I found the hardest part of this assignment was creating the predict methods in our respective classes and determining where to call the classes from the run\_algo method I created from the code you had shown us in class. You had told me my featureless data was too accurate but I was unsure of where to fix the accuracy of the set. I had spent a lot of the time researching oop in python and how to stub everything out into its respective method. I created a class for the previous assignments algorithms that was shown in class demo.

Overall it was hard to tell the difference in the approach for this assignment vs the last assignment given that the results are skewed in my code. Somewhere there is an issue with the featureless model and its accuracy being higher than all others. I was not able to get the desired solution and there are errors riddled within my knn and cv classes especially when looking at the plots for comparison. Data does not seem top get passed into our classes to determine a best fit for spam or test. I will spend this weekend trying to work on a viable solution for some feedback

Here is my code:

# 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

from sklearn.model\_selection import KFold #train/test splits

from sklearn.model\_selection import GridSearchCV #selecting best # of neighbors

from sklearn.neighbors import KNeighborsClassifier #nearest\_neighbors prediction.

from sklearn.pipeline import make\_pipeline # increase iteration sz

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

"""

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

src files to retrieve

"""

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

# 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")

"""

method to initialize our multiple frames.

- take in src file

- create a dataframe

- drop specified rows of the src file

- convert our data into numpy

"""

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

# 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\_test, df\_train])

# 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])]

# 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.

"""

# create numpy data from vectors

data\_dict = {

"test":(df\_conc.iloc[:,1:conc\_cols-1].to\_numpy(), df\_conc[0]),

"spam":(df\_spam.iloc[:,:spam\_cols-1].to\_numpy(), df\_spam[0]),

}

# print our dataframes

print(df\_spam)

print(df\_conc)

# return our values back to the call

return df\_test, df\_train, df\_spam, df\_conc, data\_dict

"""

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

sklearn.neighbors.KNeighborsClassifier

"""

class MyKNN:

"""

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

"""

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

# init neighbors attribute of instance

self.nearest = n\_neighbors

self.train\_features = []

self.train\_labels = []

"""

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

"""

def fit(self, X, y):

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

self.train\_features = X

self.train\_labels = y

"""

compute binary vector of predicted class label from demo3 in class

X = test\_features

"""

def predict(self, test\_features):

# declare list to store this computed prediction in

# \*\*NOTE\*\* following is from line 33-36 in the demo

predict\_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

best\_param = []

# 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()

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

# append result to set

for final\_list in nearest\_indices:

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

predict\_list.append(best\_param)

return(predict\_list)

"""

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

"""

class MyCV:

# from in class demo3 in repo

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

self.train\_features = []

self.train\_labels = []

self.train\_set = None

self.param\_grid = []

self.folds = const\_cv

self.estimator = estimator(self.folds)

self.best\_fit = 0

self.fold\_num = 0

"""

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

with the number of folds defined by the cv parameter

"""

def fit(self, X, y):

self.train\_features = X

self.train\_labels = y

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

# df for folds

folds\_df = pd.DataFrame()

# store defined folds in list

folds = []

# 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,

}

# from below algo class

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

print(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]

}

result\_dict = {}

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

for param\_dict in self.param\_grid:

#param\_name, param\_value in param\_dict.items():

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

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

self.est.predict(set\_data["validation"]["X"])

result\_dict[param\_value] = (prediction == set\_data["test"]["y"]).mean()\*100

# append result

result\_df = result\_df.append(result\_dict)

avg = dict(result\_df.mean())

self.best\_fit = avg

"""

should run estimator to predict the best number of neighbors

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

"""

def predict(self, test\_features):

# run our estimator passing in the assigned best estimated set

self.estimator.nearest = self.best\_fit

self.estimator.fit(\*\*self.trained\_set)

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

return result

class algo:

"""

algorithm shown in class and from our demo.

"""

def run\_algo(data\_dict):

test\_acc\_df\_list = []

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

print(data\_set)

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

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

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

print(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=const\_cv)

"""

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]

}

# \* 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(X=set\_data\_dict["train"]["X"],

# y=set\_data\_dict["train"]["y"]])

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

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

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

featureless\_model = mode(output\_vec)

#clf.best\_params\_

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 + My\_KNN":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)

return test\_acc\_df

"""

make a ggplot to visually examine which learning algorithm is

best for each data set

"""

def plot(test\_acc\_df):

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())

print(gg)

def main():

data\_dict = {}

# retrieve our data files using retrieve function

retrieve(test\_file, test\_url)

retrieve(train\_file, train\_url)

retrieve(spam\_file, spam\_url)

conc\_file = 0

(test, train, spam, conc, \_dict) = df\_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 = run\_algo(\_dict)

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

# plot our data

viz\_data = plot(data\_set)

# run main

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

main()

Here is my result:

