reg : 56863
FP for Logistic Reg : 0
FN for Logistic Reg : 10
The Precision of the Model is: 1.0
The Recall of the Model is: 0.898989898989899
The F1 Score of the Model is: 0.9468085106382979
```

Plotting Graphs

Train Error and Regularized Train Error vs Iterations:



Cross-Validation Error vs Iterations:



Python Code

Annex-A

Instruction to run code:

The code has been divided into sections. The first section includes the Data Pre-Processing. Second section includes the functions to Train the model named as ‘sigmoid’, ‘cost’, ‘gradient descent’ and ‘accuracy’. The next section is for Cross-Validation having same functions with slight changes to their names. The next section is to find the Test Error. Last section is for plotting the graphs. The instructions to run the code are as under:

- Code should be run as given.
- The value for iterations can be changed from the implementation part of the code.
- The value for lambda can be changed from the implementation part of the code.
- Similarly, the value for alpha can also be changed from implementation part.
- The trained parameters are saved in files i.e. ‘theta.npy’ and ‘bias.npy’ which can be used to Cross-Validate and Test the model and later Predict the model as well.

Annex-B

Training code:

```python
import numpy as np
# Importing 'NumPy'Library
import matplotlib.pyplot as plt
# Importing 'MatPlotlib' Library
import pandas as pd
# Importing 'Pandas' Library

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#**************************************************************************
*********************************************
# Importing the Data
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#**************************************************************************
*********************************************

# For the Data having Class '0'
#'''''''''''''''''''''''''''''''

dataset = pd.read_csv('E:/books/Machine Learning#/project/creditcard.csv')
```

```
# Importing data from .csv file
data = dataset.to_numpy()
# Converting data to numpy arrays
df = pd.DataFrame(data)
# Converting arrays to Dataframe
df.sample(frac=1)
# Sampling the Dataframe
x0, y0 = df.shape
# Assigning rows & columns to variables

# print(df.shape)
# Taking shape of data as output
# print(X.shape,Y.shape)
# Taking shape of X and Y as output
Y0_drop = df.loc[:, y0-1]
# Assigning the dropped column to variable
df.drop(df.columns[29], axis=1, inplace=True)
# Dropping the 'Labels' column from data


# For the Data having Class '1'
#''''''''''''''''''''''''''''''''

dataset1 = pd.read_csv('E:/books/Machine
Learning#/project/creditcard1.csv')   # Importing data from .csv file
data1 = dataset1.to_numpy()
# Converting data to numpy arrays
df1 = pd.DataFrame(data1)
# Converting arrays to dataframe
df1.sample(frac = 1)
# Sampling the dataframe
x1,y1 = df1.shape
# Assigning rows & columns to variables
# print(df1.shape)
# Taking shape of data as output
# print(X1.shape,Y1.shape)
# Taking shape of X and Y as output
Y1_drop = df1.loc[:, y1-1]
# Assigning dropped column to variable
df1.drop(df1.columns[29], axis=1, inplace=True)
# Dropping the 'Labels' column from data


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#***************************************************************************
*********************************************
# Scaling the Data
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#***************************************************************************
*********************************************

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#Scaling the data with class '0' / NORMALIZATION:
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

scaled_X = df.copy()
# Make a copy of X
for columns_X in df.columns:
# 'For' loop to scale all columns
```

```
    max_value = df[columns_X].max()
# Finding the maximum value from columns
    min_value = df[columns_X].min()
# Finding the minimum value from columns
    scaled_X[columns_X] = (df[columns_X]-min_value) / (max_value-min_value)
# Formula for MinMax Scaling

# print(scaled_X)
# Taking scaled data for X as output

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Scaling the data with class '1' / NORMALIZATION:
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

scaled_X1 = df1.copy()
# Make a copy of X1
for columns_X1 in df1.columns:
# 'For' loop to scale all columns
    max_value1 = df1[columns_X1].max()
# Finding maximum value from columns
    min_value1 = df1[columns_X1].min()
# Finding minimum value from columns
    scaled_X1[columns_X1] = (df1[columns_X1]-min_value1)/(max_value1-
min_value1)  # Formula for MinMax Scaling

# print(scaled_X1)
#Taking scaled data for X1 as output

#""""""""""""""""""""""""""""""""""""""""""""""""""""""
# Adding the 'Label' columns that were dropped earlier
#""""""""""""""""""""""""""""""""""""""""""""""""""""""

scaled_X['Class'] = Y0_drop
# Adding labels for dataset with '0' class
scaled_X1 ['Class'] = Y1_drop
# Adding labels for dataset with '1' class


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#***************************************************************************
*********************************************
# Splitting the Data
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#***************************************************************************
*********************************************

# For the Data having Class '0'
#==============================

# Splitting the Features' dataset into Training, Cross-Validation and Test
set :

train_0, CV_0, test_0 =
np.split(scaled_X.sample(frac=1),[int(0.6*len(scaled_X)),
                                                  int(0.8*len(scaled_X))])

# For the Data having Class '1'
#===============================
```

```
# Splitting the Features' dataset into Training, Cross-Validation and Test
set

train_1, CV_1, test_1 =
np.split(scaled_X1.sample(frac=1),[int(0.6*len(scaled_X1)),
                                                  int(0.8*len(scaled_X1))])

#Making final datasets by concatenating:
#====================================

train_frames = [train_0, train_1]
# Assigning variable to lists of training dataset
train = pd.concat(train_frames)
# Concatenating the two lists

CV_frames = [CV_0, CV_1]
# Assigning variable to lists of CV dataset
CV = pd.concat(CV_frames)
#Concatenating the two lists

test_frames = [test_0, test_1]
# Assigning variable to lists of test dataset
test = pd.concat(test_frames)
# Concatenating the two lists

n_samples_train, n_features_train = train.shape
# Assigning rows as samples and columns as features
train_X = train.iloc[:, 0:n_features_train-1]
# Assigning the input data to variable
train_Y = train.iloc[:, -1]
# Assigning the output/classifiers to variable

n_samples_CV, n_features_CV = CV.shape
# Assigning rows as samples and columns as features
CV_X = CV.iloc[:, 0:n_features_CV-1]
# Assigning the input data to variable
CV_Y = CV.iloc[:, -1]
# Assigning the output/classifier to variable

n_samples_test, n_features_test = test.shape
# Assigning rows as samples and columns as features
test_X = test.iloc[:, 0:n_features_test-1]
# Assigning the input data to variable
test_Y = test.iloc[:, -1]
# Assigning the output/classifier to variable

np.save('test_x',test_X)
# Saving the test set
np.save('test_y',test_Y)
# Saving the labels of test set

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
# Training the Data
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++
train_error = []
# Empty list for Training Error
train_r_error = []
# Empty list for Regularized Training Error
L_train = []
# Empty list for lambdas

def sigmoid(X, theta, B):
# Defining function for Sigmoid
    z = np.dot(theta, X.T) + B
# Defining hypothesis
    return 1/(1+np.exp(-(z)))
# Return Sigmoid

def regularization(X, theta):
# Defining function for Regularization
    m = len(X)
# Assigning length of 'X' set to variable
    reg = (L1/(2*m))*(np.sum(theta)**2)
# Formula for Regularization
    return reg
# Returning Regularization

def cost(X, y, theta):
# Defining function to compute Cost
    h1 = sigmoid(X, theta, B)
# Calling Sigmoid function
    cost_f = -(1 / len(X)) * \
             np.sum(y * np.log(h1) + (1 - y) * np.log(1 - h1))
# Formula for Cost
    cost_f_r = (-(1 / len(X))) * \
               (np.sum(y * np.log(h1) + (1-y) * np.log(1 - h1)))\
               + regularization(X, theta)
# Formula for cost with regularization

    train_error.append(cost_f)
# Appending Training Error to the list
    train_r_error.append(cost_f_r)
# Appending Regularized Training error to list
    L_train.append(L1)
    return train_r_error
# Returning Regularized Training Error

def gradient_descent(X, y, theta, B, alpha, iterations):
# Defining function to compute Gradient Descent
    m = int(len(X))
# Assigning length of 'X' set to a variable
    J = [cost(X, y, theta)]
# Calling 'Cost' function in a list
    for i in range(0, iterations):
# 'For' loop until no. of iterations
        h = sigmoid(X, theta, B)
# Calling Sigmoid function
        for i in range(0, len(X.columns)):
# 'For' loop until length of 'X' set
            theta[i] -= (alpha/m) * np.sum((h-y)*X.iloc[:, i]) + \
                        ((L1*theta[i])/m)
# Formula to 'Update Weights'
            B -= (alpha/m) * np.sum(h-y)
```

```
# Formula to 'Update Bias'
        J.append(cost(X, y, theta))
# Appending cost to List
        np.save('theta', theta)
# Saving Weights as .npy file
        np.save('bias', B)
# Saving Bias units as .npy file
    return J, theta
# Returning Cost list and Weights

def accuracy(X, y, theta, alpha, iterations):
# Defining function to compute accuracy
    J = gradient_descent(X, y, theta, B, alpha, iterations)
# Calling gradient Descent function
    h = sigmoid(X, theta, B)
# Assigning sigmoid function a variable
    for i in range(len(h)):
# 'For' loop until length of sigmoid
        h[i]=1 if h[i] >= 0.5 else 0
# Separating labels using threshold
    y = list(y)
# Converting the 'y' set into a list
    acc = np.sum([y[i] == h[i].any() for i in range(len(y))])/len(y)
# Formula to compute accuracy
    return J, acc
# Returning Accuracy

#***********************************
# Implementation with Training set:
#***********************************

m = train_X.shape[1]
# Assigning Columns to 'm'
n = train_X.shape[0]
# Assigning Rows to 'n
B = 0
# The initial value of base unit is '0'
theta = 0.1*np.random.rand(m)
# Defining initial thetas/weights
alpha = 0.5
# Defining the learning rate
for L1 in np.arange(0, 10, 1):
# 'For' loop to iterate through lambda
    J, acc = accuracy(train_X, train_Y, theta, alpha, 2000)
# Calling function for training model
print('Training accuracy is:', acc)
# Printing the output accuracy for training


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
# Cross-Validating the Data
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++

CV_error = []
```

```python
# Empty list for Cross-Validation Error
L_CV = []
# Empty list for Lambdas

def sigmoid_CV(X, theta_CV, B_CV):
# Defining Sigmoid function for CV
    z = np.dot(theta_CV, X.T) + B_CV
# Defining hypothesis for CV
    return 1/(1+np.exp(-(z)))
# Returning Sigmoid

def cost_CV(X, y, theta_CV):
# Defining function to compute cost for CV
    h1 = sigmoid_CV(X, theta_CV, B_CV)
# Calling Sigmoid function & assigning a variable
    cost_f_cv =  -(1 / len(X)) * np.sum(y * np.log(h1)
                                         + (1 - y) * np.log(1 - h1))
# Formula to compute cost for CV
    CV_error.append(cost_f_cv)
# Appending the error to the empty CV list
    L_CV.append(L2)

    return CV_error
# Returning the CV error

def gradient_descent_CV(X, y, theta_CV, B_CV, alpha, iterations):
# Defining function to compute Gradient Descent
    m = int(len(X))
# Assigning length of 'X' set to a variable
    J_CV = [cost_CV(X, y, theta_CV)]
# Calling 'Cost' function in a list
    for i in range(0, iterations):
# 'For' loop until no. of iterations
        h = sigmoid_CV(X, theta_CV, B_CV)
# Calling Sigmoid function
        for i in range(0, len(X.columns)):
# 'For' loop until length of 'X' set
            theta_CV[i] -= (alpha/m) * np.sum((h-y)*X.iloc[:, i])
# Formula to 'Update Weights'
            B_CV -= (alpha/m) * np.sum(h-y)
# Formula to 'Update Bias'
        J_CV.append(cost_CV(X, y, theta_CV))
# Appending cost to List
    return J_CV, theta_CV
# Returning Cost list and Weights

def accuracy_CV(X, y, theta_CV, alpha, iterations):
# Defining function for accuracy
    J = gradient_descent_CV(X, y, theta_CV, B_CV, alpha, iterations)
# Calling gradient Descent function
    h = sigmoid_CV(X, theta_CV, B_CV)
# Calling Sigmoid function
    for i in range(len(h)):
# 'For' loop until length of sigmoid
        h[i]=1 if h[i] >= 0.5 else 0
# Separating labels using threshold
    y = list(y)
# Converting labels into a list
    acc = np.sum([y[i] == h[i].any() for i in range(len(y))])/len(y)
# Formula to compute accuracy
    return J, acc
```

```
# Return accuracy


#*********************************************
# Implementation with Cross-Validation set:
#*********************************************

theta_CV = np.load('theta.npy')
# Defining weights for Cross-Validation
B_CV = np.load('bias.npy')
# Defining Bias units for Cross-Validation
alpha_CV = 0.5
# Defining the learning rate
for L2 in np.arange(0,10,1):
# 'For' loop to iterate through lambda
    J_CV, acc_CV = accuracy_CV(CV_X, CV_Y, theta_CV, alpha, 2000)
# Calling function for CV model
print('Cross-Validation accuracy is:', acc_CV)
# Taking the accuracy of model as output

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
# Testing the Data
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++

def sigmoid_test(X, theta_test, B_test):
# Defining Sigmoid function for test
    z = np.dot(theta_test, X.T) + B_test
# Defining hypothesis for test
    return 1/(1+np.exp(-(z)))
# Returning Sigmoid

def cost_test(X, y, theta_test):
# Defining function to compute cost for test
    h2 = sigmoid_test(X, theta_test, B_test)
# Assigning Sigmoid function a variable
    cost_f_t =  -(1 / len(X)) * np.sum(y * np.log(h2)
                                     + (1 - y) * np.log(1 - h2))
# Formula to compute cost for test
    return cost_f_t
# Returning the test error

def accuracy_test(X, y, theta_test, alpha, iterations):
# Defining the accuracy function for test set
    C_test = cost_test(X, y, theta_test)
# Calling the cost function
    h = sigmoid_CV(X, theta_test, B_test)
# Calling the sigmoid function
    for i in range(len(h)):
# 'For' loop until the length of Sigmoid
        h[i]=1 if h[i] >= 0.5 else 0
# Separating labels using threshold
    y = list(y)
# Converting labels into a list
    acc_cv = np.sum([y[i] == h[i].any() for i in range(len(y))])/len(y)
```

```
# Formula for accuracy
    return C_test, acc_cv
# Return accuracy


#********************************************
# Implementation with Test set:
#********************************************

theta_test = np.load('theta.npy')
# Defining thetas for test error
B_test = np.load('bias.npy')
# Defining Bias units for test error
test_error = cost_test(test_X, test_Y, theta_test)
# Calling function to compute test error
print('Test Error is:', test_error)
# Taking Test Error as output


#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
# Plotting Graphs
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++


figure, axis = plt.subplots(2, 2)
# Making Sub-Plots

axis[0, 0].plot(train_error, label='Train Error', color='r')
# Making plot for Train error
axis[0, 0].set_title("Train Error")
# Setting title for graph
axis[0, 0].set(xlabel="Iterations",ylabel="Training Cost")
# Setting x and y labels for subplots
axis[0, 0].plot(train_r_error, label='Regularized Train Error', color='b')
# Making plot for Regularized Train error
axis[0, 0].set_title("Regularized Train Error")
# Setting title for graph
axis[0, 0].legend(loc='upper right')
# Setting the legend for graph

axis[1, 0].plot(CV_error, label='CV Error',color = 'r' )
# Making plot for CV error
axis[1, 0].set_title("Cross-Validation Error")
# Setting title for graph
axis[1, 0].set(xlabel="Iterations",ylabel="Cross-Validation Cost")
# Setting x and y labels for subplots
axis[1, 0].legend(loc='upper right')
# Setting the legend for graph


axis[0, 1].plot(L_train, train_error, label='Train Error', color='y')
# Making plot for Train error
axis[0, 1].set_title("Train Error vs Lambda")
# Setting title for graph
axis[0, 1].set(xlabel="Lambda",ylabel="Training Cost")
```

```
# Setting x and y labels for subplots
axis[0, 1].legend(loc='upper right')
# Setting the legend for graph


axis[1, 1].plot(L_CV, CV_error, label='CV Error',color = 'r' )
# Making plot for CV error
axis[1, 1].set_title("Cross-Validation Error vs Lambda")
# Setting title for graph
axis[1, 1].set(xlabel="Lambda",ylabel="Cross-Validation Cost")
# Setting x and y labels for subplots
axis[1, 1].legend(loc='upper right')
# Setting the legend for graph


plt.show()
# Show plots
```

Annex-C

Prediction code:

```
import numpy as np
# Importing 'NumPy'Library

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
# Prediction Function
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++


def sigmoid(X, theta, B):
# Defining function for Sigmoid
    z = np.dot(theta, X.T) + B
# Defining hypothesis
    return 1/(1+np.exp(-(z)))
# Return Sigmoid

def predict(X, y, threshold, theta, B):
# Defining prediction function
    Y_pred = sigmoid(X, theta, B)
# Calling Sigmoid function to get predicted labels
    Y_pred = Y_pred > threshold
# Setting a condition for predicted labels
    y = np.array(y)
# Converting Actual labels into set of array
    Y_pred = np.array(Y_pred)
# Converting Predicted labels into set of array

    tp = np.sum((y == 1) & (Y_pred == 1))
# Condition to get True Positives
    tn = np.sum((y == 0) & (Y_pred == 0))
```

```
# Condition to get True Negatives
    fp = np.sum((y == 0) & (Y_pred == 1))
# Condition to get False Positives
    fn = np.sum((y == 1) & (Y_pred == 0))
# Condition to get False Negatives

    print('TP for Logistic Reg :', tp)
# Taking no.of True Positives as output
    print('TN for Logistic Reg :', tn)
# Taking no.of True Negatives as output
    print('FP for Logistic Reg :', fp)
# Taking no.of False Positives as output
    print('FN for Logistic Reg :', fn)
# Taking no.of False Negatives as output

    precision = tp /(tp + fp)
# Formula to compute precision
    recall = tp / (tp + fn)
# Formula to compute recall
    f1 = 2 * (precision * recall) / (precision + recall)
# Formula to compute F1 Score

    print('The Precision of the Model is:', precision)
# Taking Precision as output
    print('The Recall of the Model is:', recall)
# Taking Recall as output
    print('The F1 Score of the Model is:', f1)
# Taking F1 Score as output

    return Y_pred, f1 , precision, recall
# Return the Error Metrics

#********************************************
# Implementation with Test set:
#********************************************

test_X = np.load('test_X.npy')
# Importing Test_X set
test_Y = np.load('test_Y.npy')
# Importing Test_Y set
theta_test = np.load('theta.npy')
# Defining thetas for test error
B_test = np.load('bias.npy')
# Defining Bias units for test error
predict(test_X, test_Y, 0.5, theta_test, B_test)
# Calling Predict Function to get metrics
```