1. Logistic Regression

"""

This file contains the classes that implement Logistic Regression

"""

import numpy as np

import matplotlib.pyplot as plt

from preprocessing import NormalScaler

import pandas as pd

from mpl\_toolkits.mplot3d import Axes3D

**class** LogisticRegression:

    """

    This class implements Logistic Regression using both

    Batch Gradient Descent and Stochastic Gradient descent.

    Class attributes:

        W        : current set of weights

        W\_arr    : list of weights at each iteration

        Cost     : current cost

        cost\_arr : list of costs at each iteration

    """

**def** init\_weights(self, s):

        """

        This method initializes the weight matrix

        as a column vector with shape = (X rows+1, 1)

        """

        np.random.seed(11)

        self.W = np.random.randn(s[1],1)

        self.W\_arr = []

        self.cost\_arr = []

        self.cost = 0

        self.gradient = 0

**def** sigmoid(self,x):

        """

        Sigmoid activation function.

        returns the sigmoid of the given input.

        """

        return 1/(1+np.exp(-x))

**def** get\_cost(self, X, y, W):

        """

        This function returns the cost with the given set of weights

        using the formula

J

        """

        total\_cost = 0

        for i in range(X.shape[0]):

            total\_cost += y[i]\*np.log(self.get\_h\_i(X, i, W)) + (1-y[i])\*np.log(1-self.get\_h\_i(X, i, W))

        return (0.5/X.shape[0])\*total\_cost

**def** add\_bias(self, X):

        """

        This function adds bias (a column of ones) to the feature vector X.

        """

        bias = np.ones((X.shape[0],1))

        return np.concatenate((bias,X), axis=1)

**def** get\_h\_i(self, X, i, W):

        """

        This function returns the hypothesis of ith feature vector

        with the given weights W.

        """

        h\_i = np.matmul(X[i].reshape(1,-1),W)

        return self.sigmoid(h\_i[0][0])

**def** batch\_grad\_descent(self, X, y, alpha, max\_iter):

        """

        This function implements the Batch Gradient Descent algorithm.

        It runs for multiple iterations until either the weights converge or

        iterations reach max\_iter. At each iteration the weights are updated using

        the following rule

            repeat until convergence{

        """

        W\_new = self.W.copy()

        for \_ in range(max\_iter):

            grad = np.zeros((X.shape[0],1))

            for i in range(X.shape[0]):

                grad[i] = (-y[i]\*(1-self.get\_h\_i(X, i, self.W)) + (1-y[i])\*self.get\_h\_i(X, i, self.W))

            for j in range(X.shape[1]):

                W\_new[j][0] = self.W[j][0] - (alpha/X.shape[0])\*np.sum(grad\*X[:,j:j+1].reshape(-1,1))

            self.W = W\_new.copy()

            self.cost\_arr.append(self.get\_cost(X, y, self.W))

            self.W\_arr.append(self.W)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return W\_new

**def** stochastic\_grad\_descent(self, X, y, alpha, max\_iter):

        """

        This function implements the Stochastic Gradient Descent algorithm.

        It runs for multiple iterations until either the weights converge or

        iterations reach max\_iter. Weights are updated for every row of the

        training set.

            repeat until convergence{

                randomly shuffle the feature matrix rows

                for each feature vector x^i {

                    update all weights j -> 0 to n+1

                }

            }

        """

        mat = np.concatenate((X,y.reshape(-1,1)), axis=1)

        for \_ in range(max\_iter):

            W\_new = self.W.copy()

            np.random.shuffle(mat)

            X = mat[:,0:3]

            y = mat[:,3]

            for i in range(X.shape[0]):

                grad = (-y[i]\*(1-self.get\_h\_i(X, i, self.W)) + (1-y[i])\*self.get\_h\_i(X, i, self.W))

                for j in range(X.shape[1]):

                    W\_new[j][0] = self.W[j][0] - (alpha)\*(grad[0]\*X[i,j])

                self.W = W\_new.copy()

            self.cost\_arr.append(self.get\_cost(X, y, self.W))

            self.W\_arr.append(self.W)

            if len(self.W\_arr)>1:

                if sum(abs(self.W\_arr[-2]-self.W\_arr[-1]))<0.0001:

                    break

        return self.W

**def** train(self, X, y, alpha, max\_iter=100, option="batch"):

        """

        This function initiates the training process.

        It runs batch gradient descent by default and can also run

        Stochastic gradient descent if the argument is passed.

        returns the cost list which has costs at every training iteration.

        """

        X = self.add\_bias(X)

        self.init\_weights(X.shape)

        if option=="batch":

            self.batch\_grad\_descent(X,y,alpha,max\_iter)

        elif option=="stochastic":

            self.stochastic\_grad\_descent(X,y,alpha,max\_iter)

        self.cost = self.cost\_arr[len(self.cost\_arr)-1]

        return self.cost\_arr

**def** test(self,X,W=""):

        """

        This function takes a feature matrix as test data and

        predicts the target values using the trained weights.

        returns the predicted target values.

        """

        if W=="":W = self.W

        X = self.add\_bias(X)

        y\_pred = np.ones(X.shape[0])

        for i in range(X.shape[0]):

            y\_pred[i] = self.get\_h\_i(X,i,W)

            for j in range(X.shape[1]):

                y\_pred[i] += X[i][j]\*W[j][0]

            y\_pred[i] = self.sigmoid(y\_pred[i])

        return y\_pred

if \_\_name\_\_ == "\_\_main\_\_":

    model = LogisticRegression()

*# data input*

    data = pd.read\_excel("./data3.xlsx",header=None)

    data = data.sample(frac=1).reset\_index(drop=True)

    X = data[[0,1,2,3]]

    y = data[4]-1

*# data preprocessing (Normal scaling)*

    mscaler = NormalScaler()

    for j in range(X.shape[1]):

        mscaler.fit(X.loc[:,j])

        X.loc[:,j] = mscaler.transform(X.loc[:,j])

*# holdout cross validation split*

    train\_percent = 0.6

    X\_train = X[:int(train\_percent\*X.shape[0])]

    y\_train = y[:int(train\_percent\*X.shape[0])]

    X\_test = X[int(train\_percent\*X.shape[0]):]

    y\_test = y[int(train\_percent\*X.shape[0]):]

*# Training the model by choosing alpha and max\_iter values.*

*# gradient descent algorithm can be set as either ‘batch’ or ‘stochastic’*

*# in this function call.*

    alpha = 0.26

    max\_iter = 100

    algo = 'batch'

    model.train(X\_train.values,y\_train.values,alpha,max\_iter,algo)

*# Testing on train set*

    print("\nTraining..")

    y\_pred = model.test(X\_train.values)

    for i in range(y\_pred.shape[0]):

        y\_pred[i] = 0 if y\_pred[i]<0.5 else 1

    print('\n',y\_pred)

    print("\nTraining set accuracy: ",sum(y\_pred==y\_train)/y\_train.shape[0])

    print("Training set sensitivity: ",sum((y\_pred==1) & (y\_train==1))/sum(y\_train==1))

    print("Training set specificity: ",sum((y\_pred==0) & (y\_train==0))/sum(y\_train==0))

*# Testing on test set*

    print("\nTesting...")

    y\_pred = model.test(X\_test.values)

    for i in range(y\_pred.shape[0]):

        y\_pred[i] = 0 if y\_pred[i]<0.5 else 1

    print('\n',y\_pred)

    print("\nTesting set accuracy: ",sum(y\_pred==y\_test)/y\_test.shape[0])

    print("Training set sensitivity: ",sum(y\_pred\*y\_test)/sum(y\_test))

    print("Training set specificity: ",sum((y\_pred==0) & (y\_test==0))/sum(y\_test==0))

1. One vs All

from LogisticRegression import LogisticRegression,NormalScaler

if \_\_name\_\_ == "\_\_main\_\_":

*# data input*

    data = pd.read\_excel("./data4.xlsx",header=None)

    data = data.sample(frac=1).reset\_index(drop=True)

    X = data[[i for i in range(7)]]

    y = data[7]

    unique\_classes = np.unique(y)

    num\_classes = len(unique\_classes)

*# data preprocessing*

    mscaler = NormalScaler()

    for j in range(X.shape[1]):

        mscaler.fit(X[j])

        X[j] = mscaler.transform(X[j])

    y\_cat = (y==unique\_classes[0]).astype('int').values.reshape(-1,1)

    for i in unique\_classes[1:]:

        y\_cat = np.concatenate((y\_cat,(y==i).astype('int').values.reshape(-1,1)),axis=1)

*# splitting data using holdout cross validation*

    train\_percent = 0.6

    X\_train = X[:int(train\_percent\*X.shape[0])]

    y\_train = y[:int(train\_percent\*X.shape[0])]

    y\_cat\_train = y\_cat[:int(train\_percent\*X.shape[0])]

    X\_test = X[int(train\_percent\*X.shape[0]):]

    y\_test = y[int(train\_percent\*X.shape[0]):]

    y\_cat\_test = y\_cat[int(train\_percent\*X.shape[0]):]

*# creating a Logistic regression model for each class*

    models = [LogisticRegression() for i in unique\_classes]

    y\_train\_pred = np.ndarray((y\_train.shape[0],num\_classes))

    y\_test\_pred = np.ndarray((y\_test.shape[0],num\_classes))

    for c in range(num\_classes):

*# training*

        models[c].train(X\_train,y\_cat\_train[:,c],0.26,100,'batch')

        y\_train\_pred[:,c] = models[c].test(X\_train)

*# testing*

        y\_test\_pred[:,c] = models[c].test(X\_test)

        y\_p = (y\_test\_pred[:,c]>0.5)

        print("Class ",unique\_classes[c]," Accuracy = ", sum(y\_p==(y\_test==unique\_classes[c]))/(X\_test.shape[0]))

    y\_train\_t = np.argmax(y\_train\_pred, axis=1)+1

    y\_test\_t = np.argmax(y\_test\_pred, axis=1)+1

    print("Train Accuracy : ",sum(y\_train\_t==y\_train)/y\_train.shape[0])

    print("Test Accuracy : ",sum(y\_test\_t==y\_test)/y\_test.shape[0])

*# Confusion Matrix*

    conf\_mat = np.ndarray((num\_classes, num\_classes))

    for i in range(num\_classes):

        for j in range(num\_classes):

            conf\_mat[i][j] = sum((y\_test\_t==unique\_classes[i]) & (y\_test==unique\_classes[j]))

    print(conf\_mat)

1. One vs One

from LogisticRegression import LogisticRegression,NormalScaler

if \_\_name\_\_ == "\_\_main\_\_":

    model = LogisticRegression()

*# data input*

    data = pd.read\_excel("./data4.xlsx",header=None)

    data = data.sample(frac=1).reset\_index(drop=True)

    X = data[[i for i in range(7)]]

    y = data[7]

*# data preprocessing*

    mscaler = NormalScaler()

    for j in range(X.shape[1]):

        mscaler.fit(X.loc[:,j])

        X.loc[:,j] = mscaler.transform(X.loc[:,j])

    unique\_classes = np.unique(y)

    num\_classes = len(unique\_classes)

    num\_models = (int)(num\_classes\*(num\_classes-1)/2)

*# splitting data using holdout cross validation*

    train\_percent = 0.6

    X\_train = X[:int(train\_percent\*X.shape[0])]

    y\_train = y[:int(train\_percent\*X.shape[0])]

    X\_test = X[int(train\_percent\*X.shape[0]):]

    y\_test = y[int(train\_percent\*X.shape[0]):]

    models = [[0 for j in range(num\_classes)] for i in range(num\_classes)]

    y\_test\_pred = np.ndarray((y\_test.shape[0], num\_models))

    k = 0

*# training and testing n(n-1)/2 models*

    for i in range(num\_classes-1):

        for j in range(i+1, num\_classes):

            class\_i = unique\_classes[i]

            class\_j = unique\_classes[j]

            models[i][j] = LogisticRegression()

            tmp = (y\_train==class\_i) | (y\_train==class\_j)

            y\_train\_i\_j = (y\_train[tmp]==class\_i).astype('int').values

            models[i][j].train(X\_train[tmp], y\_train\_i\_j, 0.1, 100, 'batch')

            y\_test\_pred[:,k] = models[i][j].test(X\_test)

            y\_test\_pred[:,k][y\_test\_pred[:,k]>=0.5] = class\_i

            y\_test\_pred[:,k][y\_test\_pred[:,k]<0.5] = class\_j

            acc = sum(y\_test\_pred[:,k]==y\_test)/y\_test.shape[0]

            print("{0} vs {1} Accuracy: {2}".format(i+1,j+1,acc))

            k+=1

*# calculating overall accuracy*

    y\_test\_t = np.ndarray((y\_test.shape[0],))

    for i in range(y\_test.shape[0]):

        uniqu,counts = np.unique(y\_test\_pred[i],return\_counts=True)

        y\_test\_t[i] = uniqu[np.argmax(counts)]

    print("\nOverall Accuracy: ", sum(y\_test\_t==y\_test)/y\_test.shape[0])

1. Kfold

from LogisticRegression import LogisticRegression,NormalScaler

**def** predictOneVsAll(X\_train, y\_train, X\_test, y\_test, unique\_classes):

    num\_classes = len(unique\_classes)

    models = [LogisticRegression() for i in unique\_classes]

    y\_train\_pred = np.ndarray((y\_train.shape[0],num\_classes))

    y\_test\_pred = np.ndarray((y\_test.shape[0],num\_classes))

    for c in range(num\_classes):

        models[c].train(X\_train,y\_cat\_train[:,c],0.26,100,'batch')

        y\_train\_pred[:,c] = models[c].test(X\_train)

        y\_test\_pred[:,c] = models[c].test(X\_test)

        y\_p = (y\_test\_pred[:,c]>0.5)

        print("Class ",unique\_classes[c]," Accuracy = ", sum(y\_p==(y\_test==unique\_classes[c]))/(X\_test.shape[0]))

    y\_train\_t = np.argmax(y\_train\_pred, axis=1)+1

    y\_test\_t = np.argmax(y\_test\_pred, axis=1)+1

    test\_acc = sum(y\_test\_t==y\_test)/y\_test.shape[0]

    print("Train Accuracy : ",sum(y\_train\_t==y\_train)/y\_train.shape[0])

    print("Test Accuracy : \n",test\_acc)

*# Confusion Matrix*

    conf\_mat = np.ndarray((num\_classes, num\_classes))

    for i in range(num\_classes):

        for j in range(num\_classes):

            conf\_mat[i][j] = sum((y\_test\_t==unique\_classes[i]) & (y\_test==unique\_classes[j]))

    print(conf\_mat,"\n")

    return test\_acc

if \_\_name\_\_ == "\_\_main\_\_":

*# data input*

    data = pd.read\_excel("./data4.xlsx",header=None)

    data = data.sample(frac=1).reset\_index(drop=True)

    X = data[[i for i in range(7)]]

    y = data[7]

    unique\_classes = np.unique(y)

    num\_classes = len(unique\_classes)

*# data preprocessing*

    mscaler = NormalScaler()

    for j in range(X.shape[1]):

        mscaler.fit(X[j])

        X[j] = mscaler.transform(X[j])

    y\_cat = (y==unique\_classes[0]).astype('int').values.reshape(-1,1)

    for i in unique\_classes[1:]:

        y\_cat = np.concatenate((y\_cat,(y==i).astype('int').values.reshape(-1,1)),axis=1)

    k = 5

    N = X.shape[0]

    j = 0

    acc = 0

*# splitting data using k fold cross validation approach*

    for i in range(0,k):

        X\_train = np.concatenate((X[:i\*(N//k)],X[(i+1)\*(N//k):]))

        y\_train = np.concatenate((y[:i\*(N//k)],y[(i+1)\*(N//k):]))

        y\_cat\_train = np.concatenate((y\_cat[:i\*(N//k)],y\_cat[(i+1)\*(N//k):]))

        X\_test = X[i\*(N//k):(i+1)\*(N//k)]

        y\_test = y[i\*(N//k):(i+1)\*(N//k)]

        y\_cat\_test = y\_cat[i\*(N//k):(i+1)\*(N//k)]

        acc += predictOneVsAll(X\_train, y\_train, X\_test, y\_test, unique\_classes)

    print("Average Accuracy: \n", acc/k)