COMP 4900A Mini-Project 1

Connor Stewart (101041125)
Michael Kazman (101037568)
Trevor Johns(101036054)

Majid Komeilli

October 14th, 2020

1 Abstract

The purpose of this report is to provide a look at the results of a self-directed project under the supervision of Professor Majid Komeili. The project is to design and implement the well-known linear classifier, logistic regression (LC-LR), from the ground-up. The LC-LR tool is made to run a binary classification on two sets of data – namely a hepatitis data set and a bankruptcy data set. The purpose of the classification is to determine any correlations between the data and any recorded features. The tool was designed with mathematical and statistical precision in mind, as to replicate the confidence associated with the linear classification algorithm. Furthermore, the code base was centered around modularity, such that both datasets – with their differing number of observations and variables – can be automated into the LC-LR pipeline. After a multitude of analysis and computation, further insights have been inferred. In the report herein, we will discuss the data sets, results, as well as their significance.

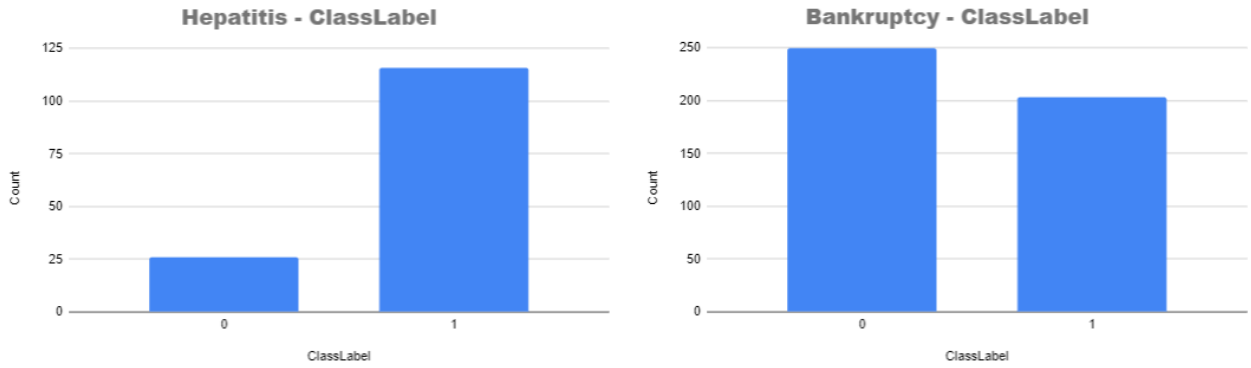
2 Introduction

The purpose of the project is to analyze the aforementioned data sets using a linear classification algorithm. This allows for the discovery of a correlation between the observations and features involved in the experiment. The data contains a large array of features with the bankruptcy set containing sixty-five and the hepatitis twenty. Without removing any features from the dataset, using an epsilon value of 1e-8, a learning rate coefficient of 0.31, and a maximum step size of 6000 it was possible to achieve an overall probability of 86.6% over the ten folds for the hepatitis data set. The following is the formula used to calculate the learning rate, $\frac{1}{e^{(k \times self.learning\_rate)}}$. The learning rate formula is calculated using a static learning_rate (0.31 in this case) and k (the current step for the current fold's gradient descent). During this process the current fold's norm vector converges to zero following a direct convergence for the k steps. When training the bankruptcy data set without feature reduction, using a learning rate coefficient of 0.5705, a learning formula of $\frac{1}{k^{(0.5705)}}$, an epsilon value of 0.001, and a maximum step size of 3000, we see a 75.3% confidence. After removing irrelevant columns from the training set, the overall accuracy for hepatitis classification increased to approximately 89.4%. The overall accuracy for the bankruptcy data set increased to approximately 99.2%. Irrelevant training data refers to features within the data set that do not appear to have any correlation with the class labels. These columns seemingly appear with random distributions and contain no apparent pattern. Removing irrelevant data increases the probability of the generated models. Epsilon values are set to the same point as in the previous program, as well as a maximum step-size set to 5000 for each fold within each data set. Both models identical learning rate functions of $\frac{1}{e^{(k \times self.learning\_rate)}}$.

3 Datasets

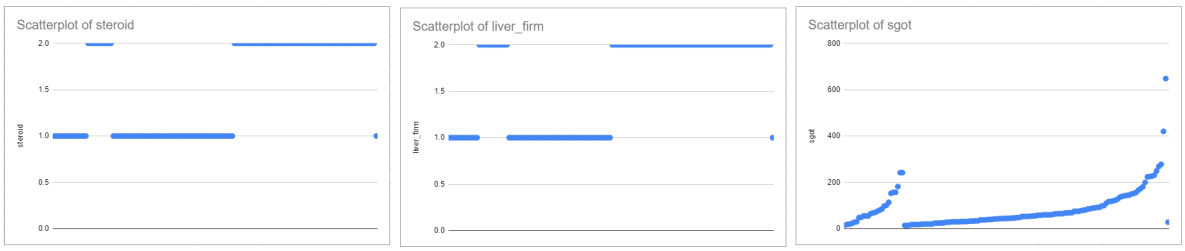
The data sets themselves both contain sets of classes organized into columns, along with rows representing independent observations. The number of classes (columns) is substantial, and combined with the number of observations (rows) provides a very high number of possible permutations. These permutations act as correlations between the classes and independent observations. In total, the bankruptcy data set contains 454 data points per class along with 65 total classes, with the last representing the corresponding class label for each observation. Note that there is a combined total of 29510 data points. In total, the hepatitis data set contains 143 data points and 20 classes providing a total of 2860 data points. Each point in the provided data sets are represented using numerical values, consisting of either real number values derived from measurements, or integer values representing a category (such as *sex*, *steroid*, etc.).

We can see that there are some clear differences between the two data sets. Hepatitis features an overwhelming majority of value 1 to 0. In contrast to the bankruptcy data set, this displays a more even distribution between the two. This can be seen in later graphs where the data is split relative to where the class changes.



To determine which attributes should be excluded from the model, the attributes were looked over manually and sorted by the class label to separate the classes. Following, graphs were made and for each attribute to identify similar distributions regarding both classifications. This provided insight into which features were strongly correlated certain class labels. The features we choose were attribute 36, 44, 59, 61 and 64 for the bankruptcy data set and *steroid*, *liver_firm* and *sgot* for the hepatitis data set.
These show the distribution for most of the removed bankruptcy features.

These show the distribution of the removed hepatitis features.



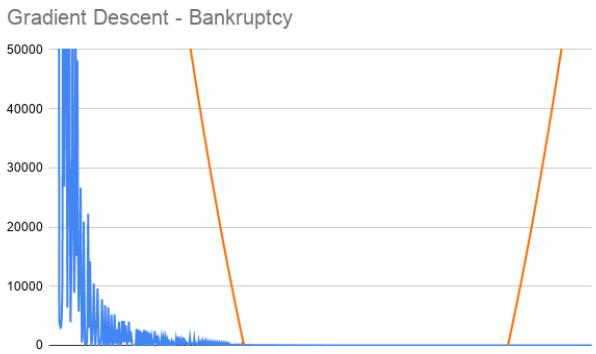
4 Results

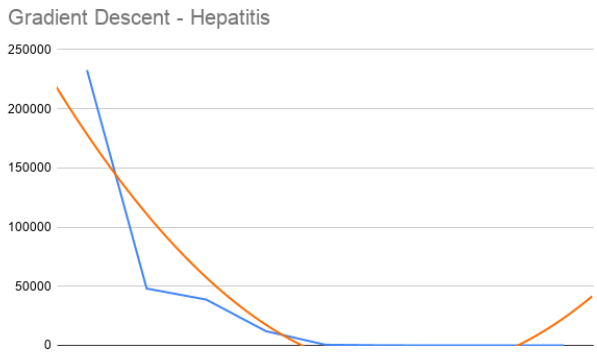
Throughout this project, a linear regression model was implemented utilizing logistic regression, gradient descent, and k-folds cross-validation. The prediction accuracies for this supervised model are 99.2% and 89.4% for bankruptcy and hepatitis, respectively. Within the bankruptcy data it was commonly found that the majority of validation sets were correctly predicted, with a few outliars (4) containing an 98% accuracy. The hepatitis commonly outputs accuracy scores of 86%, 93%, 100%, and occasionally 64% depending on the arrangement of the selected fold, with a high frequency lying on 86% and 93%.

This involved different procedures such as feature reduction, scaling learning rates, and a dynamic learning rate function. Each of the aforementioned procedures play an essential role in ensuring adequate confidence values are achieved. Feature reduction was used to reduce the irrelevant features that would mislead the machine learning model. By searching and sorting through columns containing virtually no correlation to either classes, the accuracy for both data sets increased drastically. By heavily reducing the bias caused by irrelevant data, not only does the implementation become more accurate but the run-time is reduced. This could be further improved through future expansion on the data sets known to have a high correlation between the two classes. This was not fully implemented due to time constraints, as determining the correct scaling factors for feature expansion is time consuming and often done through trial-and-error. However, by having a majority of data sets relate to certain class labels, prediction occurs at a much higher accuracy. This can be done with either feature reduction, expansion, or both, however there are pros and cons of each. With feature expansion implemented in the project, the transformations tested and applied to various columns seemed to provide a much higher inconsistency within the data set, and thus the functionality was removed. However, given additional data sets, and much more time, improved transformations regarding the feature expansion could have been implemented.

It is known that the logistic regression performance (converge speed) depends on the learning rate as the value is descending further and further towards the local minimum. This occurs until the changes between steps is less than a specified value, epsilon. Rather than using a static learning rate, which would cause run-time to be exponentially slower, the model scales the learning rate by a factor of k (where k is the current gradient descent step). This greatly increases the converge speed and allows the model to descend in a reasonable number of steps. Furthermore, the initial implementation of our machine learning model featured a dynamic learning rate function which occasionally increased accuracy. The function would shift the calculation formula depending on the current k value. It was based on a percentage of the data set, in order to prevent oscillations occurring at lower values of k. This was later abandoned in favor of an exponential learning rate which provided more consistent results.

The following images showcase a single fold from each data set, to provide a visual explanation of the model's gradient descent.





Through the above, we can see the descent as the plotted norm value reaches the local minimum. With the creation of this supervised machine learning model, much was learned about how the learning rate, epsilon value, and maximum steps (regarding gradient descent) can heavily affect the generated accuracy scores. It was found that even a slight tweak in a scaling learning rate can have drastic consequences and lead to values quite dissimilar from previous computations. The same can be said about the epsilon and maximum steps, which both prevent the algorithm from ever descending to the appropriate minimum value. Convergence speed ties hand-in-hand with run-time as the previous hyperparameters play a tremendous role in the duration of computation. Lowering the learning rate, or epsilon, can drastically increase the run-time. The same can be said about the maximum number of steps the algorithm is allowed. This reason is that these variables are heavily tied to convergence speed, and thus affect the rate at which descension occur.

5 Discussion and Conclusion

Throughout this project, a deep dive into the mechanics of gradient descent with logistic regression was conducted. This helped foster a better understanding of step-based analysis works and how convergence – uniform and oscillatory – can be reached, and how oscillation and divergence can be avoided. Custom algorithms for the learning rate was made such that smaller values for the learning rate were derived for each successive iteration of the folding loop, causing the norm vector within the folding loop to drop quickly (from a high value) at first to start dropping to smaller and smaller values as the epsilon value was approached. This allowed for both a stepwise regression model to be built and for the application to maximize the number of steps during the learning process. This in turn increased the probability of our model, thereby bringing the resulting probability for the first dataset (bankruptcy) to approximately 99.2% and the probability of our second dataset (hepatitis) to approximately 89.4% (when rounded to the first decimal). Probability scores of this magnitude across an average of ten folds represents a high probability that an accurate model was generated by the algorithm. Namely, the Bankruptcy model with its 99.2% accuracy is nearly certain to be a correct model, whereas the hepatitis model with its 89.4% accuracy is very confidently a correct model.

The model's accuracy was improved from a baseline of 86.6% confidence for the hepatitis data set and a 75.3% confidence for the bankruptcy data set by removing data columns without any correlation to the overall model. Data sets without any valid correlations were found via the use of Microsoft Excel when comparing the column values to the class label using the scatterplot function. This resulted in the removal of columns 64, 61, 59, 44, and 36 from the bankruptcy data set and columns *steroid*, *liver_firm*, and *sgot* from the hepatitis data set. In turn, this brought the culminative accuracies of both ten-fold models up to the values stated in the previous above (specifically, 99.2% and 89.4% for bankruptcy and hepatitis respectively).

We could extend the application by allowing it to have an even more accurate learning. This could be achieved through the creation of more sophisticated and precise learning rate generator functions. This would thereby allow for a more accurate model during the folding process, which has the potential to bring up accuracy for the bankruptcy data set (by a small margin) and the hepatitis data set (by a couple percent). Furthermore, added accuracy could be included using a statistical linear regression model calculator, multilinear regression

model calculator, or any other statistical significance calculator on the excel data set. This would allow for the rejection of any excel columns which do not have a valid confidence up to a specified probability (p) value. Rejecting insignificant columns would allow us to further filter out data which is irrelevant to the overall model's accuracy. Another addition could include a combinatorial pairing device which attempts to couple every possible permutation of columns and class labels together to determine the best possible combinations of columns and labels for a correlation. A combinatorial optimizer would be a final way to attempt to improve the model via data filtration, as testing all possible sets of columns against all possible sets of class labels could reveal other significant correlations between the values in the data set. Lastly, accuracy for the models could be changed by modifying the number of folds. For example, the bankruptcy model was tested to be improved by using more folds in the model (eleven folds specifically). This increase resulted in a probability of approximately 99.5% for the bankruptcy model, representing an increase of roughly 0.3%.

In conclusion, the project investigated the inner workings behind a logistic regression classifier (via gradient decent). This allowed for the building of a practical real-time application to demonstrate effective machine learning algorithms. This both granted us insight into the field of supervised machine learning, gradient decent, and logistic regression as well as provided a working understanding of learning rates, cross-fold validation, and model tuning through hyperparameters. Finally, through the completion of this project, demonstrations were made for the practical understanding of complex data sets. This provides an insightful first look into the structure of data sets present in both the real world, and machine learning models. Developing an understanding of the structures within financial and medical data is an invaluable asset that is seldom provided in academic environments.

6 Statements of Contributions

While each team member was responsible for creating components of the logistic classifier python class as well as debugging, the foundation for the logistic regression classifier was primarily maintained by Michael Kazman, as well as Trevor Johns. Mathematical functions were created and fine-tuned by Michael Kazman and Connor Stewart. Data visualization and analysis was completed by Trevor Johns using both Excel, and matplotlib to generate histograms as well as determine what columns are irrelevant for the machine learning model.

7 Appendix

7.1 Appendix A – Main Program (Program Used to Build Model)

```
#============================================================================================
# Code used for machine learning model when dropping irrelevant training data columns (
    red↪ main model)
# Generates values of 99.2% and 89.4% for the bankruptcy and hepatitis datasets
    red↪ respectively
#============================================================================================

# required libraries
import pandas as pd, numpy as np, math, random
import threading


# read in datasets ("file", epsilon)
bankruptcy_data = pd.read_csv('bankrupcy.csv')
hepatitis_data = pd.read_csv('hepatitis.csv')
```

```
"""
Name:
    logistic_classifier
Input:
    file - this dataset being used
    epsilon - error rate where we stop the current fold
    learning_rate - the rate at which we learn during folds
    max_iter - maximum number of iterations of a folding loop before program gives up on
        red↪ trying to reach the epsilon value
    number - this is the scaler number representing the dataset being used
Output:
    Returns the probability of the model generated after k folds
Description:
    Main class for the gradient decent logistic regression machine learning device
"""
class logistic_classifier :
    """
    Name:
        __init__
    Input:
        file - this dataset being used
        epsilon - error rate where we stop the current fold
        learning_rate - the rate at which we learn during folds
        max_iter - maximum number of iterations of a folding loop before program gives up
            red↪ on trying to reach the epsilon value
        number - this is the scaler number representing the dataset being used
    Output:
        Initializes the object
    Description:
        Constructor for the object
    """
    def __init__(self, file, epsilon, learning_rate, max_iter, number):
        self.dataset = number
        (x, y) = self.parse_matrices(file)
        self.x = x # X matrix (rows of data set)
        self.y = y # Y matrix (class labels)
        self.epsilon = epsilon # small convergence value
        self.learning_rate = learning_rate # learning rate (alpha)
        self.num_rows = len(x)
        self.num_features = len(x[0]) # m (number of features in x)
        self.num_folds = 10 # number of folds
        self.number = number
        self.max_iter = max_iter
    #END __init__

    #map function for the bankruptcy dataset
    def mapToBankruptcy(self, feature):
        return 'attribute' + str(feature)

    #prunes the dataset for the databases
    def prune_data(self, file, irrelevant):
        return file.drop(labels=irrelevant, axis="columns")
```

```
    """
    Name:
        parse_matrices
    Input:
        file - this is the dataset being read in
    Output:
        Outputs a parsed dataset for the machine learning application
    Description:
        converts the dataset to an array of datapoints, also removes columns which are
            red↪ deemed irrelevent for the machine learning model.
    """
    def parse_matrices(self, file):
        # identifiers for class label and bias features
        class_identifier = 'ClassLabel'
        bias_identifier = 'BiasTerm'

        # define X, Y, and row matrices
        y = np.longdouble(file[class_identifier]).reshape(-1, 1)
        x = file.drop(labels=class_identifier, axis="columns")

        # bankruptcy
        if (self.dataset == 0):
            # map each irrelevant column to a bankruptcy feature
            irrelevant = [64, 61, 59, 44, 36]
            irrelevant = list(map(self.mapToBankruptcy, irrelevant))

        # hepatitis
        elif (self.dataset == 1):
            # irrelevant columns
            irrelevant = ['steroid', 'liver_firm', 'sgot']

        # prune data
        x = self.prune_data(file, irrelevant)

        # add dummy values
        x[bias_identifier] = 1

        # convert x to numpy object
        x = x.to_numpy()

        return (x, y)
    #END parse_matrices

    #used to deparate output for debugging
    def debug(self):
        return '\n------------------------------'
    #END debug

    """
    Name:
        fit
    Input:
        x - this represents the training data
        y - this represents the class label data
```

```
    Output:
        outputs the current weight vector for the fit
    Description:
        determines the current weight vector for the given training data and class label
    """
    def fit(self, x, y):
        # initialize descent varaibles
        k = 1
        norm_vector = self.epsilon
        first_pass = True

        # weight placeholders
        #current_weight = np.random.normal(0, 0.01, self.num_features)
        current_weight = np.ones(self.num_features)

        new_weight = None

        # loop until convergence
        while ((first_pass or norm_vector >= self.epsilon) and k <= self.max_iter):
            # "do" while condition
            if (first_pass):
                first_pass = False
            #  (delta)
            delta_value = self.delta(x, y, current_weight)
            # k (learning rate)
            alpha = self.calc_learning_rate(k)
            # set new weight value
            new_weight = np.subtract(current_weight, np.multiply(alpha, delta_value))
            # change in descent
            descent_change = np.subtract(new_weight, current_weight)
            # calculate norm (for checking stop condition)
            norm_vector = np.linalg.norm(descent_change, ord=2) ** 2
            #print(norm_vector)
            # increment k
            k += 1
            # set current to newer value (for next iteration)
            current_weight = new_weight
            # in case it takes too long
            if k > self.max_iter:
                # debug
                print('~ Cannot descent, max iteration reached')
        #END WHILE
        return current_weight
    #END gradient_descent

    """
    Name:
        delta
    Input:
        x - this represents the training data
        y - this represents the class label data
        w - this is the current weight vector for the fitted model
    Output:
        determines the sums of the multipled matracies
```

```
    Description:
        It finds the change in slope for the gradient decent logistic regression model
    """
    def delta(self, x, y, w):
        # matrix of updated weights
        sums = 0

        # go through rows of X/Y
        for i in range(1, len(x)):
            # wT
            w_transpose = np.transpose(w)
            # wT * xi
            w_transpose_x = np.matmul(w_transpose, x[i])
            # yi - (wT * xi)
            true_value = y[i][0] - self.sigmoid(w_transpose_x)
            # xi * yi - (wT * xi)
            result = x[i] * true_value
            # add to sum
            sums = np.add(sums, result)
        # return -
        return -sums
    #END delta

    #determines the value of the sigmoid coefficient for the gradient decent logistic
        red↪ regression model
    def sigmoid(self, a):
        a = np.clip(a, -500, 500)
        return 1 / (1 + math.exp(-a))
    #END sigmoid

    #calcualtes the learning rate based on the learning_rate multiplier and the value of
        red↪ k (the current step in the folds iteration
    def calc_learning_rate(self, k):
        return (1 / (math.exp(k ** self.learning_rate)))
    #END learning_rate

    # predict classes (for each row passed in)
    def predict(self, rows, x, y, weights) :
        # list of predictions (n x 1)
        predictions = []
        # go through rows
        for row in rows:
            # get probability
            row_probability = self.find_probability(row, x, y, weights)
            # get label (based on probability)
            predicted_label = self.predict_label(row_probability)
            # add to list
            predictions.append(predicted_label)
        # return prediction vector
        return predictions
    #END predict


    # find probability (get estimated value)
```

```
def find_probability(self, row, x, y, weights):
    '''
    # x transpose
    x_transpose = np.transpose(x);
    # X^T * X
    x_transpose_x = np.matmul(x_transpose, x)
    # (X^T * X)^-1
    inverse = np.linalg.inv(x_transpose_x)
    # X^T * Y
    x_transpose_y = np.matmul(x_transpose, y)
    # (X^T * X)^-1 * (X^T * Y)
    w_hat = np.matmul(inverse, x_transpose_y)
    # Xn * (X^T * X)^-1 * (X^T * Y)
    result = np.matmul(row, w_hat)
    '''
    result = np.matmul(row, weights)
    return result
#END find_probability


# classifies row based on probability
def predict_label(self, probability):
    return 0 if probability < 0 else 1
#END predict_label


def Accu_eval(self, predicted_labels, true_labels):
    # total values
    total = len(predicted_labels)
    # uneven arrays
    if (total != len(true_labels)):
        return -1
    # empty arrays
    if (total == 0):
        return 1
    else:
        correct_count = 0
    # calculate true error
    for i in range(total):
        if (predicted_labels[i] == true_labels[i]):
            correct_count += 1
    # define accuracy
    accuracy = correct_count / total
    # return rounded accuracy
    return round(accuracy * 100) / 100.0
#END Accu_eval

"""
Name:
    k_folds
Input:
    None, uses class variables calcualted during the previous functions
Output:
```

```
        Returns the average error generated during the k many folds of the gradient decent
            red↪  logistic regression
    Description:
        Finds the average probability for the model being generated
    """
    def k_folds(self):
        avg_error = 0
        for i in range(0, self.num_folds):
            # number of points in each fold
            num_points_in_fold = round(self.num_rows / self.num_folds)

            # counter for list indexing
            current_counter = 0
            training_counter = 0

            # lists to separate training and validation sets
            current_fold = [None] * num_points_in_fold
            training_folds = [None] * (self.num_rows - num_points_in_fold)
            training_labels = [None] * (self.num_rows - num_points_in_fold)

            # populate lists
            for j in range(0, self.num_rows):
                if (j >= i * num_points_in_fold) and (j < (i + 1) * num_points_in_fold):
                    current_fold[current_counter] = self.x[j]
                    current_counter += 1
                else:
                    training_folds[training_counter] = self.x[j]
                    training_labels[training_counter] = self.y[j]
                    training_counter += 1
                #END IF
            #END FOR

            # train logistic regression model
            weights = self.fit(training_folds, training_labels)

            # predict using logistic regression model
            predicted_labels = self.predict(current_fold, training_folds, training_labels,
                red↪  weights)

            # sum error values (for average error)
            true_labels = self.y[i * num_points_in_fold : (i + 1) * num_points_in_fold]
            avg_error += self.Accu_eval(predicted_labels, true_labels)

            # debug
            print(' ----> Fold #', i)
        # END for
        return (avg_error / self.num_folds)
    #END k_folds
#END logistic_classifier

#set of parameters to start folding the two datasets
def execute(model):
    # for finding best values / mean accuracy
    maximum = index = 0
```

```
    # go through both models
    for index, model in enumerate(models):
        # debug
        print('Start of model ', 'Bankruptcy' if index == 0 else 'Hepatitis')

        # parse out data / hyperparameters
        (file, episilon, learning_rate, max_iter) = model

        # initialize model
        classifier = logistic_classifier(file, episilon, learning_rate, max_iter, index)

        # number of times to calculate accuracy
        iterations = 2

        # go through each iteration
        for i in range(1, iterations):
            # run k-folds
            value = classifier.k_folds()

            # save value for best fold
            if (maximum < value):
                index = i
                maximum = value

            # debug
            print(' --> Fold Accuracy: ', value)
            #input()
#END execute

#MAIN PROGRAM ENTRY POINT
models = ( #this contains information for the two datasets
    # file, episilon, learning_rate, max_iter
    (bankruptcy_data, 0.001, 0.001, 5000),
    (hepatitis_data, 0.00000001, 0.31, 5000)
)

#prepare to start folding
execute(models)
```

7.2 Appendix B – Initial Program (Program Prior to Feature Reduction)

```
#============================================================================================
# Code used for generating models without dropping irrelevant training datasets:
# Generates values of 86.6% and 75.3% for the hepatitis and bankruptcy datasets
    red↪ respectively
#============================================================================================
# required libraries
import pandas as pd, numpy as np, math, random
import threading
```

```
# read in datasets ("file", epsilon)
bankruptcy_data = pd.read_csv('bankrupcy.csv')
hepatitis_data = pd.read_csv('hepatitis.csv')

class logistic_classifier :
    def __init__(self, file, epsilon, learning_rate, iter, type):
        self.dataType = type
        (x, y) = self.parse_matrices(file)
        self.x = x # X matrix (rows of data set)
        self.y = y # Y matrix (class labels)
        self.epsilon = epsilon # small convergence value
        self.learning_rate = learning_rate # learning rate (alpha)
        self.num_rows = len(x)
        self.num_features = len(x[0]) # m (number of features in x)
        self.num_folds = 10 # number of folds
        self.max_iter = iter
    #END __init__


    def parse_matrices(self, file):
        class_identifier = 'ClassLabel'
        bias_identifier = 'BiasTerm'

        # debug
        #print('dataset\n')
        #print(file, self.debug())

        # define X, Y, and row matrices
        y = np.longdouble(file[class_identifier]).reshape(-1, 1)
        x = file.drop(labels=class_identifier, axis="columns")

        # add dummy values
        x[bias_identifier] = 1


        # shuffle the DataFrame rows
        #x = x.sample(frac = 1)

        # convert x to numpy object
        x = x.to_numpy()

        return (x, y)
    #END parse_matrices

    def debug(self):
        return '\n------------------------------'
    #END debug


    def fit(self, x, y):
        # initialize descent varaibles
        k = 1
        norm_vector = self.epsilon
```

```
        first_pass = True

        # weight placeholders
        #current_weight = np.random.normal(0, 0.01, self.num_features)
        current_weight = np.ones(self.num_features)

        new_weight = None

        # loop until convergence
        print('Fold_', end='')
        #input()
        while ((first_pass or norm_vector >= self.epsilon) and k <= self.max_iter):

            # "do" while condition
            if (first_pass):
                first_pass = False
            #  (delta)
            delta_value = self.delta(x, y, current_weight)
            # k (learning rate)
            alpha = self.calc_learning_rate(k)
            # set new weight value
            new_weight = np.subtract(current_weight, np.multiply(alpha, delta_value))
            # change in descent
            descent_change = np.subtract(new_weight, current_weight)
            # calculate norm (for checking stop condition)
            norm_vector = np.linalg.norm(descent_change, ord=2) ** 2
            # increment k
            k += 1
            # set current to newer value (for next iteration)
            #input(new_weight)
            current_weight = new_weight
            #print(k,'---------',norm_vector)
            if k > self.max_iter :
                print('max_iter', end='')
        #END WHILE
        return current_weight;
    #END gradient_descent

    def delta(self, x, y, w):
        # matrix of updated weights
        sums = 0

        # go through rows of X/Y
        for i in range(1, len(x)):
            # wT
            w_transpose = np.transpose(w);
            # wT * xi
            w_transpose_x = np.matmul(w_transpose, x[i])
            # yi - (wT * xi)
            true_value = y[i][0] - self.sigmoid(w_transpose_x)
            # xi * yi - (wT * xi)
            result = x[i] * true_value
            # add to sum
            sums = np.add(sums, result)
```

```
    # return -
    return -sums;
#END delta

def sigmoid(self, a):
    a = np.clip(a, -500, 500)
    return 1 / (1 + math.exp(-a));
#END sigmoid

def calc_learning_rate(self, k):
    if self.dataType == 0 :
        return 1/k**0.5705
    else :
        return (1/(math.exp(k**self.learning_rate)))
#END learning_rate

# predict classes (for each row passed in)
def predict(self, rows, x, y, weights) :
    # list of predictions (n x 1)
    predictions = []
    # go through rows
    for row in rows:
        # get probability
        row_probability = self.find_probability(row, x, y, weights)

        #print('row_prob: ', row_probability)
        # get label (based on probability)
        predicted_label = self.predict_label(row_probability)
        # add to list
        predictions.append(predicted_label);
    # return prediction vector
    return predictions
#END predict


# find probability (get estimated value)
def find_probability(self, row, x, y, weights):
    '''
    # x transpose
    x_transpose = np.transpose(x);
    # X^T * X
    x_transpose_x = np.matmul(x_transpose, x)
    # (X^T * X)^-1
    inverse = np.linalg.inv(x_transpose_x)
    # X^T * Y
    x_transpose_y = np.matmul(x_transpose, y)
    # (X^T * X)^-1 * (X^T * Y)
    w_hat = np.matmul(inverse, x_transpose_y)
    # Xn * (X^T * X)^-1 * (X^T * Y)
    result = np.matmul(row, w_hat)
    '''
    result = np.matmul(row, weights)
    return result
#END find_probability
```

```
# classifies row based on probability
def predict_label(self, probability):
    return 0 if probability < 0 else 1
#END predict_label


def Accu_eval(self, predicted_labels, true_labels):
    # total values
    total = len(predicted_labels)
    # uneven arrays
    if (total != len(true_labels)):
        return -1
    # empty arrays
    if (total == 0):
        return 1
    else:
        correct_count = 0
    # calculate true error
    for i in range(total):
        if (predicted_labels[i] == true_labels[i]):
            correct_count += 1
    # define accuracy
    accuracy = correct_count / total
    # return rounded accuracy
    return round(accuracy * 100) / 100.0
#END Accu_eval


def k_folds(self):
    avg_error = 0
    for i in range(0, self.num_folds):
        # number of points in each fold
        num_points_in_fold = round(self.num_rows / self.num_folds)

        # counter for list indexing
        current_counter = 0
        training_counter = 0

        # lists to separate training and validation sets
        current_fold = [None] * num_points_in_fold
        training_folds = [None] * (self.num_rows - num_points_in_fold)
        training_labels = [None] * (self.num_rows - num_points_in_fold)

        # populate lists
        for j in range(0, self.num_rows):
            if (j >= i * num_points_in_fold) and (j < (i + 1) * num_points_in_fold):
                current_fold[current_counter] = self.x[j]
                current_counter += 1
            else:
                training_folds[training_counter] = self.x[j]
                training_labels[training_counter] = self.y[j]
                training_counter += 1
```

```
            # train logistic regression model
            weights = self.fit(training_folds, training_labels)

            # predict using logistic regression model
            predicted_labels = self.predict(current_fold, training_folds, training_labels,
                red↪  weights)

            # sum error values (for average error)
            true_labels = self.y[i * num_points_in_fold : (i + 1) * num_points_in_fold]
            avg_error += self.Accu_eval(predicted_labels, true_labels)
        # END for
        # return mean error
        return (avg_error / self.num_folds)
    #END k_folds
#END logistic_classifier

# START EXECUTION
files = (
    (bankruptcy_data, 0.001, 0.01),
    (hepatitis_data, 0.00000001, 26/10000)
)
file = files[1]

factorHep = [1/100000000, 0.31, 6000]
classifier = logistic_classifier(file[0], factorHep[0], factorHep[1], factorHep[2], 1)
value = classifier.k_folds()
print(value)

file = files[0]

factorHep = [0.001, 0.5705, 3000]
classifier = logistic_classifier(file[0], factorHep[0], factorHep[1], factorHep[2], 0)
value = classifier.k_folds()
print(value)
"""
for i in range(10, 200, 10) :
  file = files[1]
  epsilon = factorHep[0] / (i/100)
  error = factorHep[1] / (i/100)
  stepsize = factorHep[2] / (i/100)
  classifier = logistic_classifier(file[0], epsilon, error, stepsize)
  value = classifier.k_folds()
  if value > maximum :
    maximum = value
    index = i
    print([maximum, i])
print([maximum, index])
"""

# END EXECUTION
```