COMP 4900 Assignment 1

Tess Landry & Darren Pierre

February 9, 2020

Abstract

The purpose of this assignment was to compare and contrast the approaches of Linear Discriminant Analysis (LDA) to Logistic Regression with Gradient Descent on two data sets. The testing datasets used were the Parkinsons dataset, which looked at voice recordings, and the Sonar dataset, which looked at sonar signal patterns. The datasets were divided further into two testing datasets: the full dataset and a dataset containing a subset of selected features. The logistic regression approach had extremely high average error on the Parksinsons dataset, likely due to the imbalance healthy to unhealthy samples. The two models performed similarly on the sonar dataset, The LDA approach was faster and generally however the LDA model did so with a shorter runtime. The LDA model was overall a faster model, and generally achieved better (or at least equal) accuracy to the logistic regression approach.

Introduction

The purpose of this project was to compare two models: linear discriminant analysis (LDA) and logistic regression, using a k-fold cross validation with each model. These models were tested on two datasets: Parkinsons Voice Recordings and Sonar: Mines vs Rocks. The Parkinsons dataset contained 195 samples of voice recordings from healthy individuals and individuals diagnosed with Parkinsons disease. The Sonar dataset contained 208 sonar signal patterns from bouncing a signal off of rocks and metal cylinders. For the purpose of this assignment, the datasets were then further split into two options: the dataset containing all features as given, and the dataset with only a subset of selected features. These features were selected using L1 (Lasso) regression, and the performance of both models was measured and compared on both the entire datasets and the datasets with selected features.

For the whole Parkinsons dataset and the Parkinsons dataset with a subset of features, LDA performed significantly better than logistic regression. Including all the features as given, logistic regression performed slightly better than LDA on the Sonar dataset, while LDA performed slightly better on the subset of chosen features for the Sonar dataset. Overall, the LDA fit on the Parkinson's dataset had the best prediction accuracy. Overall, the LDA model also had a significantly faster runtime than the logistic regression model. The logistic regression fit on the Parkinson's dataset had the worst prediction accuracy, and thus the greatest error. The imbalance between the number of healthy and unhealthy individuals in the Parkinsons dataset is likely the cause of this extreme reduction in prediction accuracy for the logistic regression model.

Datasets

A full set of histograms of distributions of the features of each of the following datasets can be found when running the statistical analysis code.

Parkinsons Voice Recordings

The Parksinsons dataset contains data from 195 voice recordings, 48 of which are from healthy individuals and 147 of which are from individuals with Parkinsons disease. This dataset includes a total of 22 features which measure aspects such as frequency, amplitude, and tone of voice. Upon running a L1 (Lasso) regression on the data set with an alpha value of 0.01, half of the features were discarded. Although a smaller alpha would provide a smaller subset of features, no performance increase was seen with a further reduction in features. The 11 features that were chosen to be used for testing as a feature subset were: MDVP:Fo(Hz), MDVP:Fhi(Hz), MDVP:Flo(Hz), MDVP:Shimmer(dB), NHR, HNR, RPDE, DFA, spread1, spread2, and D2. Of this subset, the L1 regression gave the greatest weights to NHR, spread2, and RPDE.

Sonar: Mines vs Rocks

The Sonar dataset measured 208 sonar signal patterns, 111 of which are from bouncing a sonar signal off of metal cylinders (mines) and 97 of which are from bouncing a sonar signal off of rocks. This dataset has a total of 60 features (features 0-59) which each represent the results of transmitting the sonar signal in a given frequency band. A subset of 6 features was chosen for further testing using an L1 (Lasso) Regression. These features were (in order of weighting magnitude): Feature 44, Feature 10, Feature 35, Feature 20, Feature 11, and Feature 16.

Results

Logistic Regression Performance and Learning Rates

Running a 10 fold validation using a logistic regression model on both datasets, we can see the following trends:

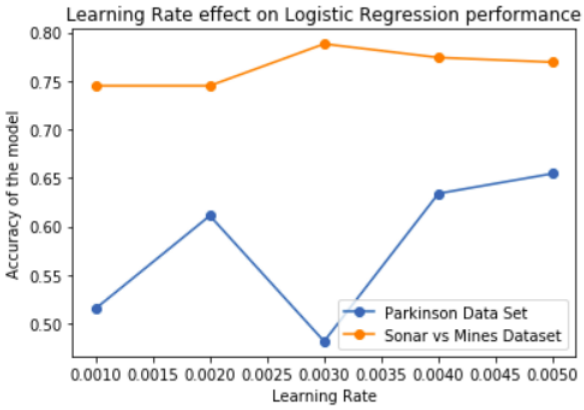

Figure 1: Learning Rates vs Accuracy of Logistic Regression

Specifically in the Parkinsons dataset, we can see that there generally tends to be an increase in accuracy as learning rate increases. This is seen to a lesser extent with the Sonar dataset. Going beyond rates depicted in this graph, this trend does not continue indefinitely. The performance of logistic regression will start to diminish once learning rates get too large (for example, a rate of 0.5). The learning rate chosen for the Parkinsons dataset was 0.1, while the best learning rate chosen for the Sonar dataset was 0.01.

Runtime of LDA and Logistic Regression

The runtime of 10 fold validation on the LDA model was significantly faster than the runtime of 10 fold validation on logistic regression with gradient descent, as seen in the table below.

	LDA	Logistic Regression
Parkinsons	0.456 s	31.7 s
Sonar	0.495 s	34.4 s

Table 1: Runtimes of both models on both data sets

Prediction Accuracy of LDA and Logistic Regression

For each dataset, a 10-fold validation was run and the prediction accuracy was computed for each fold. The accuracy was then averaged across all of the 10 folds to compute the final accuracy of the experiment.

Parkinsons Voice Recordings

The accuracy of five separate runs of the 10-fold validation on the Parkinsons dataset can be seen below for both LDA (a) and logistic regression (b) below:

All Features	Select Features
86.68%	84.05%
87.18%	84.13%
87.71%	85.63%
87.76%	85.61%
88.21%	83.58%

(a) Parkinsons Average Accuracy for LDA 10-Fold Validations

All Features	Select Features
65.16%	50.03%
70.13%	57.24%
46.00%	64.79%
63.08%	61.74%
48.11%	72.76%

(b) Parkinsons Average Accuracy for LR 10-Fold Validations

From this we can observe that the LDA prediction accuracy on the full Parkinsons dataset is $\approx 87.5\%$ (average error $\approx 12.5\%$), while the average accuracy on the subset of Parkinsons features is $\approx 84.5\%$ (average error $\approx 15.5\%$). Using all features of the Parkinsons dataset was therefore more beneficial when using LDA than using a subset of features. The logistic regression prediction accuracy on the full Parkinsons dataset is $\approx 58.5\%$ (average error $\approx 41.5\%$) while the average accuracy on the subset of Parkinsons features is $\approx 61.5\%$ (average error $\approx 38.5\%$). In the case of logistic regression, using the selected subset of features provided slightly better prediction accuracy than using the full dataset.

Comparing the two prediction models, LDA performed significantly better than logistic regression on both the full dataset and the subset of features. Looking at the confusion matrices generated in the code for both models gives some insight as to why this may be the case. In the LDA analysis, there did not seem to be a significant difference in false positive rate and false negative rate over the folds. In contrast, the logistic regression model had quite a few cases where it would either have 0 false negatives and 0 true negatives, or 0 false positives and 0 true positives. In these cases, the model was deciding that all of the validation data was in a single class. This could be explained by the discrepancy between the number of healthy (class 0) samples and Parkinson (class 1) samples. Only $\approx 25\%$ of the samples in this dataset were healthy individuals, meaning that most of the data used to fit and test would be from class 1. While there would be cases where the data used to fit was a fairly even split between class 0 and class 1, this overall imbalance means that it is likely that a lot of the fits used mostly, or even entirely, class 1 data.

Sonar: Mines vs Rocks

The accuracy of five separate runs of the 10-fold validation on the Sonar dataset can be seen below for both LDA (a) and logistic regression (b) below:

All Features	Select Features
72.07%	77.95%
74.95%	78.86%
75.00%	77.43%
78.93%	76.95%
74.52%	77.38%

(a) Sonar Average Accuracy for LDA 10-Fold Validations

All Features	Select Features
78.83%	72.93%
75.55%	71.67%
78.43%	73.52%
79.29%	72.14%
73.50%	72.55%

(b) Sonar Average Accuracy for LR 10-Fold Validations

On the Sonar dataset, the LDA prediction accuracy for the full dataset is $\approx 75\%$ (average error $\approx 25\%$), while the prediction accuracy for the selected subset of features is $\approx 78\%$ (average error $\approx 22\%$). As such, the selected subset of features had a smaller average prediction error compared to the full dataset on the LDA model. The logistic regression prediction accuracy on the full dataset is $\approx 77\%$ (average error $\approx 23\%$), while the prediction accuracy on the subset of features is $\approx 72.5\%$ (average error $\approx 27.5\%$). In contrast to the LDA model, the logistic regression model performed better on the full sonar dataset compared to the selected features.

On the full dataset, the logistic regression model performed slightly better, while on the subset of chosen features, LDA performed better. Unlike with the Parkinsons dataset, the number of false positives and false negatives was fairly balanced and minimal for both models. This can be explained by the fact that the Sonar dataset had a much more even split of samples from class 0 (rocks) and class 1 (mines) than the Parkinson's dataset.

Discussion & Conclusions

The best overall performance was seen by the LDA model on the Parkinsons dataset. Choosing a subset of features helped to improve the prediction accuracy of logistic regression, however it was still far less accurate than LDA. The two models performed similarly on the sonar dataset, with logistic regression performing better on the full dataset and LDA performing better with a subset of chosen features. Given these performance metrics and the fact that the LDA model had a much faster run time, it can be concluded that LDA was the superior model.

An additional key take away was the importance of class representation in the training data. As discussed in lecture, a lot of real world data is imbalanced and has more of one class than the other. For a model to be able to provide an accurate fit, it needs sufficient data from both classes to understand the differences between the classes and decide how to classify test data. The discrepancy in representation from both classes had a much a larger effect on the success of the logistic regression compared to feature selection. One option to fix this would have been dropping some of the class 1 data from the Parkinson's dataset, however this would provide significantly less overall data for the model to train and validate. The LDA model, on the other hand, did not seem to be as affected by the uneven split between unhealthy and healthy individuals.

There are a few items that could be considered for future investigation. For instance, the LDA model did extremely well on the unbalanced Parkinsons dataset, while doing slightly worse on the more balanced Sonar dataset. As such, trials could be done to see if this trend continues or if it was an unrelated factor that contributed to this difference in accuracy. Furthermore, the logistic regression performed similarly to LDA on the more balanced dataset. Future investigation could look at this comparison across multiple balanced datasets and multiple different feature subsets to see if the performance of logistic regression could be improved to be superior to that of LDA. Currently, it simply makes more sense to use LDA due to the benefit in both time and accuracy. This future investigation would therefore attempt to justify the use of logistic regression over LDA.

Statement of Contributions

Tess Landry (100926518) was responsible for the LDA class, KFoldValidation class, dataSetFormatter class, additions to MLMetric class for displaying data, and the written report. **Darren Pierre** (101015833) was responsible for the Logistic Regression class, MLMetric class, logistic regression additions to KFoldValidation class, colab notebook set-up, statistical analysis and graphical representations of datasets, and the discussion of logistic regression learning rates in the written report.

References

Parkinsons Dataset: https://archive.ics.uci.edu/ml/datasets/Parkinsons
Sonar: Mines vs Rocks: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+%28Sonar%2C+Mines+vs.Rocks
Slides from class: Presented by Majid Komeili

Appendix

```
from google.colab import drive
drive.mount('/content/drive')
PARKINSON_DATA="/content/drive/My Drive/Colab Notebooks/Data Sets/parkinsons.data"
SONAR_DATA="/content/drive/My Drive/Colab Notebooks/Data Sets/sonar.all-data"
```

```
#Importing libaries that we need for the project
import os
import pandas as pd
import numpy as np
import sklearn.linear_model
!pip install ipython-autotime
%load_ext autotime
```

Dataset Set-Up, Formatting, and Feature Selection

```
parkinsonDF=pd.read_csv(PARKINSON_DATA)
sonarDF=pd.read_csv(SONAR_DATA)
numCols=61
names=[f"Feature {x}"for x in range(numCols)]
# Class we will try to classify will be a mine
sonarDF=pd.read_csv(SONAR_DATA,header=None , names=names)
convertValues=lambda a :1 if a == 'M' else 0
sonarDF[names[-1]]=[convertValues(label) for label in sonarDF[names[-1]]]

#GET RID OF UNWANTED PARKINSONS COLUMNS
parkinsonDF = parkinsonDF.drop('name', 1)


"""
FEATURE SELECTION FOR PARKINSONS
L1 (Lasso) Regression: Choose features not weighted 0
"""
parkinsonX = parkinsonDF.drop('status', 1)
parkinsonY = parkinsonDF['status']
parkCols = parkinsonX.columns.to_list()

parkL1 = sklearn.linear_model.Lasso(alpha=.001) #You can play around with this to get more or less
    features
parkL1.fit(parkinsonX, parkinsonY)

#Print all items that arent weighted 0- these are the selected features
print("Parkinsons Selected Features: \n")
parkFeats = [] #List for the selected features
i = 0
while i < len(parkL1.coef_):
   if(parkL1.coef_[i] != 0):
      print(str(parkCols[i]) + ": " + str(parkL1.coef_[i]))
      parkFeats.append(parkCols[i])
   i = i+1
print("\n\n")
parkFeats.append('status')

#Make a df of just the selected features:
```

```
featSelectPark = parkinsonDF[parkFeats]


"""
FEATURE SELECTION FOR SONAR
L1 (Lasso) Regression: Choose features not weighted 0
"""
sonarX = sonarDF.drop('Feature 60', 1)
sonarY = sonarDF['Feature 60']
sonarCols = sonarX.columns.to_list()

sonarL1 = sklearn.linear_model.Lasso(alpha=0.01) #Basic highest alpha to not give all 0s
sonarL1.fit(sonarX, sonarY)

#Print all items that arent weighted 0- these are the selected features
print("Sonar Selected Features: \n")
sonarFeats = [] #List for the selected features
i = 0
while i < len(sonarL1.coef_):
   if(sonarL1.coef_[i] != 0):
      print(str(sonarCols[i]) + ": " + str(sonarL1.coef_[i]))
      sonarFeats.append(sonarCols[i])
   i = i+1
print("\n")
sonarFeats.append('Feature 60')

#Make a df of just the selected features:
featSelectSonar = sonarDF[sonarFeats]

print('All features not shown above had weight = 0 and were not selected.')
```

```
"""
A class that takes in a pandas dataset and returns the X Matrix and Y Vector as numpy arrays
Input:
   dataSet: pandas dataSet to test (get rid of any unwanted columns BEFORE doing this)
   yName: column name of output (Y) variable
NOTE: Format data set BEFORE passing in (eg. remove columns you dont want such as 'name')
"""
class dataSetFormatter:
#Returns the numpy XMatrix
def xMatrix(self, dataSet, yName):
   return (dataSet.drop(yName, 1)).to_numpy()

def yMatrix(self, dataSet, yName):
   return dataSet[[yName]].to_numpy()


#MAKE A FORMATTER
myFormatter = dataSetFormatter()
```

Implementing Machine Learning Models

```
"""
A class to represent the performance of a Machine learning model
Input: Either a list or numpy arrays representing the predicted values and the actual values
"""
```

```
class MLMetric:
   def __init__(self,predictedValues,outputValues):
      self.predictedValues=predictedValues
      self.outputValues=outputValues

   def percentage(self,num):
      return format(num,".2%")

   #Evaluates the accuracy of the model, comparing predicted values with actual output values
   def Accu_eval(self):
      correctAns=0
      i = 0
      while i < len(self.predictedValues):
         if (self.predictedValues[i] == self.outputValues[i]):
            correctAns=correctAns +1
         i=i+1
      accuracy=correctAns/len(self.predictedValues)
      print(f"The model had {correctAns} correct out of {len(self.predictedValues)}. Resulted in
          accuracy of: {self.percentage(accuracy)} ")
      self.confusionMatrix() #Prints the confusion matrix data too
      return accuracy

   #Finds the confusion metrics
   def cMetrics(self):
      tp=fp=tn=fn=0 # values to represent true positive, false positive, true negative and false
          negative

      i = 0
      while i < len(self.predictedValues):
         if(self.outputValues[i]==1):
            if(self.predictedValues[i]==1):
               tp=tp+1
            else:
               fn=fn+1
         else:
            if(self.predictedValues[i]==1):
               fp=fp+1
            else:
               tn=tn+1
         i=i+1
      return [tp,fp,tn,fn]

   #Prints the confusion metrics nicely
   def confusionMatrix(self):
      confusionData = self.cMetrics()
      # True postive rate : When a model is given a 1 (a class that we're trying to classify) , how
          often it predicts yes (1)
      if((confusionData[0] + confusionData[3]) == 0): #Prevent division by 0
         TPR = 0
      else:
         TPR= confusionData[0] /(confusionData[0] + confusionData[3]) # TPR= TP/(TP + FN)
      # True negative Rate : When a model is given a 0 (a class that we're trying to classify) ,
          how often it predicts no (0)
      if((confusionData[2] + confusionData[1]) == 0):
         TNR = 0
      else:
         TNR= confusionData[2] /(confusionData[2] + confusionData[1]) # TNR= TN/(TN + FP)
      print("\nConfusion Matrix:")
      print("  True Positives: " + str(confusionData[0]))
```

```
        print("  False Positives: " + str(confusionData[1]))
        print("  True Negatives: " + str(confusionData[2]))
        print("  False Negatives: " + str(confusionData[3]))
        print(f" True Postive Rate: {self.percentage(TPR)}")
        print(f" True Negative Rate: {self.percentage(TNR)}")
        print("\n")
```

```
"""
A class that represents a Logistic Regression Model
Input:
   dataX: numpy array with that represent a training set without the classicfications
   dataX: numpy array with that represent a training set
   learning_rate: The amount that the weights are updated during training
   training_iterations: number of times we should train our model
   y_intercept : our w0 term
"""
class LogisticRegression():
   def __init__(self,dataX,dataY,learning_rate,training_iteration,y_intercept=0):
      numFeatures=dataX.shape[1] # getting the number of features
      self.weights= np.zeros(numFeatures,dtype=np.float128) # setting our weights to zero
      self.X=dataX
      self.Y=dataY
      self.y_intercept=y_intercept
      self.fit(learning_rate,training_iteration)

   def sigmoid(self,x):
      # 1/ 1 + e ^-x   where x =wTx + y_intercept
      return 1 / (1 + np.exp(-x,dtype=np.float128))

   def computeSigmoidInput(self,xi):
      # wTx + y_intercept
      return np.dot(self.weights.T,xi) + self.y_intercept


   def fit(self,learning_rate,training_iteration):
      for _ in range(training_iteration):
         gradient=0
         numInstances=self.X.shape[0]
         for i in range(numInstances):
            xi=np.array(self.X[i],dtype=np.float128 )# xi represents an instance in our dataset
            yi=np.array(self.Y[i],dtype=np.float128) # yi represents a label in our dataset
            sigmoidInput= self.computeSigmoidInput(xi)# z= (wT * xi) + y-intercept
            predicted_output= self.sigmoid(sigmoidInput) # sigmoid(z)
            error = yi-predicted_output # error=y- sigmoid(z)
            gradient = gradient + (xi*error) # SUM(xi * error)

            # here we update our weights
            self.weights = self.weights + (learning_rate * (gradient ))

   def predict(self,data):
      #Classification of a new observation ,
      output=[self.sigmoid(self.computeSigmoidInput(row)) for row in data] #sigmoid(wT * x_new +
          intercept)
      classify= lambda x: 1 if x >= 0.5 else 0
      predictedClasses=[classify(x) for x in output ]
      return predictedClasses
```

```
"""
A class to perform a Linear Discriminant Analysis
Input:
   dataSetX: numpy array of X values
   dataSetY: numpy array of Y values such that the result for x_i is y_i
"""

class LDA:
   def __init__(self, dataSetX, dataSetY):
      self.X = dataSetX
      self.Y = dataSetY

      self.N_0 = 0 #Number of entries in class 0
      self.N_1 = 0 #Number of entries in class 1
      self.total = 0 #Total Entries

      self.pY0 = 0 #P(Y=0)
      self.pY1 = 0 #P(Y=1)

      self.mean0 = [] #Mean of class 0
      self.mean1 = [] #Mean of class 1

      self.covariance = [] #Covariance Matrix
      self.covarInverse = [] #Inverse Covariance Matrix
      self.predictions = [] #Vector of Predictions for testData

      self.fit()
      #print("Class 0: " + str(self.N_0))
      #print("Class 1: " + str(self.N_1))

   def fit(self):
      #PROBABILITIES
      i=0
      while i < len(self.Y):
         if (self.Y[i] == 0):
            self.N_0 = self.N_0 + 1
         elif (self.Y[i] == 1):
            self.N_1 = self.N_1 + 1
         i=i+1

      self.total = len(self.Y) #len(self.Y) should be equal to N_0 + N_1, if not we have a problem
      self.pY0 = self.N_0 / self.total #P(y = 0) = N_0 / (N_0 + N_1) eg. if 20/50 samples are class
          0
      self.pY1 = self.N_1 / self.total #P(y = 1) = N_1 / (N_0 + N_1) eg. if 30/50 samples are class
          1

      #MEANS
      i=0
      while i < len(self.Y):
         if(self.Y[i] == 0):
            if(len(self.mean0) == 0):
               self.mean0 = self.X[i]
            else:
               self.mean0 = np.add(self.mean0, self.X[i])
         elif(self.Y[i] == 1):
            if(len(self.mean1) == 0):
               self.mean1 = self.X[i]
            else:
               self.mean1 = np.add(self.mean1, self.X[i])
```

```
        i=i+1

    self.mean0 = (np.true_divide(self.mean0, self.N_0)).reshape(-1,1) # mu_0 = (sum of i = 1 to i
        = n): I(y_i = 0) x_i/N_0, reshape so mx1
    self.mean1 = (np.true_divide(self.mean1, self.N_1)).reshape(-1,1) # mu_1 = (sum of i = 1 to i
        = n): I(y_i = 0) x_i/N_1, reshape so mx1

    #COVARIANCE MATRIX
    i=0
    while i < len(self.Y):
        currentX = (self.X[i]).reshape(-1,1) #Reshape so that it is mx1

        if(self.Y[i] == 0):
            xMinusMean = np.subtract(currentX, self.mean0)
        elif(self.Y[i] == 1):
            xMinusMean = np.subtract(currentX, self.mean1)

        covarNumerator = np.dot(xMinusMean, (xMinusMean.transpose()))
        covarDenominator = np.subtract(self.total, 2)
        covarTotal = np.true_divide(covarNumerator, covarDenominator)
        #on the first iteration, set it to first calculated matrix
        if(i == 0):
            self.covariance = covarTotal
        else:
            self.covariance = np.add(self.covariance, covarTotal)
        i=i+1
    self.covarInverse = np.linalg.inv(self.covariance)
    #print(self.covariance)

#Predict Function, takes data, predicts using the fit we calculated
def predict(self, testData):
    i=0
    while i < len(testData):
        #This is all just that big formula line by line
        currData = testData[i].reshape(-1,1) #reshape so mx1
        currData = currData.transpose()
        currData = np.dot(currData, self.covarInverse)


        #Class 0
        part1_0 = np.dot(currData, self.mean0)
        part2_0 = self.mean0.transpose()
        part2_0 = np.multiply(part2_0, 0.5)
        part2_0 = np.dot(part2_0, self.covarInverse)
        part2_0 = np.dot(part2_0, self.mean0)
        part3_0 = np.log(self.pY0)

        discrim0 = np.subtract(part1_0, part2_0)
        discrim0 = np.add(discrim0, part3_0)

        #Class 1
        part1_1 = np.dot(currData, self.mean1)
        part2_1 = self.mean1.transpose()
        part2_1 = np.multiply(part2_1, 0.5)
        part2_1 = np.dot(part2_1, self.covarInverse)
        part2_1 = np.dot(part2_1, self.mean1)
        part3_1 = np.log(self.pY1)

        discrim1 = np.subtract(part1_1, part2_1)
```

```
        discrim1 = np.add(discrim1, part3_1)

        #Output ArgMax of Class 0 and Class 1
        if(discrim1 > discrim0):
           if (len(self.predictions) == 0):
              self.predictions = np.array([1])
           else:
              self.predictions = np.append(self.predictions, 1)
        else:
           if (len(self.predictions) == 0):
              self.predictions = np.array([0])
           else:
              self.predictions = np.append(self.predictions, 0)

        i=i+1
     self.predictions = self.predictions.reshape(-1,1)
     return self.predictions
```

```
"""
KFold class that takes in a pandas dataset to perform the fold
Input:
   dataSet: pandas dataSet to test (get rid of any unwanted columns BEFORE doing this)
   k: number of folds to perform
   yName: name of y column, to be used when splitting data
"""

class KFoldValidation:
   def __init__(self,dataSet,k, yName):
      self.kSplitData = self.splitData(dataSet,k) #Data evenly (as much as possible) split into k
          partitions
      if(k < 2):
         self.k=2
      else:
         self.k=k
      self.yName = yName
      self.kFoldSets = [] #Set of k [trainSet, valSet] pairs. Perform LDA and Logistic regression
          on each one.

      #LDA SPECIFIC VARIABLES
      self.kFoldLDAs = [] #List of LDA Fits generated from the analysis. call
          kFoldLDAs[i].predict() to test any of your data on any of these
      self.accuracyLDA = 0

      #LR SPECIFIC VARIABLES
      self.kFoldLogs = list() #List of LR Fits generated from the analysis. call
          kFoldLogs[i].predict() to test any of your data on any of these
      self.accuracyLog = 0

      #Call these functions on init so that we already have our dataSets and LDA fits ready to go
      self.kFoldDataSet()
      self.kFold_LDA_fit()



   def splitData(self, dataSet, k):
      dataSet = dataSet.sample(frac=1).reset_index(drop=True)
      kSplitList = []
```

```
    #checking to see how much is remaining if we divide our dataset by k
    remainder= len(dataSet) % k

    #CREATE LIST OF k PARTITIONS OF THE DATA
    #NO REMAINDER
    if(remainder==0):
        i = 0
        start = 0
        sizeOfPartitions = int(len(dataSet) / k)
        stop = sizeOfPartitions
        while i < k:
            partition = dataSet[start:stop]
            kSplitList.append(partition)
            start = start + sizeOfPartitions
            stop = stop + sizeOfPartitions
            i=i+1

    #REMAINDER
    else:
        i = 0
        start = 0
        sizeOfPartitions = int(len(dataSet) // k)
        stop = sizeOfPartitions + 1
        while i < k:
            partition = dataSet[start:stop]
            kSplitList.append(partition)
            if(remainder > 0):          #REMAINDER > 0 means we have partition size + 1 partitions
                start = start + sizeOfPartitions + 1 #This round had partition + remainder so we
                    MUST increment start by this much
                remainder = remainder - 1
                if(remainder > 0):        #If still > 0, we want next next round to include remainder
                    stop = stop + sizeOfPartitions + 1
                else:                     #otherwise this was our last iteration with the remainder
                    and next round should have regular partitions
                    stop = stop + sizeOfPartitions
            else:                        #REMAINDER = 0 means we have normal sized partitions
                start = start + sizeOfPartitions
                stop = stop + sizeOfPartitions
            i=i+1

        return kSplitList


#Prepares a single dataSet consisting of [training set, validation set]
def prepareDataSet(self,indexOfValSet):
    valSet = self.kSplitData[indexOfValSet].reset_index(drop=True)
    trainSet = []
    #Combine all the other data into our training set
    i = 0
    while i < self.k:
        if(i == indexOfValSet):
            i = i+1
            continue
        if(len(trainSet) == 0): #no entries added
            trainSet = self.kSplitData[i]
        else: #we concatenate
            trainSet = pd.concat([trainSet, self.kSplitData[i]], axis=0).reset_index(drop=True)
        i = i+1
```

```
    return [trainSet, valSet]

#Prepares a k-length list of all [training set, validation set] pairs for the kfold
def kFoldDataSet(self):
    i=0
    while i < self.k:
        self.kFoldSets.append(self.prepareDataSet(i))
        i=i+1
    #self.kFoldSets[i][0] is ith training set
    #self.kFoldSets[i][1] is ith validation set


def kFold_LDA_fit(self):
    i = 0
    while i < self.k:
        #Use LDA to fit with the Training Set
        myLDA = LDA(myFormatter.xMatrix(self.kFoldSets[i][0], self.yName),
            myFormatter.yMatrix(self.kFoldSets[i][0], self.yName))
        #Add our fit to the list so that it can be tested with Test Data Later
        self.kFoldLDAs.append(myLDA)
    i=i+1

def kFold_LDA_predict(self):
    predictionAccuracy = 0
    i = 0
    while i < self.k:
        #Use our fit to predict with the Validation Set
        myLDAPredictions = self.kFoldLDAs[i].predict(myFormatter.xMatrix(self.kFoldSets[i][1],
            self.yName))

        #Accuracy Metrics on Parkinsons
        myMetric = MLMetric(myLDAPredictions, myFormatter.yMatrix(self.kFoldSets[i][1],
            self.yName))
        predictionAccuracy = predictionAccuracy + myMetric.Accu_eval()
        i = i+1

    self.accuracyLDA = (predictionAccuracy/self.k)
    print("LDA Average Accuracy for this fit is: " + str(format(self.accuracyLDA,".2%")))

    return self.accuracyLDA

def kFold_Log_fit(self,lr,itterations):
    i = 0
    while i < self.k:
        myLog = LogisticRegression(myFormatter.xMatrix(self.kFoldSets[i][0], self.yName),
            myFormatter.yMatrix(self.kFoldSets[i][0], self.yName),lr,itterations)
        #Add our fit to the list so that it can be tested with Test Data Later
        self.kFoldLogs.append(myLog)
        i=i+1

def kFold_Log_predict(self):
    predictionAccuracy = 0
    i = 0
    while i < self.k:
        #Use our fit to predict with the Validation Set
        myLDAPredictions = self.kFoldLogs[i].predict(myFormatter.xMatrix(self.kFoldSets[i][1],
            self.yName))

        #Accuracy Metrics on Parkinsons
```

```
        myMetric = MLMetric(myLDAPredictions, myFormatter.yMatrix(self.kFoldSets[i][1],
            self.yName))
        print("Summary for Logistic Regression:")
        predictionAccuracy = predictionAccuracy + myMetric.Accu_eval()
        i = i+1

      self.accuracyLog = (predictionAccuracy/self.k)
      print("Logistic Regression Average Accuracy for this fit is: " +
          str(format(self.accuracyLog,".2%")))

      return self.accuracyLog
```

Getting Learning Rate for Logistic Regression

```
from matplotlib import pyplot as plt
def lineGraph(x,y,xlabel,title):
   labels =["Parkinson Data Set","Sonar vs Mines Dataset"]
   f, ax = plt.subplots()
   ax.plot(x, y[0] ,linestyle='-', label=labels[0],marker='o')
   ax.plot(x, y[1] ,linestyle='-', label=labels[1],marker='o')

   ax.set_xlabel(xlabel)
   ax.set_ylabel("Accuracy of the model")
   ax.set_title(title)
   plt.legend()
   plt.show()

def testRun(learningRate,itterations=1400):
   parkinsonsKFold = KFoldValidation(parkinsonDF, 10, 'status')
   parkinsonsKFold.kFold_Log_fit(learningRate,itterations)
   parkAvgAccuracy=parkinsonsKFold.kFold_Log_predict()
   sonarKFold = KFoldValidation(sonarDF, 10, 'Feature 60')
   sonarKFold.kFold_Log_fit(learningRate,itterations)
   sonarAvgAccuracy = sonarKFold.kFold_Log_predict()
   return parkAvgAccuracy,sonarAvgAccuracy

def learningRateTest(learningRates):
   graphTitle="Learning Rate effect on Logistic Regression performance"
   xLabel="Learning Rate"
   itterations=1400
   pScores=list()
   sScores=list()
   for lr in learningRates:
      parkinsonAccuracy,sonarAccuracy=testRun(lr,itterations)
      pScores.append(parkinsonAccuracy)
      sScores.append(sonarAccuracy)
   lineGraph(learningRates,(pScores,sScores),xLabel,graphTitle)

learningRates=[0.001,0.01,0.1]
learningRates2=[x* 0.0001 for x in range (1,11) ]
learningRates3=[x* 0.005 for x in range(1,6) ]
#learningRateTest(learningRates)


def itterationsTest(itterations):
   graphTitle="Number of itterations effect on Logistic Regression performance"
   xLabel="Number of itterations"
```

```
   learningRate=0.005
   pScores=list()
   sScores=list()
   for i in itterations:
      parkinsonAccuracy,sonarAccuracy=testRun(learningRate,i)
      pScores.append(parkinsonAccuracy)
      sScores.append(sonarAccuracy)
   lineGraph(itterations,(pScores,sScores),xLabel,graphTitle)
   print(f"Parkinsion Average KFold Scores:{pScores}")
   print(f"Scores Average KFold Scores:{sScores}")

itterations=[100,250,500,900,1400]
accuracyTest=[500 for x in range(5)]
```

Running Our Algorithms

```
"""
LDA Run of PARKINSONS
"""

print("---------------")
print("PARKINSONS DATA")
print("---------------\n")

#ALL FEATURES
parkAllAcc = []
i=0
while i<5:
   parkinsonsKFold = KFoldValidation(parkinsonDF, 10, 'status') #K fold with all features
   parkinsonsAvgAcc = parkinsonsKFold.kFold_LDA_predict()
   parkAllAcc.append(format(parkinsonsAvgAcc,".2%"))
   i=i+1

#SELECT FEATURES
parkSelectAcc = []
i=0
while i<5:
   parkinsonsKFoldFeat = KFoldValidation(featSelectPark, 10, 'status') #K fold with selected
       features
   parkinsonsFeatAvgAcc = parkinsonsKFoldFeat.kFold_LDA_predict()
   parkSelectAcc.append(format(parkinsonsFeatAvgAcc, ".2%"))
   i=i+1
```

```
"""
LDA Run of SONAR
"""

print("\n\n----------")
print("SONAR DATA")
print("----------\n")

sonarAllAcc = []
i=0
while i<5:
   sonarKFold = KFoldValidation(sonarDF, 10, 'Feature 60') #K fold with all features
   sonarAvgAcc = sonarKFold.kFold_LDA_predict()
```

```
    sonarAllAcc.append(format(sonarAvgAcc, ".2%"))
    i=i+1

sonarSelectAcc = []
i=0
while i<5:
    sonarKFoldFeat = KFoldValidation(featSelectSonar, 10, 'Feature 60') #K fold with selected
        features
    sonarFeatAvgAcc = sonarKFoldFeat.kFold_LDA_predict()
    sonarSelectAcc.append(format(sonarFeatAvgAcc, ".2%"))
    i=i+1
```

```
"""
Logistic regression Run of PARKINSONS
I pick thoses specific hyperparameters because through some experimentation I found to deliver the
    best performance
"""
startTestString="""
-----------------------------------------------------------------------------------------------
Logistic regression test run for the PARKINSONS Data set (entire and selected feature Dataset)
-----------------------------------------------------------------------------------------------
"""
print(startTestString)

experimentLearningRatePark=0.1
experimentNumItterations=140
#ALL FEATURES
parkAllAccLog = []
parkSelectAccLog= []
for _ in range(5):
    parkinsonsKFold = KFoldValidation(parkinsonDF, 10, 'status') #K fold with all features
    parkinsonsKFold.kFold_Log_fit(experimentLearningRatePark,experimentNumItterations)
    parkinsonsAvgAccLog = parkinsonsKFold.kFold_Log_predict()
    parkAllAccLog.append(format(parkinsonsAvgAccLog,".2%"))
#SELECT FEATURES
for _ in range(5):
    parkinsonsKFoldFeat = KFoldValidation(featSelectPark, 10, 'status') #K fold with selected
        features
    parkinsonsKFoldFeat.kFold_Log_fit(experimentLearningRatePark,experimentNumItterations)
    parkinsonsFeatAvgAccLog = parkinsonsKFoldFeat.kFold_Log_predict()
    parkSelectAccLog.append(format(parkinsonsFeatAvgAccLog, ".2%"))
```

```
"""
Logistic regression Run of Sonar
I pick thoses specific hyperparameters because through some experimentation I found to deliver the
    best performance
"""
startTestString="""
-----------------------------------------------------------------------------------------------
Logistic regression test run for the Sonar Data set (entire and selected feature Dataset)
-----------------------------------------------------------------------------------------------
"""
print(startTestString)

experimentLearningRate=0.01
experimentNumItterations=1400
#ALL FEATURES
```

```
sonarAllAccLog = []
sonarSelectAccLog = []
for _ in range(5):
   sonarKFold = KFoldValidation(sonarDF, 10, 'Feature 60') #K fold with all features
   sonarKFold.kFold_Log_fit(experimentLearningRate,experimentNumItterations)
   sonarAllAvgAccLog=sonarKFold.kFold_Log_predict()
   sonarAllAccLog.append(format(sonarAllAvgAccLog,".2%"))
#SELECT FEATURES
for _ in range(5):
   sonarKFoldFeat = KFoldValidation(featSelectSonar, 10, 'Feature 60') #K fold with selected
       features
   sonarKFoldFeat.kFold_Log_fit(experimentLearningRate,experimentNumItterations)
   sonarFeatAvgAccLog = sonarKFoldFeat.kFold_Log_predict()
   sonarSelectAccLog.append(format(sonarFeatAvgAccLog, ".2%"))
```

Results

```
"""
RESULTS OF LDA
- 5 runs of a 10-Fold LDA Each
"""

print("Average Accuricies for 5 runs of a 10-Fold Cross Validation Analysis of LDA\n")
print("Parkinsons (All Features): " + str(parkAllAcc))
print("Parkinsons (Select Features): " + str(parkSelectAcc))
print("Sonar (All Features):      " + str(sonarAllAcc))
print("Sonar (Select Features):   " + str(sonarSelectAcc))


logResults=f"""
Average Accuricies for 5 runs of a 10-Fold Cross Validation Analysis of Logistic Regression

Parkinsons (All Features): {parkAllAccLog}
Parkinsons (Select Features): {parkSelectAccLog}
Sonar (All Features):      {sonarAllAccLog}
Sonar (Select Features):   {sonarSelectAccLog}
"""
print(logResults)
```

Statistical Analysis

Parkinsons

```
parkinsonDF=pd.read_csv(PARKINSON_DATA)
parkinsonDF.describe()
```

```
headers=[x for x in list(parkinsonDF) if x!='name' and x!='status']
# Pie chart
import matplotlib.pyplot as plt
numOfHealthy=parkinsonDF[(parkinsonDF.status==0)].status.count()
numOfPD=parkinsonDF[(parkinsonDF.status==1)].status.count()
labels = ['Healthy', 'Parkinson Disease']
values = [numOfHealthy,numOfPD]
```

```
fig, ax = plt.subplots(1,2,False)

ax[0].pie(values, labels=labels, autopct='%1.1f%%',
shadow=True, startangle=90)
# bar chart
ax[1].bar(labels,values)
ax[1].set_xlabel("Classes")
ax[1].set_ylabel("Number of people in a class")
title="Distribution among 2 classes"
ax[0].set_title(title)
ax[1].set_title(title)
fig.tight_layout(pad=0, w_pad=0, h_pad=0)
fig.subplots_adjust(left=0.3, right=1.5, wspace=0.9)

plt.show()


for header in headers:
   plt.hist(parkinsonDF[header].tolist(),label=header)
   plt.title(f"Distribution of {header}")
   plt.show()
```

```
from statistics import mean

healthyDF=parkinsonDF[parkinsonDF["status"]==0]["MDVP:Fo(Hz)"].tolist()
sickDF=parkinsonDF[parkinsonDF["status"]==1]["MDVP:Fo(Hz)"].tolist()
avgFreq=[mean(healthyDF),mean(sickDF)]
plt.bar(labels,avgFreq)
plt.suptitle('Comparison between Heathy people and non-heatlhy people based on average vocal
    frequency ')
plt.show()


def barchart(x,y,name):
   avgFreq=[mean(healthyDF),mean(sickDF)]
   plt.bar(labels,avgFreq)
   plt.suptitle(f'Comparison between Heathy people and non-heatlhy people based on {name} ')
   plt.show()

"""
headers=[x for x in list(parkinsonDF) if x!='name' and x!="spread1"]
for header in headers:
   healthyDF=parkinsonDF[parkinsonDF["status"]==0][header].tolist()
   sickDF=parkinsonDF[parkinsonDF["status"]==1][header].tolist()
   barchart(healthyDF,sickDF,header)
"""
```

```
healthyPpl=parkinsonDF[(parkinsonDF.status==1)]
parkinsonPpl=parkinsonDF[(parkinsonDF.status==0)]

for header in headers:
   plt.hist(healthyPpl[header].tolist(),label="Healthy People")
   plt.hist(parkinsonPpl[header].tolist(),label="People diagnose with Parkinsion")
   plt.legend()
   plt.title(f"Distribution of {header}")

   plt.show()
```

Sonar: Mines vs Rocks

```
numCols=61
names=[f"Feature {x}"for x in range(numCols)]
# Class we will try to classify will be a mine
sonarDF=pd.read_csv(SONAR_DATA,header=None , names=names)
convertValues=lambda a :1 if a == 'M' else 0
sonarDF[names[-1]]=[convertValues(label) for label in sonarDF[names[-1]]]
sonarDF.describe()
```

```
# Pie chart
import matplotlib.pyplot as plt
rocks=sonarDF[(sonarDF[names[-1]]==0)][names[-1]].count()
mines=sonarDF[(sonarDF[names[-1]]==1)][names[-1]].count()
labels = ['Rocks', 'Mines']
values = [rocks,mines]
fig, ax = plt.subplots(1,2,False)

ax[0].pie(values, labels=labels, autopct='%1.1f%%',
shadow=True, startangle=90)
# bar chart
ax[1].bar(labels,values)
ax[1].set_xlabel("Classes")
ax[1].set_ylabel("Number of objects in a class")
title="Distribution among 2 classes"
ax[0].set_title(title)
ax[1].set_title(title)
fig.tight_layout(pad=0, w_pad=0, h_pad=0)
fig.subplots_adjust(left=0.3, right=1.5, wspace=0.9)

plt.show()
```

```
headers=[x for x in list(sonarDF) if x!='name' and x!=names[-1]]

for header in headers:
   plt.hist(sonarDF[header].tolist(),label=header)
   print(f"Distribution of {header}")
   plt.show()
```

```
minesDF=sonarDF[(sonarDF[names[-1]]==1)]
rocksDF=sonarDF[(sonarDF[names[-1]]==0)]

for header in headers:
   plt.hist(minesDF[header].tolist(),label="Mines")
   plt.hist(rocksDF[header].tolist(),label="Rocks")
   plt.legend()
   plt.title(f"Distribution of {header}")

   plt.show()
```