**CS 464 – Introduction to Machine Learning**

**Homework 2**

Dora Tütüncü • 21801687 • Section 2

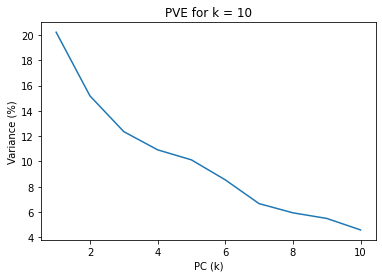
**QUESTIONS**

1. **PCA & Digits**

**(1.1)**

The PVE for k = 10 is given as in the following graph. It can be said that the variance decreases as the principal component analysis is made for more principal components. Since principal components are basically the eigenvectors of the covariance matrix of the dataset, eigenvectors sorted with ascending order with respect to their eigenvalues will have greater variance due to the proportion of variance captured by kth PC given in the equation below.

Where λi corresponds to an eigenvalue of the covariance matrix, and λk is the eigenvalue of the kth eigenvalue. Hence, the eigenvector with an eigenvalue of larger index will have hence less variance because it has a lesser value considering that eigenvalues had been sorted in ascending order. When all the proportion of variances are plotted, the following proportion of variance explained (PVE) plot is obtained:



Each principle component can be reshaped in a 28 by 28 matrix. Since the dataset contains 28x28 pixel images of different digits, such a principle component would correspond to a deconstructed version of the key features of any digit. Since this PCA was done for all digit images in the dataset, the eigenvectors would correspond to the general of structure of a digit’s image. These eigenvectors correspond to images as seen below. These images do not make sense at first glance, but is still useful information when reconstructing a digit, especially if PCA is done for a single, meaningful image.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C:\Users\DORA\Desktop\w0t\pc.png | C:\Users\DORA\Desktop\w0t\pc1.png | C:\Users\DORA\Desktop\w0t\pc2.png | C:\Users\DORA\Desktop\w0t\pc3.png | C:\Users\DORA\Desktop\w0t\pc5.png |
| C:\Users\DORA\Desktop\w0t\pc6.png | C:\Users\DORA\Desktop\w0t\pc7.png | C:\Users\DORA\Desktop\w0t\pc8.png | C:\Users\DORA\Desktop\w0t\pc9.png | C:\Users\DORA\Desktop\w0t\pc10.png |

**(1.2)**

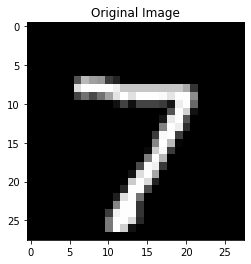
The PVE for the corresponding principle component numbers are given in the following results. It is clearly seen as the number of principal components increase, the proportional variance decreases.

|  |  |  |
| --- | --- | --- |
| C:\Users\DORA\Desktop\w0t\pve8.png | C:\Users\DORA\Desktop\w0t\pve16.png | C:\Users\DORA\Desktop\w0t\pve32.png |
| C:\Users\DORA\Desktop\w0t\pve64.png | C:\Users\DORA\Desktop\w0t\pve128.png | C:\Users\DORA\Desktop\w0t\pve256.png |

**(1.3)**

As explained in (1.1), the principle components, or rather eigenvalues, represent the general features of an image, which when dot multiplied with the reduced image, can reconstruct the original image. It is expected that the larger the principle component number, the more accurate results will be obtained. This is clearly seen if we take the example of a single digit image of the dataset. The following results are obtained with PCA analysis of a digit with different number of principle components. As expected, since the variance decreases almost to 0 after reaching a certain point, the image gets reconstructed more accurately, but naturally the bigger the number of principle components, the reconstructed image will be more complex and have a larger dimension. In order to reduce the image without losing too much of the original shape, one can use 10 or 50 principle components, as these images closely represent the original one, despite being reduced in dimension.

|  |  |  |  |
| --- | --- | --- | --- |
| C:\Users\DORA\Desktop\w0t\imrec1.png | C:\Users\DORA\Desktop\w0t\imrec2.png | C:\Users\DORA\Desktop\w0t\imrec3.png | C:\Users\DORA\Desktop\w0t\imrec4.png |
| C:\Users\DORA\Desktop\w0t\imrec5.png | C:\Users\DORA\Desktop\w0t\imrec6.png | C:\Users\DORA\Desktop\w0t\imrec7.png | C:\Users\DORA\Desktop\w0t\imrec8.png |

****

1. **Linear and Polynomial**

**(2.1)**

The matrix y corresponds to the matrix labels, ground truths or actual values of the data in the training dataset, expressed as below. Assume that there are n different data with their labels specified.

The matrix X corresponds to the modified matrix of features of the data in the training dataset, expressed as below. Assume that there are n different data with m features each. We add the row of 1’s because since this is linear regression, there needs to be a constant value, which can be only obtained if 1’s are used.

As for the β value, it will be used to represent the trained coefficient for the linear regression model. Assuming that we had m different features, the β matrix will represent the coefficients of the linear regression model in the mth dimension.

Hence, the linear regression model’s loss can be defined as, where y is the actual values, and is the predicted value as a multivariate regression model, given in open form as well.

The aim is to minimize this loss. Minimizing this loss will give the general closed form solution for the multivariate regression model, which can be achieved by taking the partial derivative of with respect to, and setting it to 0. The calculations are given below. Note that some of the derivative relationships for matrices and their transform were used in the calculations, but is not specified here.

From this equation, if we find the value of with respect to the other values, we obtain the trained coefficients for the multinomial linear regression, which is the solution we seek.

**(2.2)**

For the given dataset, the rank was found to be 5, which means that is 5 by 5 dimensional matrix. Since we have one column corresponding to 0th dimension, this tells that the best suited linear regression model is 4 dimensional. This makes sense as there are 4 different features associated to a datum.

**(2.3)**

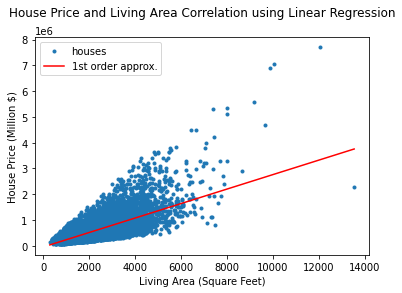
Trained Beta Coefficients for 1st Order Linear Regression are:

b0 = -43580.743094474354

b1 = 280.6235678974484

The Mean Square Error for 1st Order LR is: 201200997618.10632

This regression model shows that there are many houses accumulating in the price range of 0-1 million dollars and to 6000 square feet living area, as the slope is not too large.



**(2.4)**

Trained Beta Coefficients for 2nd Order Linear Regression are:

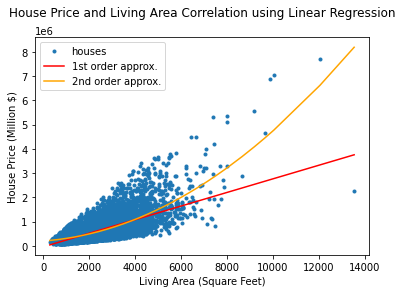
b0 = 199222.27930548566

b1 = 67.99409468579292

b2 = 0.038581260937201815

The Mean Square Error for 2nd order LR is: 206577201240.17664

This regression model shows again that there are many houses accumulating in the price range of 0-1 million dollars and to 6000 square feet living area, however the error is higher in this model, hence we can conclude that it disregards some frequent instances out of the above mentioned range, causing an accumulation in error.



1. **Logistic Regression**

**(3.1)**

Confusion Matrix for learning rate = 1e-05 :

Number of True Positives: 83

Number of True Negatives: 1100

Number of False Positives: 2

Number of False Negatives: 15

Accuracy is 98.58333333333333 %

Confusion Matrix for learning rate = 0.0001 :

Number of True Positives: 83

Number of True Negatives: 1100

Number of False Positives: 2

Number of False Negatives: 15

Accuracy is 98.58333333333333 %

Confusion Matrix for learning rate = 0.001 :

Number of True Positives: 84

Number of True Negatives: 1099

Number of False Positives: 3

Number of False Negatives: 14

Accuracy is 98.58333333333333 %

Confusion Matrix for learning rate = 0.01 :

Number of True Positives: 88

Number of True Negatives: 1091

Number of False Positives: 11

Number of False Negatives: 10

Accuracy is 98.25 %

Confusion Matrix for learning rate = 0.1 :

Number of True Positives: 89

Number of True Negatives: 1071

Number of False Positives: 31

Number of False Negatives: 9

Accuracy is 96.66666666666667 %

It seems that the learning rate = 0.001 gives the best results, as the accuracy and precision rates are as high as small rates, and computationally less expensive than the small rates, and there is no decrease in precision like in the larger rates.

=> Performance Metrics for Full Batch Gradient Ascent are:

Number of True Positives: 84

Number of True Negatives: 1099

Number of False Positives: 3

Number of False Negatives: 14

Accuracy is 98.58333333333333 %

Precision is 96.55172413793103 %

Recall is 85.71428571428571 %

Negative Predictive Value is 98.74213836477988 %

False Positive Rate is 0.27223230490018147 %

False Discovery Rate is 3.4482758620689653 %

F1 measure is 0.908108108108108

F1 measure is 0.9655172413793103

Total time elapsed for full batch gradient ascent is 0.3534 seconds. Full batch analysis is usually faster, however it may be prone to errors as it might have gotten stuck in a local extremum. Mini batch and stochastic may also be used, but they are normally computationally more expensive.

**(3.2)**

Performing the Mini Patch Gradient Ascent for Batch Size = 100, the following Performance Metrics are obtained:

Number of True Positives: 84

Number of True Negatives: 1099

Number of False Positives: 3

Number of False Negatives: 14

Accuracy is 98.58333333333333 %

Precision is 96.55172413793103 %

Recall is 85.71428571428571 %

Negative Predictive Value is 98.74213836477988 %

False Positive Rate is 0.27223230490018147 %

False Discovery Rate is 3.4482758620689653 %

F1 measure is 0.908108108108108

F1 measure is 0.9655172413793103

Total time elapsed for mini batch gradient ascent is 11.6631 seconds

The results received here are the same as the full batch gradient ascent, so it seems that the results are accurate. However note that these computations were done for 1000 iterations. When less iterations were selected, (n=10), it seemed that mini batch gradient ascent was more effective. So the same results can be obtained in the batch gradient ascent with less iteration.

=> Performing the Stochastic Gradient Ascent, the following Performance Metrics are obtained:

Number of True Positives: 84

Number of True Negatives: 1099

Number of False Positives: 3

Number of False Negatives: 14

Accuracy is 98.58333333333333 %

Precision is 96.55172413793103 %

Recall is 85.71428571428571 %

Negative Predictive Value is 98.74213836477988 %

False Positive Rate is 0.27223230490018147 %

False Discovery Rate is 3.4482758620689653 %

F1 measure is 0.908108108108108

F1 measure is 0.9655172413793103

Total time elapsed for stochastic gradient ascent is 124.3753 seconds

The result here is again the same, however it is too computationally expensive, given that it took over 2 minutes to complete. Mini-batch should be preferred compared to this method. Note that stochastic gradient ascent is the same thing as mini-batch gradient ascent with batch size =1, so the same function was used by altering the batch size to 1.

**(3.3)**

There is a possibility that the given datasets might be uneven; that is, for a binary classification example like this, for example, there might be an unbalanced number of True and False values in the dataset, which may affect the training process negatively. The accuracy, precision and recall measure unfortunately may give misleading results, telling that the accuracy is very high, but in truth, the model might have become too rigid, and when exposed to newer test data, the model fail may give bad results. The F1 and F2 measures may be more informative regarding this, as it allows us to weigh the precision and recall, and allows us to receive a measure that considers both these facts, so basically they might be useful when determining if there is balance between recall and precision, and by what factor. False positive rate, FPR, determines how much data that is actually negative has been classified incorrectly among all actually negative values. On the other hand, FPR may be used with recall to determine the Receiver Operating Characteristic curve of the data. This might be helpful in determining how conservative or liberal our classifier might be; which is a measure of determining how evenly distributed the TP, TN, FN and FP values are. Negative Predictive Value, NPV, may be useful if we want to determine how accurately we can determine that a negative result is actually correct among all negatively classified objects. This is an important measure as if this is too low, we’d be classifying data instances that are actually positive as negative, and for this example, and these people would be able to get away with tax fraud. False Discovery Rate is also another important metric, because it determines how much data we’ve classified as positive among those who are actually negative, among all data that has been predicted as positive. This would mean that, if this value is too high, for this example, we’d be accusing the wrong people of tax fraud. This is actually the compliment probability of precision.

**CODES**

**Initialization for all Questions**

#------------------------------INITIALIZATION----------------------------------

from google.colab import drive

drive.mount('/content/gdrive')

!ls /content/gdrive/My\ Drive/Dora/Bilkent/CS464/HW2 # Use YOUR OWN DIRECTORY!!

import os

import csv

import math

import random

import operator

import pdb

import time

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

from timeit import default\_timer as timer

from pylab import imshow,title,figure,show, gray

np.random.seed(123)

root = '/content/gdrive/My Drive/Dora/Bilkent/CS464/HW2'

**Code for Question 1**

#-------------------------------QUESTION 1--------------------------------------

#-------------------------Question 1.1-----------------------------

print("\n------------------------- QUESTION 1 -------------------------\n" )

digits\_csv = os.path.join(root, 'digits.csv')

#read the csv files with the first column being the header

pd\_digits = pd.read\_csv(digits\_csv, header=0)

pd\_digits\_new = pd\_digits.drop(columns = "label")

digits = pd\_digits\_new.to\_numpy()

#uncomment the option below to see all the columns clearly.

#pd.set\_option('display.max\_columns', 28)

pd.reset\_option('display.max\_columns')

#PCA as a function, returns eigenvlaues, principle component matrix, and the reduced matrix

def PCA(data, k\_pcomponents) :

    #computing eigenvalues and eigenvectors of the covariance matrix

    data\_meaned = (data - np.mean(data.T,axis=1)).T # subtract the mean (along columns)

    eigenvalues, eigenvectors = np.linalg.eigh(np.cov(data\_meaned))

    p = np.size(eigenvectors,axis=1)

    sorted\_index = np.argsort(eigenvalues)[::-1] #sort eigenvalue indexes in ascending order

    #sorting the eigenvectors according to the sorted eigenvalue indexes

    eigenvectors = eigenvectors[:,sorted\_index]  # sorting eigenvectors

    eigenvalues = eigenvalues[sorted\_index] # sorting eigenvalues

    if k\_pcomponents < p and k\_pcomponents >= 0:

        eigenvectors = eigenvectors[:,range(k\_pcomponents)] #obtaining PCs

        eigenvalues = eigenvalues[range(k\_pcomponents)]

    data\_reduced = np.dot(eigenvectors.T, data\_meaned) #projectioning data into the reduced space

    return eigenvalues, eigenvectors, data\_reduced

def PVE(principal\_eigenvalues) :

    sum\_eigenvalues = principal\_eigenvalues.sum(axis=0)

    variance\_matrix = principal\_eigenvalues/sum\_eigenvalues

    #each column in variance\_matrix corresponds to the PVE of a PC

    return variance\_matrix

def Reshape28x28(principal\_components, k\_pcomponents) :

    principal\_components = principal\_components.T

    reshaped\_matrix = []

    for k in range(k\_pcomponents) :

        square\_component = principal\_components[k].reshape(28, 28)

        reshaped\_matrix.append(square\_component)

    return reshaped\_matrix

k\_pcomponents = 10

digits\_eigenvalues, digits\_pcomponents , digits\_reduced = PCA(digits, k\_pcomponents)

digits\_variance = PVE(digits\_eigenvalues)

reshaped\_pcomponents = Reshape28x28(digits\_pcomponents, k\_pcomponents)

print("\nPrincipal Components Matrix, where columns refer to each principle component:")

display(pd.DataFrame(digits\_pcomponents)) #transform PC's into database

print("\nReduced Matrix")

display(pd.DataFrame(digits\_reduced))

print("\nPrincipal Values Explained Matrix, rows refer to each principle component's PVE:")

display(pd.DataFrame(digits\_variance))

print()

#this for loop is for visualizing each principal component as a 28 by 28 matrix.

for k in range(k\_pcomponents) :

    print("\nPrincipal component {} reshaped as a 28x28 matrix: ".format(k+1))

    display(pd.DataFrame(reshaped\_pcomponents[k]))

print()

myarr = []

for j in range(k\_pcomponents):

  myarr.append(j+1)

plt.plot( myarr , digits\_variance.transpose()\*100)

plt.title("PVE for k = {}".format(k\_pcomponents))

plt.ylabel('Variance (%)')

plt.xlabel('PC (k)')

plt.show()

#displaying eigenvectors of the dataset

for k in range(k\_pcomponents) :

    figure()

    imshow(reshaped\_pcomponents[k])

    title("Principal component {} shown as a image: ".format(k+1))

show()

#-------------------------Question 1.2-----------------------------

k\_matrix = [8, 16, 32, 64, 128, 256]

variance = [0]\*6

for i in range(len(k\_matrix)):

    peigens, pcomponents , reduced = PCA(digits, k\_matrix[i])

    variance[i] = PVE(peigens)

    num = k\_matrix[i]

    arr = []

    for j in range(num):

        arr.append(j+1)

    print()

    plt.plot( arr , variance[i].transpose()\*100)

    plt.title("PVE for k = {}".format(num))

    plt.ylabel('Variance (%)')

    plt.xlabel('PC (k)')

    plt.show()

#-------------------------Question 1.3-----------------------------

print("\nImage Reconstructions are: \n")

first\_image = digits[0].reshape(28, 28)

k\_pcomponents = [1,3,5,10,50,100,200,300]

def Reconstruction(image28x28, k\_pcomponent):

    eigenvalues, pcomponents , reduced = PCA(image28x28, k\_pcomponent) # image reconstruction

    recon = np.dot(pcomponents, reduced).T + np.mean(image28x28,axis=0)

    return recon

first\_digit = digits[0].reshape(28, 28)

for k\_pcomponent in k\_pcomponents:

    recon\_image = Reconstruction(first\_digit, k\_pcomponent)

    figure()

    imshow(recon\_image)

    title('PCs = '+str(k\_pcomponent))

figure()

imshow(first\_digit)

title('Original Image')

show()

**Code for Question 2**

#-------------------------------QUESTION 2--------------------------------------

print("\n------------------------- QUESTION 2 -------------------------\n" )

home\_features = os.path.join(root, 'question-2-features.csv')

home\_prices = os.path.join(root, 'question-2-labels.csv')

df\_home\_features = pd.read\_csv(home\_features, header=0)

df\_home\_prices = pd.read\_csv(home\_prices, header=0)

array\_home\_features = df\_home\_features.to\_numpy()

array\_home\_prices = df\_home\_prices.to\_numpy()

ones\_array = np.array([np.ones(len(array\_home\_features[:]))])

array\_concat\_features = np.concatenate((ones\_array.T, array\_home\_features), axis = 1)

xTx = np.matmul(array\_concat\_features.transpose(), array\_concat\_features)

yTy = np.matmul(array\_home\_prices.transpose(), array\_home\_prices)

rank\_features = np.linalg.matrix\_rank(xTx)

rank\_labels = np.linalg.matrix\_rank(yTy)

print('The rank of X transpose times X is: {}'.format(rank\_features))

def SolveClosedForm(features, labels):

    first\_matrix = np.matmul(features.transpose(), features)

    first\_matrix = np.linalg.inv(first\_matrix) # this calculates ((x^T)x)^(-1), however it is a constant, so it cannot be inverted, since its the same

    second\_matrix = np.matmul(features.transpose(), labels) # this calculates (x^T)y

    predicted\_coeffs = np.matmul(first\_matrix, second\_matrix) # predicted beta set, since both are just integers, this operation is ok.

    return predicted\_coeffs

#linear regression to fit into a first order equation

print("\n1ST ORDER LINEAR REGRESSION ANALYSIS:")

array\_concat\_sqftliving = array\_concat\_features[:,[0,1]]

beta\_prediction = SolveClosedForm(array\_concat\_sqftliving, array\_home\_prices)

b0 = beta\_prediction[0][0]

b1 = beta\_prediction[1][0]

print("\nTrained Beta Coefficients for 1st Order Linear Regression are: \n b0 = {} \n b1 = {} \n ".format(b0, b1))

living\_area = array\_concat\_sqftliving[:,1]

sorted\_livingarea = np.sort(living\_area)

predicted\_prices = b0+b1\*sorted\_livingarea

plt.plot(living\_area, array\_home\_prices, '.')

plt.plot(sorted\_livingarea, predicted\_prices, 'red')

plt.title("House Price and Living Area Correlation using Linear Regression\n")

plt.ylabel('House Price (Million $)')

plt.xlabel('Living Area (Square Feet)')

plt.legend(["houses", "1st order approx."])

plt.show()

MSE = np.square(np.subtract(array\_home\_prices, predicted\_prices)).mean()

print('\nThe Mean Square Error for 1st Order LR is : {} \n'.format(MSE))

#linear regression to fit into a second order equation

print("\n2ND ORDER LINEAR REGRESSION ANALYSIS:")

array\_sqftliving\_squared = np.array([np.square(array\_concat\_sqftliving[:,1])])

array\_concatsqr\_sqftliving = np.concatenate((array\_concat\_sqftliving, array\_sqftliving\_squared.T), axis = 1)

sqrbeta\_prediction = SolveClosedForm(array\_concatsqr\_sqftliving, array\_home\_prices)

b0 = sqrbeta\_prediction[0][0]

b1 = sqrbeta\_prediction[1][0]

b2 = sqrbeta\_prediction[2][0]

print("\nTrained Beta Coefficients for 2nd Order Linear Regression are: \n b0 = {} \n b1 = {} \n b2 = {} \n ".format(b0, b1, b2))

living\_area\_sqr = np.square(living\_area)

sorted\_livingareasqr = np.sort(living\_area\_sqr)

sqrpredicted\_prices = b0 + b1\*sorted\_livingarea + b2\*sorted\_livingareasqr

plt.plot(living\_area, array\_home\_prices, '.')

plt.plot(sorted\_livingarea, predicted\_prices,'red')

plt.plot(sorted\_livingarea, sqrpredicted\_prices, 'orange' )

plt.title("House Price and Living Area Correlation using Linear Regression\n")

plt.ylabel('House Price (Million $)')

plt.xlabel('Living Area (Square Feet)')

plt.legend(["houses", "1st order approx.", "2nd order approx."])

plt.show()

MSE = np.square(np.subtract(array\_home\_prices, sqrpredicted\_prices)).mean()

print('\nThe Mean Square Error for 2nd order LR is : {}'.format(MSE))

**Code for Question 3**

#-------------------------------QUESTION 3--------------------------------------

print("\n------------------------- QUESTION 3 -------------------------\n" )

train\_features = os.path.join(root, 'question-3-features-train.csv')

train\_labels = os.path.join(root, 'question-3-labels-train.csv')

test\_features = os.path.join(root, 'question-3-features-test.csv')

test\_labels = os.path.join(root, 'question-3-labels-test.csv')

df\_train\_features = pd.read\_csv(train\_features, header=0)

df\_train\_labels = pd.read\_csv(train\_labels, header=0)

df\_test\_features = pd.read\_csv(test\_features, header=0)

df\_test\_labels = pd.read\_csv(test\_labels, header=0)

concat\_train = pd.concat([df\_train\_labels, df\_train\_features] , axis = 1)

concat\_test =  pd.concat([df\_test\_labels, df\_test\_features] , axis = 1)

array\_train\_features = concat\_train.to\_numpy()[:,1:30]

array\_train\_labels = concat\_train.to\_numpy()[:,0]

def sigmoid(score):

    return 1/(1 + np.exp(-score))

def Performance(weights, test\_features, test\_label, best\_one = False):

    w0 = weights[0]

    wi = weights[1:len(weights)]

    #this function is defined to compute the confusion matrix values for the ML models.

    prediction\_matrix = []\*len(test\_features)

    true\_positive = 0

    true\_negative = 0

    false\_negative = 0

    false\_positive = 0

    for c in range(len(test\_features)):

        score = w0 + np.sum(np.dot(test\_features[c], wi))

        fraud\_true = sigmoid(score)

        fraud\_false = 1-sigmoid(score)

        if fraud\_false > fraud\_true:

            prediction = 0

        elif fraud\_false < fraud\_true:

            prediction = 1

        elif fraud\_false == fraud\_true:

            prediction = 0

        #for i in range(len(prediction\_matrix)):

        if prediction == 1 and prediction == test\_label[c] :

            true\_positive += 1

        elif prediction == 0 and prediction == test\_label[c] :

            true\_negative += 1

        elif prediction == 1 and prediction != test\_label[c] :

            false\_positive += 1

        elif prediction == 0 and prediction != test\_label[c] :

            false\_negative += 1

    accuracy = (true\_positive+true\_negative)/(true\_negative+true\_positive+false\_negative+false\_positive)\*100

    precision = true\_positive / (true\_positive + false\_positive)\*100

    recall = true\_positive / (true\_positive + false\_negative)\*100

    npv = true\_negative / (true\_negative + false\_negative)\*100

    fpr = false\_positive / (true\_negative + false\_positive)\*100

    fdr = false\_positive / (true\_positive + false\_positive)\*100

    f1 = 2\*precision\*recall/(precision + recall)/100

    beta = 2

    f2 = (beta^2+1)\*precision\*recall/((beta^2)\*precision + recall)/100  #beta f-measure with beta = 2, giving extra weight to recall

    print('Number of True Positives: ', true\_positive)

    print('Number of True Negatives: ', true\_negative)

    print('Number of False Positives: ', false\_positive)

    print('Number of False Negatives: ', false\_negative)

    print('Accuracy is ', accuracy, ' %')

    if best\_one :

        print('Precision is ', precision, '%')

        print('Recall is ', recall, '%')

        print('Negative Predictive Value is ', npv, '%')

        print('False Positive Rate is ', fpr, '%')

        print('False Discovery Rate is ', fdr, '%')

        print('F1 measure is ', f1)

        print('F1 measure is ', f2)

def Normalize(input\_array):

    normalized\_array = np.zeros(len(input\_array))

    minimum = min(input\_array)

    maximum = max(input\_array)

    for i in range(len(input\_array)):

        normalized\_array[i] = (input\_array[i] - minimum)/(maximum - minimum)

    return normalized\_array

def BatchGradAscent(features, labels, rate):

    weights = np.zeros(len(features[0])+1)

    w0 = weights[0]

    wi = weights[1:len(weights)]

    iterations = 1000

    for iteration in range(iterations):

        score = w0 + np.dot(features, wi)

        predictions = sigmoid(score)

        error = labels - predictions

        dgradient = np.dot(features.T, error)

        w0 += rate \* np.sum(error)

        wi += rate \* dgradient

    predicted\_weights = np.append(w0, wi)

    return predicted\_weights

# function to create a list containing mini-batches

def MiniBatches(features, labels, batch\_size):

    mini\_batches = []

    featuresdf = pd.DataFrame(features)

    labelsdf = pd.DataFrame(labels)

    concat\_matrix = pd.concat([featuresdf, labelsdf], axis=1)

    data = concat\_matrix.to\_numpy()

    np.random.shuffle(data)

    n\_minibatches = data.shape[0] // batch\_size

    for i in range(n\_minibatches + 1):

        mini\_batch = data[i \* batch\_size:(i + 1)\*batch\_size, :]

        features\_mini = mini\_batch[:, :-1]

        labels\_mini = mini\_batch[:, -1].reshape((-1, 1))

        mini\_batches.append((features\_mini, labels\_mini))

    if data.shape[0] % batch\_size != 0:

        mini\_batch = data[i \* batch\_size:data.shape[0]]

        features\_mini = mini\_batch[:, :-1]

        labels\_mini = mini\_batch[:, -1].reshape((-1, 1))

        mini\_batches.append((features\_mini, labels\_mini))

    return mini\_batches

# function to perform mini-batch gradient descent

def MiniGradientDescent(features, labels, rate, batch\_size = 100):

    weights = np.random.normal(0, 0.01, len(features[0])+1)

    w0 = weights[0]

    wi = weights[1:len(weights)]

    iterations = 1000

    for iteration in range(iterations):

        mini\_batches = MiniBatches(features, labels, batch\_size)

        for mini\_batch in mini\_batches :

            features\_mini, labels\_mini = mini\_batch

            score = w0 + np.dot(features\_mini, wi)

            predictions = sigmoid(score)

            error = labels\_mini.flatten() - predictions

            dgradient = np.dot(features\_mini.T, error)

            w0 += rate \* np.sum(error)

            wi += rate \* dgradient

    predicted\_weights = np.append(w0, wi)

    return predicted\_weights

array\_train\_features[:, 28] = Normalize(array\_train\_features[:, 28])

log\_scale = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

trained\_weights = [0]\*len(log\_scale)

array\_test\_labels = concat\_test.to\_numpy()[:,0]

array\_test\_features = concat\_test.to\_numpy()[:,1:30]

array\_test\_features[:, 28] = Normalize(array\_test\_features[:, 28])

tic = [0]\*len(log\_scale)

toc = [0]\*len(log\_scale)

for r in range(len(log\_scale)):

    tic[r] = time.perf\_counter()

    trained\_weights[r] = BatchGradAscent(array\_train\_features, array\_train\_labels, log\_scale[r])

    toc[r] = time.perf\_counter()

    print('\nConfusion Matrix for learning rate = {} : '.format(log\_scale[r]))

    Performance(trained\_weights[r], array\_test\_features, array\_test\_labels)

    print()

print('\nIt seems that the learning rate = 0.001 gives the best results, as the accuracy and precision rates are as high as small rates, and\nis less computationally expensive than the small rates, and there is no decrease in precision like in the larger rates.')

print('\n=> Performance Metrics for Full Batch Gradient Ascent are:\n')

Performance(trained\_weights[2], array\_test\_features, array\_test\_labels, best\_one = True)

print(f"\nTotal time elapsed for full batch gradient ascent is {toc[2] - tic[2]:0.4f} seconds")

tic = time.perf\_counter()

minibatch\_weights = MiniGradientDescent(array\_train\_features, array\_train\_labels, log\_scale[2])

toc = time.perf\_counter()

print('\n => Performing the Mini Patch Gradient Ascent for Batch Size = 100, the following Performance Metrics are obtained: \n')

Performance(minibatch\_weights, array\_test\_features, array\_test\_labels, best\_one = True)

print(f"\nTotal time elapsed for mini batch gradient ascent is {toc - tic:0.4f} seconds")

#stochastic gradient ascent is basically mini batch with batch size of 1.

tic = time.perf\_counter()

stochastic\_weights = MiniGradientDescent(array\_train\_features, array\_train\_labels, log\_scale[2], batch\_size=1)

toc = time.perf\_counter()

print('\n=> Performing the Stochastic Gradient Ascent, the following Performance Metrics are obtained: \n')

Performance(stochastic\_weights, array\_test\_features, array\_test\_labels, best\_one = True)

print(f"\nTotal time elapsed for stochastic gradient ascent is {toc - tic:0.4f} seconds")