**Assignment Four**

**CS 499**

Richard McCormick (RLM443)

**Python Program:**

# <-- BEGIN IMPORTS / HEADERS -->

import os

import urllib

import urllib.request

import pandas as pd

import numpy as np

import plotnine as p9

import sklearn

from sklearn.model\_selection import KFold

from sklearn.model\_selection import GridSearchCV

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from statistics import mode

import inspect

import warnings

# <-- END IMPORTS / HEADERS -->

# <-- BEGIN INITIALIZATION -->

# FILE VARIABLES

download\_directory = "."

# - Spam data variables

spam\_data\_url = "https://hastie.su.domains/ElemStatLearn/datasets/spam.data"

spam\_data\_file = "spam.data"

spam\_file\_path = os.path.join(download\_directory, spam\_data\_file)

# - Zip data (Training) variables

ziptrain\_url = "https://hastie.su.domains/ElemStatLearn/datasets/zip.train.gz"

ziptrain\_file = "zip.train.gz"

ziptrain\_file\_path = os.path.join(download\_directory, ziptrain\_file)

# - Zip data (Test) variables

ziptest\_url = "https://hastie.su.domains/ElemStatLearn/datasets/zip.test.gz"

ziptest\_file = "zip.test.gz"

ziptest\_file\_path = os.path.join(download\_directory, ziptest\_file)

# CONSTANT VARIABLES

spam\_label\_col = 57

zip\_empty\_col = 257

MyKNN\_N\_NEIGHBORS\_VAL = 20

CV\_VAL = 5

# MISC. VARIABLES

kf = KFold(n\_splits=3, shuffle=True, random\_state=1)

test\_acc\_df\_list = []

pipe = make\_pipeline(StandardScaler(), LogisticRegression(max\_iter=1000))

#CLASS DEFINITIONS

class MyCV():

    def \_\_init\_\_(self, \*\*kwargs):

        # Initialize parameters and setup variables

        self.train\_features = []

        self.train\_labels = []

        self.training\_data = None

        kwargs.setdefault("num\_folds", 5)

        for key, value in kwargs.items():

            setattr(self, key, value)

        self.estimator = self.estimator()

        self.best\_model = None

        self.plotting\_df = pd.DataFrame()

    def fit(self, X, y):

        # Populate internal data structures

        self.train\_features = X

        self.train\_labels = y

        self.training\_data = {'X':self.train\_features, 'y':self.train\_labels}

        # Create a dataframe to temporarily hold results from each fold

        best\_paramter\_df = pd.DataFrame()

        # Calculate folds

        fold\_indicies = []

        # Pick random entries for validation/subtrain

        fold\_vec = np.random.randint(low=0,

                                     high=self.num\_folds,

                                     size=self.train\_labels.size)

        # for each fold,

        for fold\_number in range(self.num\_folds):

            subtrain\_indicies = []

            validation\_indicies = []

            # check if index goes into subtrain or validation list

            for index in range(len(self.train\_features)):

                if fold\_vec[index] == fold\_number:

                    validation\_indicies.append(index)

                else:

                    subtrain\_indicies.append(index)

            fold\_indicies.append([subtrain\_indicies, validation\_indicies])

        printing\_df = pd.DataFrame()

        # Loop over the folds

        for foldnum, indicies in enumerate(fold\_indicies):

            print("(MyCV) Subfold #" + str(foldnum))

            # Get indicies of data chosen for this fold

            index\_dict = dict(zip(["subtrain", "validation"], indicies))

            set\_data\_dict = {}

            # Dictionary for test and train data

            for set\_name, index\_vec in index\_dict.items():

                set\_data\_dict[set\_name] = {

                    "X":self.train\_features[index\_vec],

                    "y":self.train\_labels.iloc[index\_vec].reset\_index(drop=True)

                }

            # Create a dictionary to hold the results of the fitting

            results\_dict = {}

            parameter\_index = 0

            # Loop over each parameter in the param\_grid

            for parameter\_entry in self.param\_grid:

                for param\_name, param\_value in parameter\_entry.items():

                    setattr(self.estimator, param\_name, param\_value)

                # Fit fold data to estimator

                self.estimator.fit(\*\*set\_data\_dict["subtrain"])

#                printing\_df = printing\_df.append({'loss': self.estimator.avg\_loss, 'iterations': self.estimator.max\_iterations, 'step\_size': self.estimator.step\_size, 'fold':foldnum}, ignore\_index=True)

                # Make a prediction of current fold's test data

                prediction = \

                    self.estimator.predict(set\_data\_dict["validation"]['X'])

                # Determine accuracy of the prediction

                results\_dict[parameter\_index] = \

                (prediction == set\_data\_dict["validation"]["y"]).mean()\*100

                # index only serves to act as key for results dictionary

                parameter\_index += 1

            # Store the results of this param entry into dataframe

            best\_paramter\_df = best\_paramter\_df.append(results\_dict,

                                                       ignore\_index=True)

        # all of this stuff is for plotting loss vs iterations...

#        printing\_df = printing\_df.groupby(['step\_size', 'iterations']).loss.apply(list)

#        printing\_df = printing\_df.to\_frame().reset\_index()

#        printing\_df['iteration\_list'] = ""

#        for index, row in printing\_df.iterrows():

#            new\_loss\_row = row['loss']

#            new\_loss\_row = np.mean(new\_loss\_row, axis=0)

#            printing\_df.at[index, 'loss'] = new\_loss\_row

#            new\_iter\_row = row['iterations']

#            new\_iter\_row = np.arange(new\_iter\_row)

#            printing\_df.at[index, 'iteration\_list'] = new\_iter\_row

#

#        printing\_df = printing\_df.explode(['loss', 'iteration\_list'])

        # Average across all folds for each parameter

        averaged\_results = dict(best\_paramter\_df.mean())

        # From the averaged data, get the single best model

        best\_result = max(averaged\_results, key = averaged\_results.get)

        # Store best model for future reference

        self.best\_model = self.param\_grid[best\_result]

    def predict(self, test\_features):

        # Load best model into estimator

        for param\_name, param\_value in self.best\_model.items():

            setattr(self.estimator, param\_name, param\_value)

        # Fit estimator to training data

        self.estimator.fit(\*\*self.training\_data)

        # Make a prediction of the test features

        prediction = self.estimator.predict(test\_features)

        return(prediction)

class MyLogReg():

    def \_\_init\_\_(self, \*\*kwargs):

        kwargs.setdefault("num\_folds", 5)

        kwargs.setdefault("max\_iterations", 10) # trained through cv

        kwargs.setdefault("step\_size", 0.0001) # trained through cv

        self.train\_data = None

        self.train\_labels = None

        self.coef\_ = None

        self.intercept\_ = None

        self.plotting\_df = {}

        #self.pipe = \

        #    make\_pipeline(StandardScaler(), LogisticRegression(max\_iter=1000))

        for key, value in kwargs.items():

            setattr(self, key, value)

    def fit(self, X, y):

        self.train\_data = X

        self.train\_labels = y

        self.avg\_loss = []

        # Create a dictionary to hold the results of the fitting

        results\_dict = {}

        best\_weights = {}

        # If input labels are 0/1 then make sure to convert labels to -1 and 1

        # for learning with the logistic loss.

        self.train\_labels = np.where(self.train\_labels==1, 1, -1)

        # Calculate folds

        fold\_indicies = []

        self.plotting\_dict = {

            "max\_iterations": [],

            "avg\_loss": []

        }

        # Pick random entries for validation/subtrain

        fold\_vec = np.random.randint(low=0,

                                     high=self.num\_folds,

                                     size=self.train\_labels.size)

        # for each fold,

        for fold\_number in range(self.num\_folds):

            subtrain\_indicies = []

            validation\_indicies = []

            # check if index goes into subtrain or validation list

            for index in range(len(self.train\_data)):

                if fold\_vec[index] == fold\_number:

                    validation\_indicies.append(index)

                else:

                    subtrain\_indicies.append(index)

            fold\_indicies.append([subtrain\_indicies, validation\_indicies])

        # Loop over the folds

        for foldnum, indicies in enumerate(fold\_indicies):

            # Get indicies of data chosen for this fold

            index\_dict = dict(zip(["subtrain", "validation"], indicies))

            set\_data\_dict = {}

            # Dictionary for test and train data

            for set\_name, index\_vec in index\_dict.items():

                set\_data\_dict[set\_name] = {

                    "X":self.train\_data[index\_vec],

                    "y":self.train\_labels[index\_vec]

                }

            # Define a variable called scaled\_mat which has

            subtrain\_data = set\_data\_dict["subtrain"]['X']

            subtrain\_labels = set\_data\_dict["subtrain"]['y']

            scaled\_mat = subtrain\_data

            # (1) filtered/removed any zero variance features

            #non\_variant\_indicies = \

        #        np.argwhere(np.all(scaled\_mat[..., :] == 0, axis=0))

            #scaled\_mat = np.delete(scaled\_mat,

            #                       non\_variant\_indicies,

            #                       axis=1)

            # (2) scaled any other features

            # self.pipe.fit(scaled\_mat, self.train\_labels)

            # (3) and an extra column of ones (for learning the intercept).

            #intercept\_col = np.ones((len(scaled\_mat), 1))

            #scaled\_mat = np.append(scaled\_mat,

            #                       intercept\_col,

            #                       axis=1)

            # Initialize an weight vector with size equal to the number of columns

            # in scaled\_mat.

            nrow, ncol = scaled\_mat.shape

            learn\_features = np.column\_stack([

                np.repeat(1, nrow),

                scaled\_mat

            ])

            weight\_vec = np.zeros(ncol+1)

            #learn\_features = learn\_features[:,0]

            subtrain\_mean = subtrain\_data.mean(axis=0)

            subtrain\_sd = np.sqrt(subtrain\_data.var(axis=0))

            # Then use a for loop from 0 to max\_iterations to iteratively compute

            # linear model parameters that minimize the average logistic loss over

            #the subtrain data.

            min\_loss = np.array([10])

            best\_iter = 0

            best\_coef = weight\_vec

            avg\_iter\_loss = []

            # Loop for each of the max iterations

            for index in range(self.max\_iterations):

                # Calculate prediction and log loss

                pred\_vec = np.matmul(learn\_features, weight\_vec)

                log\_loss = np.ma.log(1+np.exp(-subtrain\_labels \* pred\_vec))

                #print("iteration=%d log\_loss=%s"%(index,log\_loss.mean()))

                grad\_loss\_pred = -subtrain\_labels / \

                                    (1+np.exp(subtrain\_labels \* pred\_vec))

                grad\_loss\_pred = grad\_loss\_pred

                grad\_loss\_weight\_mat = grad\_loss\_pred \* learn\_features.T

                grad\_vec = grad\_loss\_weight\_mat.sum(axis=1)

                weight\_vec -= self.step\_size \* grad\_vec

                # get the smallest log loss

                if( not np.isinf(log\_loss.mean()) <= min\_loss.mean() ):

                    min\_loss = log\_loss

                    best\_iter = index

                    best\_coef = weight\_vec

                # build list of loss values

                avg\_iter\_loss.append(log\_loss.mean())

            # save best stuff from each pass

            results\_dict[best\_iter] = min\_loss.mean()

            best\_weights[best\_iter] = best\_coef

            self.avg\_loss.append(avg\_iter\_loss)

        # get single best weight and intercept

        best\_result = max(results\_dict, key = results\_dict.get)

        self.coef\_ = best\_weights[best\_result][1:]

        self.intercept\_ = best\_weights[best\_result][0]

        # these get saved for plotting

        self.avg\_loss = np.asarray(self.avg\_loss)

        self.avg\_loss = self.avg\_loss.mean(axis=0)

        # At the end of the algorithm you should save the learned

        # weights/intercept (on original scale) as the coef\_ and intercept\_

        # attributes of the class (values should be similar to attributes of

        # LogisticRegression class in scikit-learn).

    def decision\_function(self, X):

        # Implement a decision\_function(X) method which uses the learned weights

        # and intercept to compute a real-valued score (larger for more likely

        # to be predicted positive)

        # use best coef and inter to build result

        pred\_vec = np.matmul(X, self.coef\_) + self.intercept\_

        return pred\_vec

    def predict(self, test\_features):

        # Implement a predict(X) method which uses np.where to threshold the

        # predicted values from decision\_function, and obtain a vector of

        # predicted classes (1 if predicted value is positive, 0 otherwise).

        pred\_vec = self.decision\_function(test\_features)

        # positive values are 1, anything else is 0

        pred\_vec[pred\_vec > 0] = 1

        pred\_vec[pred\_vec <= 0] = 0

        # predicted values using either scaled or unscaled features agree:

        # print(pred\_vec)

        return( pred\_vec )

# <-- END INITIALIZATION -->

# <-- BEGIN FUNCTIONS -->

# FUNCTION: MAIN

#   Description  : Main driver for Assignment Three

#   Inputs       : None

#   Outputs      : PlotNine graphs saved to program directory

#   Dependencies : build\_image\_df\_from\_dataframe

def main():

    # Display the title

    print("\nCS 499: Homework 4 Program Start")

    print("================================\n")

    # Suppress annoying plotnine warnings

    warnings.filterwarnings('ignore')

    # Download data files

    download\_data\_file(spam\_data\_file, spam\_data\_url, spam\_file\_path)

    download\_data\_file(ziptrain\_file, ziptrain\_url, ziptrain\_file\_path)

    download\_data\_file(ziptest\_file, ziptest\_url, ziptest\_file\_path)

    # Open each dataset as a pandas dataframe

    spam\_df = pd.read\_csv(spam\_data\_file, header=None, sep=" ")

    zip\_train\_df = pd.read\_csv(ziptrain\_file, header=None, sep=" ")

    zip\_test\_df = pd.read\_csv(ziptest\_file, header=None, sep=" ")

    # Concat the two zip dataframes together

    zip\_df = pd.concat([zip\_train\_df, zip\_test\_df])

    # Drop rows of dataframes where the label is not ( 0 or 1)

    zip\_df[0] = zip\_df[0].astype(int)

    zip\_df = zip\_df[zip\_df[0].isin([0, 1])]

    # Drop empty col from zip dataframe

    zip\_df = zip\_df.drop(columns=[zip\_empty\_col])

    # Create label vectors

    zip\_labels = zip\_df[0]

    spam\_labels = spam\_df[spam\_label\_col]

    # Create numpy data

    zip\_data = zip\_df.iloc[:, 1:256].to\_numpy()

    spam\_data = spam\_df.iloc[:, :56].to\_numpy()

    pipe.fit(spam\_data, spam\_labels)

    # Create data dictionary

    data\_dict = {

        'spam' : [spam\_data, spam\_labels],

        'zip' : [zip\_data, zip\_labels]

    }

    # Loop through each data set

    for data\_set, (input\_data, output\_array) in data\_dict.items():

        # Output message for logging

        print("Working on set: " + str(data\_set))

        current\_set = str(data\_set)

        # Scale the data set

        # Loop over each fold for each data set

        for foldnum, indicies in enumerate(kf.split(input\_data)):

            print("Fold #" + str(foldnum))

            # Set up input data structs

            index\_dict = dict(zip(["train", "test"], indicies))

            param\_dicts = [{'n\_neighbors':[x]} for x in range(1, 21)]

            logreg\_param\_dicts = \

                [{'max\_iterations':max\_it, 'step\_size':steps} \

                    for max\_it in [100, 1000, 2000] \

                    for steps in [1, 0.1, 0.01, 0.001]]

            logreg\_param\_nosteps\_dicts = \

                [{'max\_iterations':max\_it} \

                    for max\_it in [100, 1000, 2000]]

            # Establish different models

            clf = GridSearchCV(KNeighborsClassifier(), param\_dicts)

            linear\_model = sklearn.linear\_model.LogisticRegressionCV(cv=5)

            RegressionCV = MyCV(estimator=MyLogReg,

                                param\_grid=logreg\_param\_dicts,

                                cv=CV\_VAL)

            RegressionCVNoSteps = MyCV(estimator=MyLogReg,

                                param\_grid=logreg\_param\_nosteps\_dicts,

                                cv=CV\_VAL)

            # Creating dictionary with input and outputs

            set\_data\_dict = {}

            for set\_name, index\_vec in index\_dict.items():

                set\_data\_dict[set\_name] = {

                    "X":input\_data[index\_vec],

                    "y":output\_array.iloc[index\_vec].reset\_index(drop=True)

                }

            # Train the models with given data

            clf.fit(\*\*set\_data\_dict["train"])

            linear\_model.fit(\*\*set\_data\_dict["train"])

            RegressionCV.fit(\*\*set\_data\_dict["train"])

            RegressionCVNoSteps.fit(\*\*set\_data\_dict["train"])

            # Get most common output from outputs for featureless set

            most\_common\_element = mode(set\_data\_dict["train"]['y'])

            # Get results

            cv\_df = pd.DataFrame(clf.cv\_results\_)

            cv\_df.loc[:, ["param\_n\_neighbors", "mean\_test\_score"]]

            pred\_dict = {

                "GridSearchCV + KNeighborsClassifier": \

                    clf.predict(set\_data\_dict["test"]["X"]),

                "LogisticRegressionCV": \

                    linear\_model.predict(set\_data\_dict["test"]["X"]),

                "MyCV + MyLogReg (No Step Size Training)": \

                    RegressionCVNoSteps.predict(set\_data\_dict["test"]["X"]),

                "MyCV + MyLogReg": \

                    RegressionCV.predict(set\_data\_dict["test"]["X"]),

                "Featureless":most\_common\_element

            }

            # Build results dataframe for each algo/fold

            for algorithm, pred\_vec in pred\_dict.items():

                test\_acc\_dict = {

                    "test\_accuracy\_percent":(

                        pred\_vec == set\_data\_dict["test"]["y"]).mean()\*100,

                    "data\_set":data\_set,

                    "fold\_id":foldnum,

                    "algorithm":algorithm

                }

                test\_acc\_df\_list.append(pd.DataFrame(test\_acc\_dict, index=[0]))

    # Build accuracy results dataframe

    test\_acc\_df = pd.concat(test\_acc\_df\_list)

    # Print results

    print("\n")

    print(test\_acc\_df)

    # Plot results

    plot = (p9.ggplot(test\_acc\_df,

                        p9.aes(x='test\_accuracy\_percent',

                        y='algorithm'))

                   + p9.facet\_grid('. ~ data\_set')

                   + p9.geom\_point())

    print(plot)

    print("\nCS 499: Homework 4 Program End")

    print("==============================\n")

# FUNCTION : DOWNLOAD\_DATA\_FILE

#   Description: Downloads file from source, if not already downloaded

#   Inputs:

#       - file      : Name of file to download

#       - file\_url  : URL of file

#       - file\_path : Absolute path of location to download file to.

#                     Defaults to the local directory of this program.

#   Outputs: None

def download\_data\_file(file, file\_url, file\_path):

    # Check for data file. If not found, download

    if not os.path.isfile(file\_path):

        try:

            print("Getting file: " + str(file) + "...\n")

            urllib.request.urlretrieve(file\_url, file\_path)

            print("File downloaded.\n")

        except(error):

            print(error)

    else:

        print("File: " + str(file) + " is already downloaded.\n")

# Launch main

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Program Output:**

Diagram

Description automatically generated with low confidence

(Figure 1) Attempt at plotting loss vs iterations, facetted by step size and max iteration

**Calendar

Description automatically generated with medium confidence**

(Figure 2) Test accuracy for Zip and Spam training set. Step size training clearly increases the training accuracy.

Text

Description automatically generatedText

Description automatically generated with medium confidenceText

Description automatically generated

**Question Answers / Commentary:**

I was unable to complete some of the major requirements of this assignment. While the Logistic Regression model and accompanying CV were able to correctly predict the correct outcomes with great accuracy, I was not able to create the plots of Loss vs. Iterations. Additionally, I had difficulty configuring GGPlot to correctly display axis labels and information in an aesthetic manner.

For the extra credit portion of the assignment, I added the step size training parameters into my CV. Because it was correctly built last assignment to allow for this functionality, it can take any number of parameters without having to be rebuilt, and has K-fold cross validation built in. I plotted the accuracy of the CV with and without step size training, and step size training clearly increases accuracy.

Overall, I would cold the model a success. Despite not being able to correctly plot the training/validation loss over time, the end results speak for themselves.