My submission for this assignment does not stray too far away from last weeks assignment guidelines. My homework 3 submission was not able to get the MyCV and KNN class models to display to our plot or even be calculated in the first place correctly. I had ran into some issues with declare proper types in our class constructor for KNN and had to add a repetitive conditional to check if our n\_neighbors attribute is an integer or list then adjust accordingly from there. The code is pretty long for just one file and if this continues to be used for future projects I will separate some of the methods/classes into new files and keep main in a main.py file. This final program does not implement the classes designated for the assignment as this was a bit much for me for one week to get something working :(.

# import for debugging with pdb

import pdb

import traceback

# lib for retrieving src file from web

import urllib.request

# lib for reading files on OS

import os

# lib used for copying src file info into destination

import pandas as pd

import plotnine as p9

import numpy as np

# could not figure out how to calculate the mode of a list, using mode from lib

from statistics import mode

import sklearn

#train/test splits

from sklearn.model\_selection import KFold

#selecting best # of neighbors

from sklearn.model\_selection import GridSearchCV

#nearest\_neighbors prediction

from sklearn.neighbors import KNeighborsClassifier

# increase iteration sz

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

# directory for data files

data\_dir = 'data/'

# our src files we want to download; test set

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

test\_file = 'data/zip.test.gz'

# train set

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

train\_file = 'data/zip.train.gz'

# spam set

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

spam\_file = 'data/spam.data'

# number of columns in test file (257) count from zero

conc\_cols = 257

# number of columns in spam file (56) count from zero

spam\_cols = 57

# split our data set into train and test sets

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

# increase the max iteration from default 100

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

# declare list for set

test\_acc\_df\_list = []

# cv constant used in cv= and cv class call, to minimize reuse of vals

const\_cv = 5

# neighbors constant val

const\_n = 20

class data:

def retrieve(src\_file, src\_url):

"""

method to download specified files. call from main, pass in

src files to retrieve

"""

# lets store these files in a directory, create /data if DNE

if not os.path.exists(data\_dir):

os.makedirs(data\_dir)

"""

check if a file exists in the current directory

retrieve a file given the url

"""

if not os.path.isfile(src\_file):

urllib.request.urlretrieve(src\_url, src\_file)

print("Downloading src file into " + src\_file + " from " + src\_url +

"...\n")

else:

print(src\_file + " already exists in this folder...continuing anyway\n")

def init(test\_file, train\_file, spam\_file, conc\_file, data\_dict):

"""

method to initialize our multiple frames.

- take in src files

- create a dataframe specific to the src file

- concatenate test and train files into 1 PD DF

- drop specified rows of the src file

- convert our data into numpy matrices

- store in dictionary

- return dictionary for further manipulation

"""

# read in downloaded src file as a pandas dataframe

# seperate dataframes because different manipulations will be done

df\_test = pd.read\_csv(test\_file, header=None, sep=" ")

df\_train = pd.read\_csv(train\_file, header=None, sep=" ")

df\_spam = pd.read\_csv(spam\_file, header=None, sep=" ")

# reassign concatenated test and train frame

df\_conc = pd.concat([df\_train, df\_test])

# remove any rows which have non-01 labels

df\_conc[0] = df\_conc[0].astype(int)

# df\_spam[0] = df\_spam[0].astype(int)

df\_conc = df\_conc[df\_conc[0].isin([0, 1])]

# df\_spam = df\_spam[df\_spam[0].isin([0, 1])]

df\_conc = df\_conc.drop(columns=[conc\_cols])

# initialize and convert outputs to a label vector

df\_conc\_labels = df\_conc[0]

df\_spam\_labels = df\_spam[spam\_cols]

"""

Convert our dataframe to a dictionary with numpy array exlcuding the

first column; iloc for row and col specifying.

"""

data\_conc = df\_conc.iloc[:, 1:256].to\_numpy()

data\_spam = df\_spam.iloc[:, :56].to\_numpy()

# create numpy data from vectors

data\_dict = {

"zip":[data\_conc, df\_conc\_labels],

"spam":[data\_spam, df\_spam\_labels]

}

# return our values back to the call

return data\_dict

class MyKNN:

"""

MyKNN class, according to \*.org guideline, that \*should\* work just like

sklearn.neighbors.KNeighborsClassifier

"""

def \_\_init\_\_(self, n\_neighbors):

"""

instantiate neighbors param stored as an attribute of our instance

\_\_init\_\_: recieves constructors args initializing new obj

self: instance of class for attribute manipulation, always first

attribute of instance. convention ! keyword

member

"""

"""

issues iterating over int types, int obj ! subscriptable, etc

use conditional to determine if folds are declared as list or int

adjust accordingly

"""

if isinstance(n\_neighbors, list):

self.n\_neighbors = n\_neighbors[0]

else:

self.n\_neighbors = n\_neighbors

self.train\_features = []

self.train\_labels = []

def fit(self, X, y):

"""

fit method with X=train\_features, y=train\_labels, storing data as

attributes of our instance

features: input data

label: output data based on input

"""

# store feats/labs in respective lists; can do for loop for many members

self.train\_features = X

self.train\_labels = y

def predict(self, test\_features):

"""

compute binary vector of predicted class label from demo3 in class

X = test\_features

y = train\_labels

features represent data we want to pass in, labels represent the data

we run our computations on

"""

# declare list to store this computed prediction in

future\_list = []

# traverse each test data row; features

for test\_data\_row in range(len(test\_features)):

# we want to store each iteration in a list representing best param

neighbors\_list = []

if isinstance(self.n\_neighbors, list):

self.n\_neighbors = self.n\_neighbors[0]

# compute distances with all of train data

test\_i\_features = test\_features[test\_data\_row,:]

diff\_mat = self.train\_features - test\_i\_features

"""

Each distance is the square root of the sum of squared

differences over all features

"""

squared\_diff\_mat = diff\_mat \*\* 2

# sum over columns, for each row

squared\_diff\_mat.sum(axis=0)

# sum over rows

distance\_vec = squared\_diff\_mat.sum(axis=1)

# sort distances w/ numpy.argsort to find smallest n

sorted\_indices = distance\_vec.argsort()

# n\_neighbors is list type, must convert to int type

nearest\_indices = sorted\_indices[:self.n\_neighbors]

# append result to set

for final\_list in nearest\_indices:

neighbors\_list.append(self.train\_labels[final\_list])

future\_list.append(mode(neighbors\_list))

return future\_list

class MyCV:

"""

MyCV class, according to \*.org guideline, that \*should\* work just like

sklearn.model\_selection.GridSearchCV. this class should perform

best parameter selection thru cross-validation for any estimator

\*<--NOTE-->\*: nothing in this class should be specific to the nearest

neighbors algorithm! It should not have any reference to “n\_neighbors” in

the class definition. These methods are sort of copied from the class MyKNN

and is similar to our run\_algo method

"""

def \_\_init\_\_(self, estimator, param\_grid, cv):

"""

describe this constructor

"""

self.train\_features = []

self.train\_labels = []

self.inputs = None

self.param\_grid = param\_grid

self.folds = cv

self.estimator = estimator(self.folds)

self.best\_fit = None

def fit(self, X, y):

"""

should compute the best number of neighbors using K-fold cross-validation,

with the number of folds defined by the cv parameter

"""

self.train\_features = X

self.train\_labels = y

# inputs of our model

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

# create a pd df for folds

best\_param = pd.DataFrame()

# store defined folds in list

fold\_index = []

# assigning random fold ID numbers to each observation

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

size=self.train\_labels.size)

# traverse k subtrain/validation splits

#for folds in range(self.fold\_num):

# is\_set\_dict = {

# "validation":fold\_vec == fold,

# "subtrain":fold\_vec != fold,

# }

"""

declare folds var for traversing folds and populating subtrain

and validation lists

"""

for current\_fold in range(self.folds):

# empty list for subtrain and validation

sub = []

val = []

# make sure current element populates the above lists

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

# maybe use while loop here instead of conditional

if fold\_vec[current\_element] == current\_fold:

# append our validation list

val.append(current\_element)

else:

# append our sub list

sub.append(current\_element)

# add in sub and val lists into our fold\_index list

fold\_index.append([sub, val])

# from below algo class

for fold\_id, indices in enumerate(fold\_index):

print("SUBFOLD: " + str(fold\_id))

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

# param\_dicts = [self.param\_grid]

set\_data\_dict = {}

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)

}

# empty populated dict

populated\_dict = {}

# current attribute iterator used in the following traversal

current\_attr = 0

# from demo3 in class, iterating of param grid prediction sub/val

for param\_index in self.param\_grid:

for param\_name, param\_val in param\_index.items():

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

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

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

populated\_dict[current\_attr] = \

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

# update curr attr

current\_attr += 1

# append result into our dict

best\_param = best\_param.append(populated\_dict, ignore\_index=True)

# calculate the average of our params given the fold

avg = dict(best\_param.mean())

# from the calculated average determine best fit using max()

determined\_result = max(avg, key = avg.get)

# store our determine result in our param\_grid

self.best\_fit = self.param\_grid[determined\_result]

def predict(self, test\_features):

"""

should run estimator to predict the best number of neighbors

which is a set attribute of estimator at the end of fit

"""

# traverse thru our models and append into our esimator

# from above ^

for param\_name, param\_val in self.best\_fit.items():

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

# run our estimator passing in the assigned best estimated set

self.estimator.fit(\*\*self.inputs)

# assign prediction to future val

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

# return our prediction

return future

class algo:

def run(data\_dict):

"""

algorithms shown from first class demo that we've been working with

"""

#test\_acc\_df\_list = []

for data\_set, (input\_mat, output\_vec) in data\_dict.items():

print("SET: " + str(data\_set))

pipe.fit(input\_mat, output\_vec)

for fold\_id, indices in enumerate(kf.split(input\_mat)):

print("FOLD: " + str(fold\_id))

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

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

# does subtrain/validation splits.

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

# copy above for linear model. call cv=5 in initial pipe was not

# recognized; try a call here

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

"""

call our MyCV class to run our models passing in our MyKNN class

am unsure of accuracy and placement of this call but am curious

if parameters passed in are what is expected

"""

cv\_model = MyCV(MyKNN, param\_dicts, const\_cv)

set\_data\_dict = {}

# add in our new parameters be want to be working with

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

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

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

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

}

"""

\* is unpacking a tuple to use as the different positional arguments

clf.fit(set\_data\_dict["train"][0], set\_data\_dict["train"][1])

train models and stub out linear\_model

\*\* is unpacking a dict to use as the named arguments

train models and stub out linear\_model and create algo for finding

mode

"""

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

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

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

featureless\_model = mode(set\_data\_dict["train"]['y'])

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

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

pred\_dict = {

"GridSearchCV \n + \nKNeighborsClassifier": \

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

"LogisticRegressionCV": \

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

"MyCV + MyKNN": \

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

# featureless is inaccurate

"Featureless": featureless\_model

}

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

test\_acc\_dict = {

"test\_accuracy\_percentage":(

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

"data\_set":data\_set,

"fold\_id":fold\_id,

"algorithm":algorithm

}

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

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

"""

print our final data dict to view accuracy percentage, dataset,

fold\_id, and algo used

"""

print(test\_acc\_df)

# return data frame for passing into plot

return test\_acc\_df

def plot(test\_acc\_df):

"""

make a ggplot to visually examine which learning algorithm is

best for each data set

"""

# define plot variable

gg = (p9.ggplot(test\_acc\_df,

p9.aes(x = 'test\_accuracy\_percentage', y = 'algorithm'))

# .~ spreads vals across columns

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

# Use geom\_point to create scatterplots

+p9.geom\_point())

#+p9.theme(subplots\_adjust={'right':2.0,'bottom':0.2}))

print(gg)

def main():

# empty dictionary representing data frames

data\_dict = {}

# retrieve our data files using retrieve function in data class

data.retrieve(test\_file, test\_url)

data.retrieve(train\_file, train\_url)

data.retrieve(spam\_file, spam\_url)

# to be populated

conc\_file = None

"""

initialize out respective data frames

passed in:

- test file

- train file

- spam file

- empty file that represents concatenated test + train files

- empty dictionary representing our final dataframes

returns:

- conc: concatenated test + train dataframes

- spam: spam dataframe

- data\_dict: data dictionary containing zip + spam frame to plot

"""

(data\_dict) = data.init(test\_file, train\_file,

spam\_file, conc\_file, data\_dict)

# run our manipulations on our data, calling both KNN and CV classes

data\_set = algo.run(data\_dict)

# plot our data

viz\_data = algo.plot(data\_set)

# run main

if \_\_name\_\_ == '\_\_main\_\_':

main()