Overall the assignment did not stray too much from what was given in the demo. However, to

try and keep a modularized approach to how we implement code, I attempted to create different methods for some of the big functions we were passing around. In the end my code did not end up working how I wanted due to some python programming errors. I was unsure of how to pass variables around the program and which ones to pass and assign, etc. The final error I got was not able to concat test\_acc\_df\_list. This has to be because of the data\_dict not being seeing inside of the train method. Just like last time I am confident if I was given a bit more time I would be able to tackle this problem. The method for plotting our points was eventually not met so I am unsure if it was implemented correctly. Based off of the demos and my previous assignment, I decided to use the code for plotting and hoping it would work. This time I was able to find a decent documentation on plotting using the facet\_grid but again, I am not sure if this was implemented correctly. In the coming days I might submit a new file that is working.

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 shutil

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

import statistics

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

# our src files we want to download. spam and test sets

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

test\_file = 'zip.test.gz'

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

spam\_file = 'spam.data'

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

test\_cols = 257

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

spam\_cols = 57

# dataframe list

#test\_acc\_df\_list = []

"""

our last assignment was only downloading 1 file, to adhere to guidelines and

avoid repeatable code, created a method we will use to retrieve multiple files.

"""

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

"""

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 + " from " + src\_url + "...\n")

else:

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

"""

since we are initializing multiple dataframes for out multiple

sources of data, I wanted to try and minimize repeating code.

this method will

- take in src file

- create a dataframe

- drop specified rows of the src file

- convert our data into numpy

Hopefully this is allowed!

notice: (am repeating code) but spent too much time creating traveral

type solution going over a list of src\_files and src\_urls.

Wanted to seperate file into some seperate functions for the sake of

readability.

"""

def df\_init(src1, src2, src1\_cols, src2\_cols, data\_dict):

# read in downloaded src file as a pandas dataframe

# seperate dataframes because different manipulations will be done

df1 = pd.read\_csv(src1, header=None, sep=" ")

df2 = pd.read\_csv(src2, header=None, sep=" ")

# remove any rows which have non-01 labels

df1[0] = df1[0].astype(int)

df2[0] = df2[0].astype(int)

df1 = df1[df1[0].isin([0, 1])]

df2 = df2[df2[0].isin([0, 1])]

# initialize and convert outputs to a label vector

df1\_labels = df1[0]

df2\_labels = df2[0]

"""

Convert our dataframe to a dictionary with numpy array exlcuding the

first column; iloc for row and col specifying.

"""

data\_dict = {

"test":(df1.loc[:,1:].to\_numpy(), df1[0]),

"spam":(df2.loc[:,1:].to\_numpy(), df2[0]),

}

# print our dataframes to visualize in tabular form

print(df1)

print(df2)

#print(data\_dict)

return df1, df2, data\_dict

"""

algorithm shown in class and from our demo.

"""

def train(data\_dict):

test\_acc\_df\_list = []

# increase the max iteration from default 100

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

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

print(data\_set)

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.LogisticRegression(cv=5)

set\_data\_dict = {}

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

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

input\_mat [ index\_vec ],

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

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

# method 2: dict instead of tuple.

set\_data\_dict = {}

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 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"])

#mode = max(set(output\_vec))

import statistics

MCE = mode(output\_vec)

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

clf.best\_params\_

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

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

pred\_dict = {

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

#TODO add featureless and linear\_model.

"featureless": MCE,

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

}

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":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

"""

an attempt to modularize our code to improve readibility, this method

will 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\_percent',y='algorithm'))

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

+p9.geom\_point())

print(gg)

def main():

data\_dict = {}

test\_acc\_df = []

# retrieve our source files. Spam and zip.

retrieve(test\_file, test\_url)

retrieve(spam\_file, spam\_url)

# call method to initialize our data frames, convert to numpy arrays,

df\_init(test\_file, spam\_file, test\_cols, spam\_cols, data\_dict)

# call method to perform our algorithm shown in class and the demo

train(data\_dict)

plot(test\_acc\_df)

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

# run main

main()