ECSE 551 Machine Learning for Engineers
Mini-project 1 Report

Fei Peng
260712440
fei.peng@mail.mcgill.ca

Yukai Zhang
260710915
yukai.zhang@mail.mcgill.ca

Yi Zhu
260716006
yi.zhu6@mail.mcgill.ca

Abstract

Logistic regression is one of the most popular linear classification techniques in machine learning and it has been widely used in binary classification and predictions. In this mini-project, we investigated and evaluated the performance of logistic regression by implementing it using two benchmark datasets: *Hepatitis* and *Bankruptcy*. Both two datasets were pre-processed to explore their characteristics, thereby achieving the best outcome in the following training. Fitting, predicting, and testing were then performed to train and compare different models. The accuracy evaluations were analysed, and detailed results will be presented in the following parts of this report. Among them, we found that the accuracy of the logistic regression classifier could be improved by normalizing the datasets, changing order of the model, or removing certain features of the numerical datasets.

1 Introduction

In recent years, machine learning has been adapted to many different fields, including medical research and financial analysis [1]. As one of the most feasible and efficient classification methods, logistic regression is widely used to predict binary outcomes. The main objective of this project is to implement a logistic regression classifier and investigate its accuracy using k-fold cross validation, as well as analysis the impact of data preprocessing on the validation accuracy. We took the probabilistic views to process the binary classification. In probabilistic approaches, we focused on discriminative learning which directly estimates the probability of y (i.e., output class/label) given x (i.e., input data). Using Bayes' Rule, the logistic function (also known as sigmoid function) could be obtained, which takes a linear function of x as independent variable $w^T x$, and produces the conditional probability $P(y|x)$.

The most critical task of implementing the logistic regression algorithm was to find the linear term w (i.e., weights) that produces the minimum error/maximum accuracy in prediction while ensuring the efficiency of this algorithm. This was achieved using gradient descent with momentum [2], and tweaking the hyperparameters (e.g., learning rate, momentum term, and stopping criteria) of the fit function. It was found that, with a carefully chosen set of hyperparameters, accuracy could be increased while remaining satisfying training efficiency.

During data pre-processing, normalization was performed and proved to increase the accuracy. While analysing the characteristics of the datasets, it was noticeable that some features of input x satisfies the null hypothesis [3], which means that these features are not helping discriminate between two classes and should be removed to improve performance. Moreover, through the distribution of the two classes of the two datasets, it was discovered that the entropy of classes could affect the accuracy of the model, resulting in the model accuracy for hepatitis data (with higher entropy) being higher

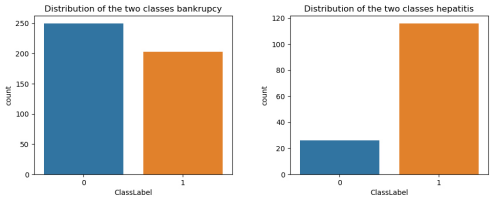

Figure 1: *Bankruptcy* data class distribution (left) and *Hepatitis* data class distribution (right)

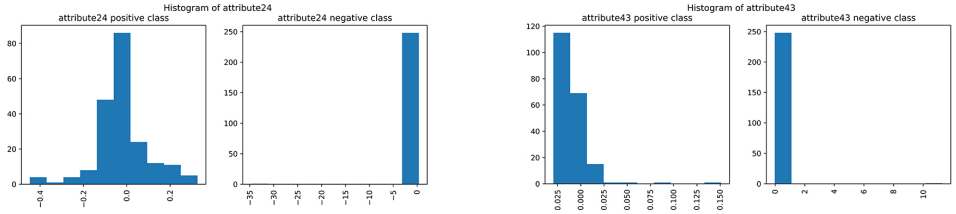

Figure 2: Attribute 24 (left) and attribute 43 (right) feature distribution for *Bankruptcy* data

than that of *bankruptcy* data. Last but not least, properly increasing the order of the model (order 3 for *bankruptcy* data) would also help increase accuracy. These aspects will be further discussed.

2 Datasets

Data preprocessing usually lays the groundwork for the later data analysis and helps it yield a higher accuracy efficiently. In this project, the following data preprocessing had been performed.

2.1 Entropy Analysis

Both datasets *Bankruptcy* and *Hepatitis* are examples of binary classification. The distributions of each class are shown in Figure 1. The uncertainty in prediction can be quantified by the entropy of each dataset with equation 1.

$$H(X) = -\sum_{x \epsilon X} p(x) log_2 p(x) \tag{1}$$

2.2 Features Analysis

For the purpose of gaining a better understanding toward the dataset, the distributions of each feature in both classes are plotted. Some of the plots are shown in Figure 2.

Note that both pairs of plots come from *Bankruptcy* dataset and in each pair, the distribution of positive class is shown on the left, otherwise right. When comparing positive and negative classes, it is noticeable that the data distribution differs greatly in attribute 24 but slightly in attribute 43. This indicates that features with similar distribution to attribute 43 will have little contribution in the prediction process and should be considered as null distribution. Kolmogorov-Smirnov method [4] was performed to efficiently find all null distributions. Attributes 43 and 60 in the *Bankruptcy* dataset were identified. The removal of these two features may improve the performance of the machine learning process. The Kolmogorov-Smirnov method, however, failed to yield a promising result in the *Hepatitis* dataset because of the existence of multiple binary features. The effect of removing features with null distribution will be further discussed in the result section.

2.3 Data Shuffling

It was found that the *Bankruptcy* dataset is ordered by class label. Proceeding without shuffling the data would result in some validation/training sets all filled by data from one class. Thus, rows will be shuffled every time after importation. Doing so will also help reduce the variance and generalize the algorithm, therefore the *Hepatitis* dataset should also be shuffled.

2.4 Data Normalization

Normalizing the data into z-score can centralize the value distribution of the data while keeping its precision [5]. When a calculation involves large float numbers, they can easily trigger overflow/underflow and lose precision without normalization. Calculating the gradient with high order data is an example of that. Luckily, using z-score should help prevent introducing such errors and thus increase the model accuracy.

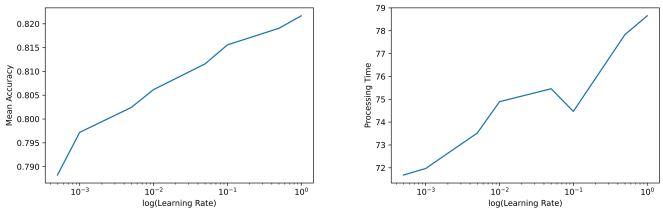

Figure 3: Mean accuracy (left) and processing time (right) vs. logarithm of learning rate

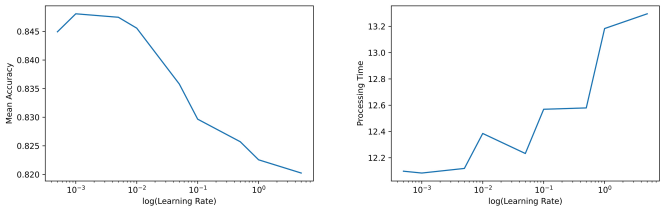

Figure 4: Mean accuracy (left) and processing time (right) vs. logarithm of learning rate

3 Results

Multiple experiments have been conducted to evaluate the performance of the logistic regression with respect to different hyperparameters and data preprocessing techniques. The detailed results are shown below.

3.1 Learning Rate

Learning rate is a parameter that determines the step size on the way approaching a (local) minimum. A large step size can accelerate the convergence but can also result in jumping over the desired solution (oscillate forever). A small step size can avoid skipping the solution but is very time consuming. To find a learning rate best fit for each model, values between 0.0005 and 5 are tested. The performance/processing time vs. logarithm of learning rate are shown in Figure 3 and Figure 4.

3.2 Stopping Criteria:

Theoretically, if a model suits the data distribution, the training accuracy will keep increasing with more iteration calculated. However, failing to set up a proper stopping criteria will not only waste the computational power but also lead to overfitting [6]. In this case, max iteration and epsilon(minimum gradient) are introduced as the upper limit of the iterations. In this experiment, the max iteration term is sampled from 500 to 25000 and the epsilon term is sampled from $1e^{-3}$ to 0.5. The plots of accuracy/processing time versus max iteration/epsilon are shown in Figure 5 and Figure 6.

Note that all the data displayed come from the *Bankruptcy* dataset. The *Hepatitis* dataset has a similar trend and will not be displayed. The plots confirm that in a reasonable iteration range, accuracy tends to rise as the maximum iteration increases and the minimum acceptable gradient decreases. For the trade-off between accuracy and efficiency, 25000 iterations with an epsilon of $1e^{-3}$ is selected for the *Bankruptcy* dataset and $(10000, 5e^{-3})$ is selected for the *Hepatitis* dataset.

3.3 Momentum Gradient Descent Constant - Beta

When a regular gradient descent method is performed, it is very unlikely that the algorithm can move directly from the starting point to the (local) minimum. Instead, the path usually oscillates back

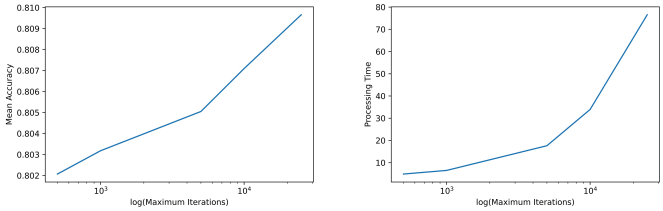

Figure 5: Mean accuracy (left) and Processing time (right) with respect to log of Max. Iteration

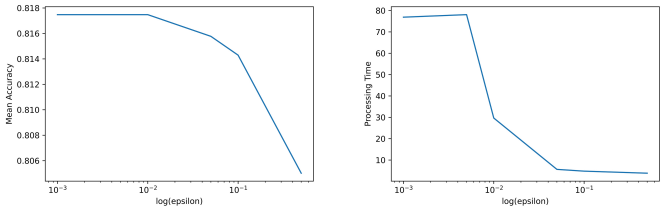

Figure 6: Mean accuracy (left) and processing time (right) with respect to log of epsilon

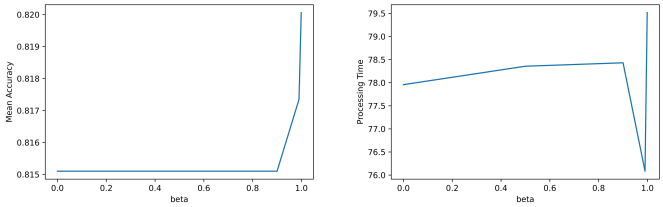

Figure 7: Mean accuracy (left) and processing time (right) with respect to beta

and forth around that optimum shortcut. As described in part 3.1, this oscillation will have a larger amplitude if the learning rate is high. A momentum gradient descent method can be used to flatten this oscillation and take a rather straightforward approach in the search of the minimum point. It does so by calculating the weighted sum of the current gradient and the previous step size and then using it as the new step size. β value is in charge of controlling the weight of the previous step size, if β is set to 0, it becomes equivalent to a regular gradient descent method [2]. In this experiment, value β is tested in the range of 0 to 0.999 and its results are shown in Figure 7.

The data displayed all come from *Bankruptcy* dataset. The trends of that in the *Hepatitis* dataset is also very similar to those above. According to the figures, a β of 0.99 yields the best overall performance regarding the accurateness and efficiency.

3.4 Normalization

During data preprocessing, it was discovered that most of the features in *Bankruptcy* data are floating points. Therefore, calculation may result in precision loss due to overflow/underflow. To improve model accuracy, data should be normalized before performing training or validation. The normalization technique used in this project is called z-score [5] which calculates the mean (j) and standard deviation (μ_j) for each feature (σ_j) of the training data, and perform $\frac{x_j - \mu_j}{\sigma_j}$ for both the training and validation data. Therefore the model accuracy could be improved by bringing more precision to the calculation of gradient. During the testing, it was found that the overflow warning indeed disappeared after normalization. It is shown in Figure 8 on the left that compared to that of without normalization, the validation accuracy of the normalized original dataset is higher. This advantage is even pronounced when more features (higher orders) are introduced, which will be discussed later.

3.5 Removing Features

As discussed previously in datasets, null distributions in the *Bankruptcy* dataset were found using Kolmogorov-Smirnov method [4]. This means that the distribution of some attributes in two classes

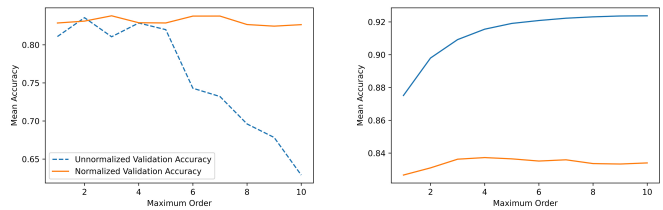

Figure 8: Comparison between with and without normalization (left) and the training and validation accuracy vs. order of *Bankruptcy* data (right)

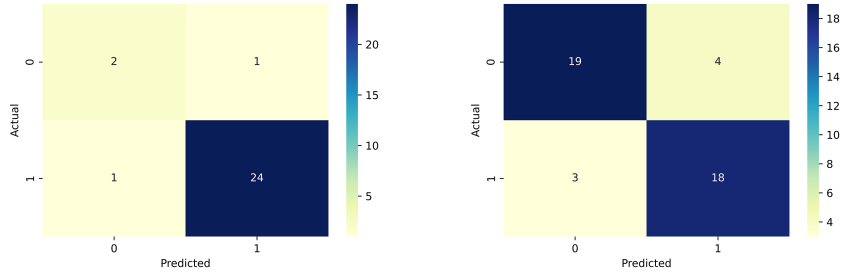

Figure 9: The Confusion Matrix. for *Hepatitis* data (left) and *Bankruptcy* data (right)

are similar and may not be useful for training and class prediction. Attribute 43 and 60 of *Bankruptcy* dataset were identified to be unhelpful, therefore were removed during training and validation. However, it came out that the mean validation accuracy and training efficiency did not vary significantly from before. One of the assumptions that could explain this result is that the linear term, w, of the log-odds ratio (i.e., the independent variable, $w^T x$, of the logistic function) as already taken care of the weight of each attribute, therefore removing any attribute would not have much impact on the accuracy.

3.6 Adding features

The approach of adding more features to the dataset was also adopted to improve model accuracy. The order of each attribute in the *Bankruptcy* dataset was increased and added to the original dataset. As shown in Figure 8 on the right, the blue curve indicates the training accuracy, while the orange curve represents the validation accuracy. It clearly shows that when the order is increased to around 3, the maximum validation accuracy is achieved. However, continuing increasing the order could cause overfitting [6], as the training accuracy remains increasing but the validation accuracy starts to decrease. Therefore, order 3 was chosen to be the most suitable model for training *Bankruptcy* dataset. This approach, however, was tested to be ineffective for *Hepatitis* dataset due to the fact that most of its features are binary.

4 Discussion and Conclusion

In conclusion, the logistic regression classifier implemented in this project performed as expected overall on the two datasets, *Bankruptcy* and *Hepatitis*. Here are the two representative confusion matrices, shown in Figure 9, calculated during testing to measure the accuracy, precision and sensitivity of the models. The selected model used for calculating the confusion matrix was the one that performed the best during testing (as explained in the previous section). The accuracy for the best *Hepatitis* model could achieve 92.86%, with 96% specificity. However the number of instances in this dataset is too small (only 142 total) to take any common measurements, therefore 5-fold validation was performed to obtain this confusion matrix. Moreover, this is one of the best case scenarios, thus in the future, more testing should be performed, ideally with more instances, to obtain a convincing average accuracy. For the best *Bankruptcy* model, the accuracy reached 84.09%, while the precision was 82.61% and the sensitivity was 86.36%. These measurements confirmed the previous hypothesis, that lower entropy can yield a higher accuracy.

Possible directions for future investigation could include choosing other types of model (e.g., with log, exponential, or interaction terms), exploring more efficient but complex gradient descent algorithms, and comparing logistic regression with genetic learning as *Hepatitis* is a relatively small dataset. These aspects should help not only improve model accuracy, but also accelerate the training process.

5 Statement of Contributions

The workload of this project is distributed equally between the team members (i.e., the three authors of this report). Yi Zhu implemented the fit function for training datasets, and explored various gradient descent algorithms to improve the computation speed. Fei Peng was responsible for developing the k-fold partition, predict and accuracy evaluation functions for predicting the classes/labels of the input data and evaluating the model accuracy, as well as performing thorough testing of the models. Yukai was in charge of data pre-processing and model selection, which helped to significantly improve the model accuracy.

References

[1] T. Peña, S. Martínez, and B. Abudu, "Bankruptcy Prediction: A Comparison of Some Statistical and Machine Learning Techniques," in *Computational Methods in Economic Dynamics,* H. Dawid and W. Semmler Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 109-131.

[2] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747,* 2016.

[3] R. W. Vitral, M. J. Campos, and M. R. Fraga, "The null hypothesis," *American journal of orthodontics and dentofacial orthopedics : official publication of the American Association of Orthodontists, its constituent societies, and the American Board of Orthodontics,* vol. 144, no. 4, pp. 498-9, 2013, doi: 10.1016/j.ajodo.2013.08.010.

[4] T. Liu, "A Kolmogorov-Smirnov type test for two inter-dependent random variables," *arXiv preprint arXiv:1802.09899,* 2018.

[5] A. E. Curtis, T. A. Smith, B. A. Ziganshin, and J. A. Elefteriades, "The Mystery of the Z-Score," *(in eng), Aorta (Stamford),* vol. 4, no. 4, pp. 124-130, 2016, doi: 10.12945/j.aorta.2016.16.014.

[6] AH. Allamy, "METHODS TO AVOID OVER-FITTING AND UNDER-FITTING IN SUPERVISED MACHINE LEARNING (COMPARATIVE STUDY)," 12/27 2014.

Appendix

(Please see following pages for code implementations attached)

File - E:\Study\ECSE551\Mini_Project_1\code\data_preprocessing.py

```
# -*- coding: utf-8 -*-
"""Data_Preprocessing

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1cqiOtupPBH0ZZWkk2pkLA-k84lVnQgSB

<center><h1>Mini Project 1 - Logistic Regression</h1>
<h4>This file is for data preprocessing. Important characteristics of features and the distribution of classes could be found in the result of this 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 seaborn as sns
import matplotlib.pyplot as plt
import random
from scipy import stats
from google.colab import files

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"

hepatitis_data = pd.read_csv(path1)
bankrupcy_data = pd.read_csv(path2)

class Data_Processing:
    def __init__(self, data, name = 'New Data'):
        self.data = data
        self.name = name

    def partition_by_class(self):
        pos, neg = [], []
        data = self.data
        for row in range(1, data.shape[0]):
            if data[row, -1] == 1:
                pos.append(list(data[row]))
            else:
                neg.append(list(data[row]))
        self.pos = pos
        self.neg = neg

    def show_y_dist(self, ydata):
        plt.figure(figsize=(5,4))
        plt.subplot(111), sns.countplot(x='ClassLabel', data=ydata)
        plt.title('Distribution of the two classes {}'.format(self.name))
        plt.savefig("Distribution of the two classes {}.png".format(self.name), dpi = 1200)
        files.download("Distribution of the two classes {}.png".format(self.name))
        plt.show()
```

```
 60
 61     def show_x_dist(self, xdata):
 62         pos = self.pos
 63         neg = self.neg
 64         for i in range(0,len(pos[1])):
 65             fig, (ax1, ax2) = plt.subplots(1, 2)
 66             fig.set_size_inches(10,4)
 67             plt.setp(ax1.xaxis.get_majorticklabels(), rotation=90)
 68             plt.setp(ax2.xaxis.get_majorticklabels(), rotation=90)
 69             fig.suptitle('Histogram of {}'.format(xdata.keys()[i]))
 70             ax1.title.set_text('{} positive class'.format(xdata.keys()[i]))
 71             ax1.hist([pos[j][i] for j in range(len(pos))])
 72             ax2.title.set_text('{} negative class'.format(xdata.keys()[i]))
 73             ax2.hist([neg[j][i] for j in range(len(neg))])
 74
 75     def find_null_data(self):
 76         data, pos, neg = self.data, self.pos, self.neg
 77         num_data = min(len(pos), len(neg))
 78         num_feature = len(pos[0])
 79         null_feature_count = np.zeros(num_feature)
 80         pos = random.sample(pos, k = num_data)
 81         neg = random.sample(neg, k = num_data)
 82         for i in range(len(pos[1])):
 83             posi_list = []
 84             nega_list = []
 85             for j in range(len(pos)):
 86                 posi_list.append(pos[j][i])
 87                 nega_list.append(neg[j][i])
 88             a, b = stats.ks_2samp(posi_list, nega_list)
 89             if(b > 0.35):
 90                 null_feature_count[i] += 1
 91             elif ((b > 0.10) and (a > 0.10)):
 92                 null_feature_count[i] += 0.5
 93         if sum(null_feature_count) > num_feature * 0.1:
 94             print("cannot remove this many features")
 95         else:
 96             print("Here are the features we recomend you to delete")
 97             for i in range(len(null_feature_count)):
 98                 if null_feature_count[i] > 0:
 99                     print("{}: {}".format(i, null_feature_count[i]))
100
101 """### Plot the classes and features distribution for Hepatitis data"""
102
103 hepatitis_data = pd.read_csv(path1)
104 data = hepatitis_data
105 data1 = Data_Processing(data.values, 'hepatitis')
106 data1.partition_by_class()
107 data1.show_y_dist(data)
108 data1.show_x_dist(data)
109 data1.find_null_data()
110
111 """### Plot the classes and features distribution for Bankruptcy data"""
112
113 bankrupcy_data = pd.read_csv(path2)
114 data = bankrupcy_data
115 data1 = Data_Processing(data.values, 'bankrupcy')
116 data1.partition_by_class()
117 data1.show_y_dist(data)
118 data1.show_x_dist(data)
119 data1.find_null_data()
120
121 """After tests, we find that in bankrupcy.csv, if features: 43, 60, (36) are
    deleted, it may deliver a better result.
```

```
122 <br>
123 43, 60 very high p-value in ks test
124 <br>
125 36 relative high but steady p-value
126 <hr>
127
128 Hepatitis cannot use this method since we have many features are only in 1 or
    0.
129 """
```

```
# -*- 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
```

```
61          train_add = 1 if fold < self.folds else 0
62          # to exclude last term in previous partition for testing data
63          test_add = 1 if fold > 0 else 0
64
65          # number of data sets
66          n = len(self.data)
67
68          train_set_1 = data.iloc[0:int((fold)/self.folds*n), :]
69          train_set_2 = data.iloc[int((fold+1)/self.folds*n)+train_add:n, :]
70          train_set = pd.concat([train_set_1, train_set_2])
71
72          test_set = data.iloc[int((fold)/self.folds*n+test_add):int((fold+1)/
    self.folds*n), :]
73
74          train_X = train_set.iloc[:, :-1].values
75          train_y = train_set.iloc[:, -1].values
76          train_y = np.reshape(train_y, (-1,1))
77
78          test_X = test_set.iloc[:, :-1].values
79          test_y = test_set.iloc[:, -1].values
80          test_y = np.reshape(test_y, (-1,1))
81
82          return train_X, train_y, test_X, test_y
83
84      def normalization(self, X, v_X):
85          '''
86              This function performs the z-score normalization
87
88              X - training data
89              v_X - validation data
90          '''
91          mean = np.mean(X[:,1:], axis = 0)
92          sigma = np.std(X[:,1:], axis = 0)
93          mean = np.reshape(mean, (1,-1))
94          sigma = np.reshape(sigma, (1,-1))
95          X[:,1:] = (X[:,1:] - mean) / sigma
96          v_X[:,1:] = (v_X[:,1:] - mean) / sigma
97          return X, v_X
98
99      def fit(self, X, y, v_X, v_y, normalize=False):
100         '''
101             This function takes the training data X and its corresponding
    labels vector y
102             as well as other hyperparameters (such as learning rate) as input,
103             and execute the model training through modifying the model
    parameters (i.e. W).
104
105             X - training data
106             y - class of training data
107             v_X - validation data
108             v_y - class of validation data
109             epsilon - the threshold value for gradient descent
110             normalize - whether to perform normalization
111         '''
112         gradient_values, t_acc_val, v_acc_val = [], [], []
113
114         if normalize:
115             X, v_X = self.normalization(X, v_X)
116
117         # Retrive the learning rate, maximum iteration, momentum (beta)
118         lr, max_iter, beta, epsilon = self.lr, self.max_iter, self.beta, self.
    epsilon
119
```

```
120         # initial weight vector
121         w = np.zeros((len(X[0]), 1))
122         # record the best weight vector
123         best_w = w
124         # iteration number, validation accuracy, last validation accuracy
125         # the step to take in gradient descent, maximum validation accuracy
126         iteration, v_acc, step, v_acc_max = 0, 0, 0, 0
127 
128         dw = np.inf
129         # if the gradient delta w is smaller than threshold or achieved
    maximum iteration, stop
130         while (np.linalg.norm(dw) > epsilon and iteration <= max_iter):
131             dw = self.gradient(X, y, w)
132             gradient_values.append(np.linalg.norm(dw))
133             # if beta = 0, it will be the same as general gradient descent
134             step = beta * step + (1 - beta) * dw  # gradient descent with
    momentum
135             w = w - lr * step
136 
137             # predict once every 10 interations
138             if iteration % 10 == 0:
139                 t_y_pred = self.predict(X, w)
140                 t_acc = self.accu_eval(t_y_pred, y)
141                 v_y_pred = self.predict(v_X, w)
142                 v_acc = self.accu_eval(v_y_pred, v_y)
143 
144             # record the next best value
145             if v_acc >= v_acc_max:
146                 v_acc_max = v_acc
147                 best_w = w
148                 self.marker = iteration  # move the iteration marker
149 
150             t_acc_val.append(t_acc)
151             v_acc_val.append(v_acc)
152 
153             iteration = iteration + 1
154 
155         return gradient_values, t_acc_val, v_acc_val, best_w
156 
157     def predict(self, X, w):
158         '''
159             This function takes a set of data as input and outputs predicted
    labels for the input points.
160         '''
161         result = self.log_func(np.dot(X, w))
162         # the prediction result converted to binary
163         predict_bin = []
164         for i in result:
165             if i>=0.5:
166                 predict_bin.append(1)
167             else:
168                 predict_bin.append(0)
169         return predict_bin
170 
171     def accu_eval(self, y_pred, y):
172         '''
173             This function evaluates the models' accuracy.
174         '''
175         count = 0
176         for i in range(len(y_pred)):
177             if y_pred[i] == y[i]:
178                 count = count + 1
179         # return the accuracy ratio: #corret prediction / #data points
```

```
180         return count / len(y)
181
182     def log_func(self, alpha):
183         return 1 / (1 + np.exp(-alpha))
184
185     def gradient(self, X, y, w):
186         N = len(X[0])
187         y_hat = self.log_func(np.dot(X, w))
188         delta = np.dot(X.T, y_hat - y) / N
189         return delta
190
191 class KFoldValidation:
192     def __init__(self, folds, path, lr, max_iter, epsilon, beta):
193         self.folds = folds
194         self.data = pd.read_csv(path)
195         self.lr = lr
196         self.max_iter = max_iter
197         self.epsilon = epsilon
198         self.beta = beta
199
200     def k_fold_validation(self, normalize=False, inc_od=False, order=3):
201         '''
202             This function performs the k-fold validation
203
204             normalize - whether to perform normalization
205             inc_od - whether to increase the feature order
206             order - the order of the added feature
207         '''
208         folds = self.folds
209         data = self.data
210         accuracies = []
211
212         if inc_od:
213             data = self.rise_order(data, order)
214
215         log_reg = LogisticRegression(data=data, folds=self.folds, lr=self.lr,
    max_iter=self.max_iter, beta=self.beta, epsilon=self.epsilon)
216
217         log_reg.shuffle_data()
218
219         for fold in range(folds):
220             t_X, t_y, v_X, v_y = log_reg.partition(fold)
221             # t_X --> test value X, v_X --> validation value X
222             gradient_val, t_acc_val, v_acc_val, best_w = log_reg.fit(t_X, t_y
    , v_X, v_y, normalize=normalize)
223
224             accuracies.append(np.max(v_acc_val))
225
226             # Uncommant this block to display the accuracy diagram
227             plt.figure()
228             plt.plot(t_acc_val, label = 'Training accuracy')
229             plt.plot(v_acc_val, label='Validation accuracy')
230             plt.axvline(log_reg.marker, color='r', label='Best Weights')
231             plt.xlabel('Iteration Number')
232             plt.ylabel('Accuracy')
233             plt.legend()
234             plt.show()
235             print("Learning Rate: " + str(log_reg.lr))
236             print("Average Accuracy: "+str(np.mean(accuracies)))
237
238             # Uncommant this block to display the gradiant diagram
239             plt.figure()
240             plt.plot(gradient_val)
```

```
241             plt.xlabel('Iteration Number')
242             plt.ylabel('Gradiant')
243             plt.show()
244             print("-------------------------------------------------------")
245 
246         mean_acc = np.mean(accuracies)
247         return mean_acc
248 
249     def rise_order(self, data, order=3):
250         ret_val = data
251         for i in range(2, order + 1):
252             data_powered = data.pow(i)
253             ret_val = ret_val.iloc[:, :-1]
254             ret_val = pd.concat([ret_val, data_powered],axis=1)
255         return ret_val
256 
257 """### Perform 10-fold validation for Hepatitis dataset"""
258 
259 path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/hepatitis.csv"
260 dataset_name = "Hepatitis"
261 defult_lr = 0.01
262 default_max_iter = 10000
263 default_epsilon = 5e-3
264 defulat_beta = 0.99
265 
266 # the input is the optimum hyperparameters found during testing
267 hepatitis_learning = KFoldValidation(folds=10, path=path, lr=defult_lr,
    max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
268 # the input is the optimum model found during testing
269 mean_acc = hepatitis_learning.k_fold_validation()
270 
271 """### Perform 10-fold validation for Bankruptcy dataset"""
272 
273 path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/bankrupcy.csv"
274 dataset_name = "Bankruptcy"
275 defult_lr = 0.1
276 default_max_iter = 25000
277 default_epsilon = 1e-3
278 defulat_beta = 0.99
279 
280 # the input is the optimum hyperparameters found during testing
281 bankruptcy_learning = KFoldValidation(folds=10, path=path, lr=defult_lr,
    max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
282 # the input is the optimum model found during testing
283 mean_acc = bankruptcy_learning.k_fold_validation(normalize=True, inc_od=True)
```

```
# -*- coding: utf-8 -*-
"""Hyperparameter_Testing

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Tkjs6AN6HaRHO9FqtLdYzPa6BQHDaYwC

<center><h1>Mini Project 1 - Logistic Regression</h1>
<h4>This is a testing file aiming to find the best hyperparameters for the model.</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
import time

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 set_learning_rate(self, lr):
        self.lr = lr

    def set_max_iter(self, max_iter):
        self.max_iter = max_iter

    def set_epsilon(self, epsilon):
        self.epsilon = epsilon

```

```
61      def set_beta(self, beta):
62          self.beta = beta
63
64      def partition(self, fold):
65          '''
66              This function divides the dataset into training and validation set
    .
67
68              fold - the current fold
69          '''
70          data = self.data
71          # to exclude last term in previous partition for training data
72          train_add = 1 if fold < self.folds else 0
73          # to exclude last term in previous partition for testing data
74          test_add = 1 if fold > 0 else 0
75
76          # number of data sets
77          n = len(self.data)
78
79          train_set_1 = data.iloc[0:int((fold)/self.folds*n), :]
80          train_set_2 = data.iloc[int((fold+1)/self.folds*n)+train_add:n, :]
81          train_set = pd.concat([train_set_1, train_set_2])
82
83          test_set = data.iloc[int((fold)/self.folds*n+test_add):int((fold+1)/
    self.folds*n), :]
84
85          train_X = train_set.iloc[:, :-1].values
86          train_y = train_set.iloc[:, -1].values
87          train_y = np.reshape(train_y, (-1,1))
88
89          test_X = test_set.iloc[:, :-1].values
90          test_y = test_set.iloc[:, -1].values
91          test_y = np.reshape(test_y, (-1,1))
92
93          return train_X, train_y, test_X, test_y
94
95      def normalization(self, X, v_X):
96          '''
97              This function performs the z-score normalization
98
99              X - training data
100             v_X - validation data
101         '''
102         mean = np.mean(X[:,1:], axis = 0)
103         sigma = np.std(X[:,1:], axis = 0)
104         mean = np.reshape(mean, (1,-1))
105         sigma = np.reshape(sigma, (1,-1))
106         X[:,1:] = (X[:,1:] - mean) / sigma
107         v_X[:,1:] = (v_X[:,1:] - mean) / sigma
108         return X, v_X
109
110     def fit(self, X, y, v_X, v_y, normalize=False):
111         '''
112             This function takes the training data X and its corresponding
    labels vector y
113             as well as other hyperparameters (such as learning rate) as input,
114             and execute the model training through modifying the model
    parameters (i.e. W).
115
116             X - training data
117             y - class of training data
118             v_X - validation data
119             v_y - class of validation data
```

```
120             epsilon - the threshold value for gradient descent
121             normalize - whether to perform normalization
122         '''
123         gradient_values, t_acc_val, v_acc_val = [], [], []
124 
125         if normalize:
126             X, v_X = self.normalization(X, v_X)
127 
128         # Retrive the learning rate, maximum iteration, momentum (beta)
129         lr, max_iter, beta, epsilon = self.lr, self.max_iter, self.beta, self.
    epsilon
130 
131         # initial weight vector
132         w = np.zeros((len(X[0]), 1))
133         # record the best weight vector
134         best_w = w
135         # iteration number, validation accuracy, last validation accuracy
136         # the step to take in gradient descent, maximum validation accuracy
137         iteration, v_acc, step, v_acc_max = 0, 0, 0, 0
138 
139         dw = np.inf
140         # if the gradient delta w is smaller than threshold or achieved
    maximum iteration, stop
141         while (np.linalg.norm(dw) > epsilon and iteration <= max_iter):
142             dw = self.gradient(X, y, w)
143             gradient_values.append(np.linalg.norm(dw))
144             # if beta = 0, it will be the same as general gradient descent
145             step = beta * step + (1 - beta) * dw  # gradient descent with
    momentum
146             w = w - lr * step
147 
148             # predict once every 10 interations
149             if iteration % 10 == 0:
150                 t_y_pred = self.predict(X, w)
151                 t_acc = self.accu_eval(t_y_pred, y)
152                 v_y_pred = self.predict(v_X, w)
153                 v_acc = self.accu_eval(v_y_pred, v_y)
154 
155             # record the next best value
156             if v_acc >= v_acc_max:
157                 v_acc_max = v_acc
158                 best_w = w
159                 self.marker = iteration  # move the iteration marker
160 
161             t_acc_val.append(t_acc)
162             v_acc_val.append(v_acc)
163 
164             iteration = iteration + 1
165         return gradient_values, t_acc_val, v_acc_val, best_w
166 
167     def predict(self, X, w):
168         '''
169             This function takes a set of data as input and outputs predicted
    labels for the input points.
170         '''
171         result = self.log_func(np.dot(X, w))
172         # the prediction result converted to binary
173         predict_bin = []
174         for i in result:
175             if i>=0.5:
176                 predict_bin.append(1)
177             else:
178                 predict_bin.append(0)
```

```
179         return predict_bin
180
181     def accu_eval(self, y_pred, y):
182         '''
183             This function evaluates the models' accuracy.
184         '''
185         count = 0
186         for i in range(len(y_pred)):
187             if y_pred[i] == y[i]:
188                 count = count + 1
189         # return the accuracy ratio: #corret prediction / #data points
190         return count / len(y)
191
192     def log_func(self, alpha):
193         return 1 / (1 + np.exp(-alpha))
194
195     def gradient(self, X, y, w):
196         N = len(X[0])
197         y_hat = self.log_func(np.dot(X, w))
198         delta = np.dot(X.T, y_hat - y) / N
199         return delta
200
201 class KFoldValidation:
202     def __init__(self, folds, path, lr, max_iter, epsilon, beta):
203         self.folds = folds
204         self.data = pd.read_csv(path)
205         self.lr = lr
206         self.max_iter = max_iter
207         self.epsilon = epsilon
208         self.beta = beta
209         self.log_reg = LogisticRegression(data=self.data, folds=folds, lr=lr,
    max_iter=max_iter, beta=beta, epsilon=epsilon)
210         self.log_reg.shuffle_data()
211
212     def set_learning_rate(self, lr):
213         self.log_reg.set_learning_rate(lr)
214
215     def set_max_iter(self, max_iter):
216         self.log_reg.set_max_iter(max_iter)
217
218     def set_epsilon(self, epsilon):
219         self.log_reg.set_epsilon(epsilon)
220
221     def set_beta(self, beta):
222         self.log_reg.set_beta(beta)
223
224     def k_fold_validation(self):
225         '''
226             This function performs the k-fold validation
227
228             normalize - whether to perform normalization
229             inc_od - whether to increase the feature order
230             order - the order of the added feature
231         '''
232         folds = self.folds
233         data = self.data
234         log_reg = self.log_reg
235         accuracies = []
236         tic = time.time()
237
238         for fold in range(folds):
239             t_X, t_y, v_X, v_y = log_reg.partition(fold)
240             # t_X --> test value X, v_X --> validation value X
```

```
241             gradient_val, t_acc_val, v_acc_val, best_w = log_reg.fit(t_X, t_y
    , v_X, v_y)
242 
243             accuracies.append(np.mean(v_acc_val))
244 
245             # # Uncommant this block to display the accuracy diagram
246             # plt.figure()
247             # plt.plot(t_acc_val, label = 'Training accuracy')
248             # plt.plot(v_acc_val, label='Validation accuracy')
249             # plt.axvline(log_reg.marker, color='r', label='Best Weights')
250             # plt.xlabel('Iteration Number')
251             # plt.ylabel('Accuracy')
252             # plt.legend()
253             # plt.show()
254             # print("Learning Rate: " + str(log_reg.lr))
255             # print("Average Accuracy: "+str(np.mean(accuracies)))
256 
257             # # Uncommant this block to display the gradiant diagram
258             # plt.figure()
259             # plt.plot(gradient_val)
260             # plt.xlabel('Iteration Number')
261             # plt.ylabel('Gradiant')
262             # plt.show()
263             # print("-----------------------------------------------------")
264 
265         mean_acc = np.max(accuracies)
266         toc = time.time()
267         return mean_acc, toc-tic
268 
269     def rise_order(self, data, order=3):
270         ret_val = data
271         for i in range(2, order + 1):
272             data_powered = data.pow(i)
273             ret_val = ret_val.iloc[:, :-1]
274             ret_val = pd.concat([ret_val, data_powered],axis=1)
275         return ret_val
276 
277 def plot_fig(x, y, xlabel, ylabel, log_x=False, plt_name="Untitled_Figure.png"
    , download=False):
278     plt.plot(x, y)
279     if log_x:
280         plt.xscale("log")
281     plt.xlabel(xlabel)
282     plt.ylabel(ylabel)
283     if download:
284         plt.savefig(plt_name, dpi = 1200)
285         files.download(plt_name)
286     plt.show()
287 
288 """### Default Values for Hepatitis Analysis"""
289 
290 path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/hepatitis.csv"
291 dataset_name = "Hepatitis"
292 defult_lr = 0.01
293 default_max_iter = 10000
294 default_epsilon = 5e-3
295 defulat_beta = 0.99
296 
297 """### Learning rate testing"""
298 
299 lr_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=
    default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
300 
```

```
301 Learning_rates = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1]
302
303 mean_acc = []
304 proc_time = []
305
306 for lr in Learning_rates:
307     lr_testing.set_learning_rate(lr)
308     mean_acc_temp, time_temp = lr_testing.k_fold_validation()
309     mean_acc.append(mean_acc_temp)
310     proc_time.append(time_temp)
311
312 plot_fig(Learning_rates, mean_acc, "log(Learning Rate)", "Mean Accuracy",
    log_x=True, plt_name="Validation_Accuracy_vs_Learning_Rate_for_{}.png".format(
    dataset_name))
313 plot_fig(Learning_rates, proc_time, "log(Learning Rate)", "Processing Time",
    log_x=True, plt_name="Validation_Accuracy_vs_Learning_Rate_for_{}.png".format(
    dataset_name))
314
315 """### Maximum Iteration Test"""
316
317 max_iter_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter
    =default_max_iter, epsilon=1e-3, beta=defulat_beta)
318
319 max_iters = [500, 1000, 5000, 10000, 25000]
320
321 mean_acc = []
322 proc_time = []
323
324 for max_iter in max_iters:
325     max_iter_testing.set_max_iter(max_iter)
326     mean_acc_temp, time_temp = max_iter_testing.k_fold_validation()
327     mean_acc.append(mean_acc_temp)
328     proc_time.append(time_temp)
329
330 plot_fig(max_iters, mean_acc, "log(Maximum Iterations)", "Mean Accuracy",
    log_x=True, plt_name="Validation_Accuracy_vs_Maximum_Iterations_for_{}.png".
    format(dataset_name))
331 plot_fig(max_iters, proc_time, "log(Maximum Iterations)", "Processing Time",
    log_x=True, plt_name="Processing_Time_vs_Maximum_Iterations_for_{}.png".format
    (dataset_name))
332
333 """### Epsilon Test"""
334
335 epsilon_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=
    default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
336
337 epsilons = [1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 0.5]
338
339 mean_acc = []
340 proc_time = []
341
342 for epsilon in epsilons:
343     epsilon_testing.set_epsilon(epsilon)
344     mean_acc_temp, time_temp = epsilon_testing.k_fold_validation()
345     mean_acc.append(mean_acc_temp)
346     proc_time.append(time_temp)
347
348 plot_fig(epsilons, mean_acc, "log(Epsilons)", "Mean Accuracy", log_x=True,
    plt_name="Validation_Accuracy_vs_Epsilons_for_{}.png".format(dataset_name))
349 plot_fig(epsilons, proc_time, "log(Epsilons)", "Processing Time", log_x=True,
    plt_name="Processing_Time_vs_Epsilons_for_{}.png".format(dataset_name))
350
351 """### Momentum Gradient Descent Constant - Beta Testing"""
```

```
352
353 beta_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=
    default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
354
355 betas = [0, 0.5, 0.9, 0.99, 0.999]
356
357 mean_acc = []
358 proc_time = []
359
360 for beta in betas:
361     beta_testing.set_beta(beta)
362     mean_acc_temp, time_temp = beta_testing.k_fold_validation()
363     mean_acc.append(mean_acc_temp)
364     proc_time.append(time_temp)
365
366 plot_fig(betas, mean_acc, "betas", "Mean Accuracy", log_x=False, plt_name="
    Validation_Accuracy_vs_betas_for_{}.png".format(dataset_name))
367 plot_fig(betas, proc_time, "betas", "Processing Time", log_x=False, plt_name="
    Processing_Time_vs_betas_for_{}.png".format(dataset_name))
368
369 """<hr>
370
371 ### Default Values for Bankruptcy Analysis
372 """
373
374 path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/bankrupcy.csv"
375 dataset_name = "Bankruptcy"
376 defult_lr = 0.1
377 default_max_iter = 25000
378 default_epsilon = 1e-3
379 defulat_beta = 0.99
380
381 """### Learning rate testing"""
382
383 lr_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=
    default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
384
385 Learning_rates = [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1]
386
387 mean_acc = []
388 proc_time = []
389
390 for lr in Learning_rates:
391     lr_testing.set_learning_rate(lr)
392     mean_acc_temp, time_temp = lr_testing.k_fold_validation()
393     mean_acc.append(mean_acc_temp)
394     proc_time.append(time_temp)
395
396 plot_fig(Learning_rates, mean_acc, "log(Learning Rate)", "Mean Accuracy",
    log_x=True, plt_name="Validation_Accuracy_vs_Learning_Rate_for_{}.png".format(
    dataset_name))
397 plot_fig(Learning_rates, proc_time, "log(Learning Rate)", "Processing Time",
    log_x=True, plt_name="Validation_Accuracy_vs_Learning_Rate_for_{}.png".format(
    dataset_name))
398
399 """### Maximum Iteration Test"""
400
401 max_iter_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter
    =default_max_iter, epsilon=1e-3, beta=defulat_beta)
402
403 max_iters = [500, 1000, 5000, 10000, 25000]
404
405 mean_acc = []
```

```
proc_time = []

for max_iter in max_iters:
    max_iter_testing.set_max_iter(max_iter)
    mean_acc_temp, time_temp = max_iter_testing.k_fold_validation()
    mean_acc.append(mean_acc_temp)
    proc_time.append(time_temp)

plot_fig(max_iters, mean_acc, "log(Maximum Iterations)", "Mean Accuracy", log_x=True, plt_name="Validation_Accuracy_vs_Maximum_Iterations_for_{}.png".format(dataset_name))
plot_fig(max_iters, proc_time, "log(Maximum Iterations)", "Processing Time", log_x=True, plt_name="Processing_Time_vs_Maximum_Iterations_for_{}.png".format(dataset_name))

"""### Epsilon Test"""

epsilon_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)

epsilons = [1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 0.5]

mean_acc = []
proc_time = []

for epsilon in epsilons:
    epsilon_testing.set_epsilon(epsilon)
    mean_acc_temp, time_temp = epsilon_testing.k_fold_validation()
    mean_acc.append(mean_acc_temp)
    proc_time.append(time_temp)

plot_fig(epsilons, mean_acc, "log(Epsilons)", "Mean Accuracy", log_x=True, plt_name="Validation_Accuracy_vs_Epsilons_for_{}.png".format(dataset_name))
plot_fig(epsilons, proc_time, "log(Epsilons)", "Processing Time", log_x=True, plt_name="Processing_Time_vs_Epsilons_for_{}.png".format(dataset_name))

"""### Momentum Gradient Descent Constant - Beta Testing"""

beta_testing = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)

betas = [0, 0.5, 0.9, 0.99, 0.999]

mean_acc = []
proc_time = []

for beta in betas:
    beta_testing.set_beta(beta)
    mean_acc_temp, time_temp = beta_testing.k_fold_validation()
    mean_acc.append(mean_acc_temp)
    proc_time.append(time_temp)

plot_fig(betas, mean_acc, "betas", "Mean Accuracy", log_x=False, plt_name="Validation_Accuracy_vs_betas_for_{}.png".format(dataset_name))
plot_fig(betas, proc_time, "betas", "Processing Time", log_x=False, plt_name="Processing_Time_vs_betas_for_{}.png".format(dataset_name))
```

```
# -*- coding: utf-8 -*-
"""Normalization_Feature_Testing

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/103D1WTRS_XIebFMiTu3iE6eSALqeQn_S

<center><h1>Mini Project 1 - Logistic Regression</h1>
<h4>This is a testing file aiming to find the effect of normalization, increasing feacures on the model.</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
import time
from google.colab import files

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
```

```
61          # to exclude last term in previous partition for testing data
62          test_add = 1 if fold > 0 else 0
63
64          # number of data sets
65          n = len(self.data)
66
67          train_set_1 = data.iloc[0:int((fold)/self.folds*n), :]
68          train_set_2 = data.iloc[int((fold+1)/self.folds*n)+train_add:n, :]
69          train_set = pd.concat([train_set_1, train_set_2])
70
71          test_set = data.iloc[int((fold)/self.folds*n+test_add):int((fold+1)/
    self.folds*n), :]
72
73          train_X = train_set.iloc[:, :-1].values
74          train_y = train_set.iloc[:, -1].values
75          train_y = np.reshape(train_y, (-1,1))
76
77          test_X = test_set.iloc[:, :-1].values
78          test_y = test_set.iloc[:, -1].values
79          test_y = np.reshape(test_y, (-1,1))
80
81          return train_X, train_y, test_X, test_y
82
83      def normalization(self, X, v_X):
84          '''
85              This function performs the z-score normalization
86
87              X - training data
88              v_X - validation data
89          '''
90          mean = np.mean(X[:,1:], axis = 0)
91          sigma = np.std(X[:,1:], axis = 0)
92          mean = np.reshape(mean, (1,-1))
93          sigma = np.reshape(sigma, (1,-1))
94          X[:,1:] = (X[:,1:] - mean) / sigma
95          v_X[:,1:] = (v_X[:,1:] - mean) / sigma
96          return X, v_X
97
98      def fit(self, X, y, v_X, v_y, normalize=False):
99          '''
100             This function takes the training data X and its corresponding
    labels vector y
101             as well as other hyperparameters (such as learning rate) as input,
102             and execute the model training through modifying the model
    parameters (i.e. W).
103
104             X - training data
105             y - class of training data
106             v_X - validation data
107             v_y - class of validation data
108             epsilon - the threshold value for gradient descent
109             normalize - whether to perform normalization
110         '''
111         gradient_values, t_acc_val, v_acc_val = [], [], []
112
113         if normalize:
114             X, v_X = self.normalization(X, v_X)
115
116         # Retrive the learning rate, maximum iteration, momentum (beta)
117         lr, max_iter, beta, epsilon = self.lr, self.max_iter, self.beta, self.
    epsilon
118
119         # initial weight vector
```

```
120         w = np.zeros((len(X[0]), 1))
121         # record the best weight vector
122         best_w = w
123         # iteration number, validation accuracy, last validation accuracy
124         # the step to take in gradient descent, maximum validation accuracy
125         iteration, v_acc, step, v_acc_max = 0, 0, 0, 0
126 
127         dw = np.inf
128         # if the gradient delta w is smaller than threshold or achieved
    maximum iteration, stop
129         while (np.linalg.norm(dw) > epsilon and iteration <= max_iter):
130             dw = self.gradient(X, y, w)
131             gradient_values.append(np.linalg.norm(dw))
132             # if beta = 0, it will be the same as general gradient descent
133             step = beta * step + (1 - beta) * dw  # gradient descent with
    momentum
134             w = w - lr * step
135 
136             # predict once every 10 interations
137             if iteration % 10 == 0:
138                 t_y_pred = self.predict(X, w)
139                 t_acc = self.accu_eval(t_y_pred, y)
140                 v_y_pred = self.predict(v_X, w)
141                 v_acc = self.accu_eval(v_y_pred, v_y)
142 
143             # record the next best value
144             if v_acc >= v_acc_max:
145                 v_acc_max = v_acc
146                 best_w = w
147                 self.marker = iteration  # move the iteration marker
148 
149             t_acc_val.append(t_acc)
150             v_acc_val.append(v_acc)
151 
152             iteration = iteration + 1
153 
154         # Uncommant to display the confusion matrix
155         # self.confusion_matrix(v_y_pred, v_y)
156         return gradient_values, t_acc_val, v_acc_val, best_w
157 
158     def predict(self, X, w):
159         '''
160             This function takes a set of data as input and outputs predicted
    labels for the input points.
161         '''
162         result = self.log_func(np.dot(X, w))
163         # the prediction result converted to binary
164         predict_bin = []
165         for i in result:
166             if i>=0.5:
167                 predict_bin.append(1)
168             else:
169                 predict_bin.append(0)
170         return predict_bin
171 
172     def accu_eval(self, y_pred, y):
173         '''
174             This function evaluates the models' accuracy.
175         '''
176         count = 0
177         for i in range(len(y_pred)):
178             if y_pred[i] == y[i]:
179                 count = count + 1
```

```
180         # return the accuracy ratio: #corret prediction / #data points
181         return count / len(y)
182 
183     def log_func(self, alpha):
184         return 1 / (1 + np.exp(-alpha))
185 
186     def gradient(self, X, y, w):
187         N = len(X[0])
188         y_hat = self.log_func(np.dot(X, w))
189         delta = np.dot(X.T, y_hat - y) / N
190         return delta
191 
192     def confusion_matrix(self, y_pred, y):
193         y = y.reshape(-1)
194         data = {'Actual_y': y,
195                 'Predicted_y': y_pred}
196         df = pd.DataFrame(data, columns=['Actual_y', 'Predicted_y'])
197         confusion_matrix = pd.crosstab(df['Actual_y'], df['Predicted_y'],
    rownames=['Actual'], colnames=['Predicted'])
198         svm = sns.heatmap(confusion_matrix, annot=True, cmap="YlGnBu")
199 
200         # Uncomment this part to download the confusion matrix.
201         # temp_time = time.time()
202         # figure = svm.get_figure()
203         # figure.savefig("Confusion_Matrix_{}.png".format(temp_time), dpi =
    1200)
204         # files.download("Confusion_Matrix_{}.png".format(temp_time))
205 
206 class KFoldValidation:
207     def __init__(self, folds, path, lr, max_iter, epsilon, beta):
208         self.folds = folds
209         self.path = path
210         self.lr = lr
211         self.max_iter = max_iter
212         self.epsilon = epsilon
213         self.beta = beta
214 
215     def k_fold_validation(self, normalize=False, inc_od=False, order=3,
    rm_features=False):
216         '''
217             This function performs the k-fold validation
218 
219             normalize - whether to perform normalization
220             inc_od - whether to increase the feature order
221             order - the order of the added feature
222         '''
223         folds = self.folds
224         data = pd.read_csv(self.path)
225 
226         accuracies = []
227         accuracies_train = []
228         tic = time.time()
229 
230         if rm_features:
231             data.drop(columns=['attribute43', 'attribute60'])
232 
233         if inc_od:
234             print("order rise to {}".format(order))
235             data = self.rise_order(data, order)
236 
237         log_reg = LogisticRegression(data=data, folds=self.folds, lr=self.lr,
    max_iter=self.max_iter, beta=self.beta, epsilon=self.epsilon)
238 
```

```
239         log_reg.shuffle_data()
240 
241         for fold in range(folds):
242             t_X, t_y, v_X, v_y = log_reg.partition(fold)
243             # t_X --> test value X, v_X --> validation value X
244             gradient_val, t_acc_val, v_acc_val, best_w = log_reg.fit(t_X, t_y
    , v_X, v_y, normalize=normalize)
245 
246             accuracies.append(np.max(v_acc_val))
247             accuracies_train.append(np.max(t_acc_val))
248 
249             # Uncommant this block to display the accuracy diagram
250             plt.figure()
251             plt.plot(t_acc_val, label = 'Training accuracy')
252             plt.plot(v_acc_val, label='Validation accuracy')
253             plt.axvline(log_reg.marker, color='r', label='Best Weights')
254             plt.xlabel('Iteration Number')
255             plt.ylabel('Accuracy')
256             plt.legend()
257             plt.show()
258             print("Learning Rate: " + str(log_reg.lr))
259             print("Average Accuracy: "+str(np.mean(accuracies)))
260 
261             # Uncommant this block to display the gradiant diagram
262             plt.figure()
263             plt.plot(gradient_val)
264             plt.xlabel('Iteration Number')
265             plt.ylabel('Gradiant')
266             plt.show()
267             print("------------------------------------------------------------")
268 
269         mean_acc = np.mean(accuracies)
270         mean_acc_train = np.mean(accuracies_train)
271         toc = time.time()
272         return mean_acc, mean_acc_train, toc-tic
273 
274     def rise_order(self, data, order=3):
275         ret_val = data
276         for i in range(2, order + 1):
277             data_powered = data.pow(i)
278             ret_val = ret_val.iloc[:, :-1]
279             ret_val = pd.concat([ret_val, data_powered],axis=1)
280         return ret_val
281 
282 """### Default Values"""
283 
284 # path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/hepatitis.
    csv"
285 # dataset_name = "Hepatitis"
286 # defult_lr = 0.01
287 # default_max_iter = 10000
288 # default_epsilon = 5e-3
289 # defulat_beta = 0.99
290 
291 path = "/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_01/bankrupcy.csv"
292 dataset_name = "Bankruptcy"
293 defult_lr = 0.1
294 default_max_iter = 25000
295 default_epsilon = 1e-3
296 defulat_beta = 0.99
297 
298 unnorm_ordering = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=
    default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
```

```
normed_ordering = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)

order_min = 1
order_max = 10

orders = np.arange(order_min, order_max+1)

mean_acc = []
mean_acc_train = []
proc_time = []
mean_acc_norm = []
mean_acc_train_norm = []
proc_time_norm = []

for order in orders:
    mean_acc_temp, mean_acc_train_temp, time_temp = unnorm_ordering.k_fold_validation(normalize=False, inc_od=True, order=order)
    mean_acc.append(mean_acc_temp)
    mean_acc_train.append(mean_acc_train_temp)
    proc_time.append(time_temp)
    mean_acc_temp, mean_acc_train_temp, time_temp = normed_ordering.k_fold_validation(normalize=True, inc_od=True, order=order)
    mean_acc_norm.append(mean_acc_temp)
    mean_acc_train_norm.append(mean_acc_train_temp)
    proc_time_norm.append(time_temp)

plt.plot(orders, mean_acc_train,'--', label='Unnormalized Training Accuracy')
plt.plot(orders, mean_acc,'--', label='Unnormalized Validation Accuracy')
plt.plot(orders, mean_acc_train_norm, label='Normalized Training Accuracy')
plt.plot(orders, mean_acc_norm, label='Normalized Validation Accuracy')
plt.legend()
plt.xlabel("Maximum Order")
plt.ylabel("Mean Accuracy")
# Uncomment this to save the accuracy vs Order figure
# plt.savefig("Validation_Accuracy_vs_Maximum_Order.png", dpi = 1200)
# files.download("Validation_Accuracy_vs_Maximum_Order.png")
plt.show()

plt.plot(orders, proc_time,'--', label='Unnormalized Processing Time')
plt.plot(orders, proc_time_norm, label='Normalized Processing Time')
plt.legend()
plt.xlabel("Maximum Order")
plt.ylabel("Processing Time")
# Uncomment this to save the Processing time vs Order figure
# plt.savefig("Processing_Time_vs_Maximum_Order.png", dpi = 1200)
# files.download("Processing_Time_vs_Maximum_Order.png")
plt.show()

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

full_feature_test = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)
rm_feature_test = KFoldValidation(folds=10, path=path, lr=defult_lr, max_iter=default_max_iter, epsilon=default_epsilon, beta=defulat_beta)

mean_acc = []
proc_time = []
```

```
357
358 mean_acc_temp, mean_acc_train_temp, time_temp = full_feature_test.
    k_fold_validation(normalize=True, inc_od=True)
359 mean_acc.append(mean_acc_temp)
360 proc_time.append(time_temp)
361 mean_acc_temp, mean_acc_train_temp, time_temp = rm_feature_test.
    k_fold_validation(normalize=True, inc_od=True, rm_features=True)
362 mean_acc.append(mean_acc_temp)
363 proc_time.append(time_temp)
364
365 print("Without removing attribute 43 and 60\tmean accuracy is {} \tprocessing
    time is {}\nAfter removing those two attribute\tmean accuracy is {} \t
    processing time is {}".format(mean_acc[0],proc_time[0],mean_acc[1],proc_time[1
    ]))
```