```python
# -*- coding: utf-8 -*-
"""Logistic_Regression

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1qioJbplkgpPKdiDEP2SYKmu6t7V-Maoo

<center><h1>Mini Project 1 - Logistic Regression</h1>
<h4>The hyperparameters and models used in this file are chosen based on the findings in the testing file.</h4></center>

<h3>Team Members:</h3>
<center>
Yi Zhu, 260716006<br>
Fei Peng, 260712440<br>
Yukai Zhang, 260710915
</center>
"""

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

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

path1 = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/hepatitis.csv"
path2 = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/bankrupcy.csv"

class LogisticRegression:
    '''
        This is the logistic regression class, containing fit, perdict and accu_eval functions,
        as well as many other useful functions.
    '''

    def __init__(self, data, folds, lr=0.01, max_iter=10000, beta=0.99, epsilon=5e-3):
        self.data = data
        self.folds = folds
        self.lr = lr
        self.max_iter = max_iter
        self.beta = beta
        self.epsilon = epsilon

    def shuffle_data(self):
        '''
            This function randomly shuffles the input dataset.
        '''
        # Load data from data file.
        self.data.insert(0, column='Bias', value=1)
        self.data = self.data.sample(frac=1)

    def partition(self, fold):
        '''
            This function divides the dataset into training and validation set.

            fold - the current fold
        '''
        data = self.data
        # to exclude last term in previous partition for training data
```

```
        train_add = 1 if fold < self.folds else 0
        # to exclude last term in previous partition for testing data
        test_add = 1 if fold > 0 else 0

        # number of data sets
        n = len(self.data)

        train_set_1 = data.iloc[0:int((fold)/self.folds*n), :]
        train_set_2 = data.iloc[int((fold+1)/self.folds*n)+train_add:n, :]
        train_set = pd.concat([train_set_1, train_set_2])

        test_set = data.iloc[int((fold)/self.folds*n+test_add):int((fold+1)/self.folds*n), :]

        train_X = train_set.iloc[:, :-1].values
        train_y = train_set.iloc[:, -1].values
        train_y = np.reshape(train_y, (-1,1))

        test_X = test_set.iloc[:, :-1].values
        test_y = test_set.iloc[:, -1].values
        test_y = np.reshape(test_y, (-1,1))

        return train_X, train_y, test_X, test_y

    def normalization(self, X, v_X):
        '''
            This function performs the z-score normalization

            X - training data
            v_X - validation data
        '''
        mean = np.mean(X[:,1:], axis = 0)
        sigma = np.std(X[:,1:], axis = 0)
        mean = np.reshape(mean, (1,-1))
        sigma = np.reshape(sigma, (1,-1))
        X[:,1:] = (X[:,1:] - mean) / sigma
        v_X[:,1:] = (v_X[:,1:] - mean) / sigma
        return X, v_X

    def fit(self, X, y, v_X, v_y, normalize=False):
        '''
            This function takes the training data X and its corresponding labels vector y
            as well as other hyperparameters (such as learning rate) as input,
            and execute the model training through modifying the model parameters (i.e. W).

            X - training data
            y - class of training data
            v_X - validation data
            v_y - class of validation data
            epsilon - the threshold value for gradient descent
            normalize - whether to perform normalization
        '''
        gradient_values, t_acc_val, v_acc_val = [], [], []

        if normalize:
            X, v_X = self.normalization(X, v_X)

        # Retrive the learning rate, maximum iteration, momentum (beta)
        lr, max_iter, beta, epsilon = self.lr, self.max_iter, self.beta, self.epsilon

```

```
        # initial weight vector
        w = np.zeros((len(X[0]), 1))
        # record the best weight vector
        best_w = w
        # iteration number, validation accuracy, last validation accuracy
        # the step to take in gradient descent, maximum validation accuracy
        iteration, v_acc, step, v_acc_max = 0, 0, 0, 0

        dw = np.inf
        # if the gradient delta w is smaller than threshold or achieved maximum iteration, stop
        while (np.linalg.norm(dw) > epsilon and iteration <= max_iter):
            dw = self.gradient(X, y, w)
            gradient_values.append(np.linalg.norm(dw))
            # if beta = 0, it will be the same as general gradient descent
            step = beta * step + (1 - beta) * dw  # gradient descent with momentum
            w = w - lr * step

            # predict once every 10 interations
            if iteration % 10 == 0:
                t_y_pred = self.predict(X, w)
                t_acc = self.accu_eval(t_y_pred, y)
                v_y_pred = self.predict(v_X, w)
                v_acc = self.accu_eval(v_y_pred, v_y)

            # record the next best value
            if v_acc >= v_acc_max:
                v_acc_max = v_acc
                best_w = w
                self.marker = iteration  # move the iteration marker

            t_acc_val.append(t_acc)
            v_acc_val.append(v_acc)

            iteration = iteration + 1

        return gradient_values, t_acc_val, v_acc_val, best_w

    def predict(self, X, w):
        '''
            This function takes a set of data as input and outputs predicted labels for the input points.
        '''
        result = self.log_func(np.dot(X, w))
        # the prediction result converted to binary
        predict_bin = []
        for i in result:
            if i>=0.5:
                predict_bin.append(1)
            else:
                predict_bin.append(0)
        return predict_bin

    def accu_eval(self, y_pred, y):
        '''
            This function evaluates the models' accuracy.
        '''
        count = 0
        for i in range(len(y_pred)):
            if y_pred[i] == y[i]:
                count = count + 1
        # return the accuracy ratio: #corret prediction / #data points
```

```python
        return count / len(y)

    def log_func(self, alpha):
        return 1 / (1 + np.exp(-alpha))

    def gradient(self, X, y, w):
        N = len(X[0])
        y_hat = self.log_func(np.dot(X, w))
        delta = np.dot(X.T, y_hat - y) / N
        return delta

class KFoldValidation:
    def __init__(self, folds, path, lr, max_iter, epsilon, beta):
        self.folds = folds
        self.data = pd.read_csv(path)
        self.lr = lr
        self.max_iter = max_iter
        self.epsilon = epsilon
        self.beta = beta

    def k_fold_validation(self, normalize=False, inc_od=False, order=3):
        '''
            This function performs the k-fold validation

            normalize - whether to perform normalization
            inc_od - whether to increase the feature order
            order - the order of the added feature
        '''
        folds = self.folds
        data = self.data
        accuracies = []

        if inc_od:
            data = self.rise_order(data, order)

        log_reg = LogisticRegression(data=data, folds=self.folds, lr=self.lr, max_iter=self.max_iter, beta=self.beta, epsilon=self.epsilon)

        log_reg.shuffle_data()

        for fold in range(folds):
            t_X, t_y, v_X, v_y = log_reg.partition(fold)
            # t_X --> test value X, v_X --> validation value X
            gradient_val, t_acc_val, v_acc_val, best_w = log_reg.fit(t_X, t_y, v_X, v_y, normalize=normalize)

            accuracies.append(np.max(v_acc_val))

            # Uncommant this block to display the accuracy diagram
            plt.figure()
            plt.plot(t_acc_val, label = 'Training accuracy')
            plt.plot(v_acc_val, label='Validation accuracy')
            plt.axvline(log_reg.marker, color='r', label='Best Weights')
            plt.xlabel('Iteration Number')
            plt.ylabel('Accuracy')
            plt.legend()
            plt.show()
            print("Learning Rate: " + str(log_reg.lr))
            print("Average Accuracy: "+str(np.mean(accuracies)))

            # Uncommant this block to display the gradiant diagram
            plt.figure()
            plt.plot(gradient_val)
```

```python
            plt.xlabel('Iteration Number')
            plt.ylabel('Gradiant')
            plt.show()
            print("-------------------------------------------------------")

        mean_acc = np.mean(accuracies)
        return mean_acc

    def rise_order(self, data, order=3):
        ret_val = data
        for i in range(2, order + 1):
            data_powered = data.pow(i)
            ret_val = ret_val.iloc[:, :-1]
            ret_val = pd.concat([ret_val, data_powered],axis=1)
        return ret_val

"""### Perform 10-fold validation for Hepatitis dataset"""

path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/hepatitis.csv"
dataset_name = "Hepatitis"
defult_lr = 0.01
default_max_iter = 10000
default_epsilon = 5e-3
defulat_beta = 0.99

# the input is the optimum hyperparameters found during testing
hepatitis_learning = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
# the input is the optimum model found during testing
mean_acc = hepatitis_learning.k_fold_validation()

"""### Perform 10-fold validation for Bankruptcy dataset"""

path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/bankrupcy.csv"
dataset_name = "Bankruptcy"
defult_lr = 0.1
default_max_iter = 25000
default_epsilon = 1e-3
defulat_beta = 0.99

# the input is the optimum hyperparameters found during testing
bankruptcy_learning = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
# the input is the optimum model found during testing
mean_acc = bankruptcy_learning.k_fold_validation(normalize=True, inc_od=True)
```