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:

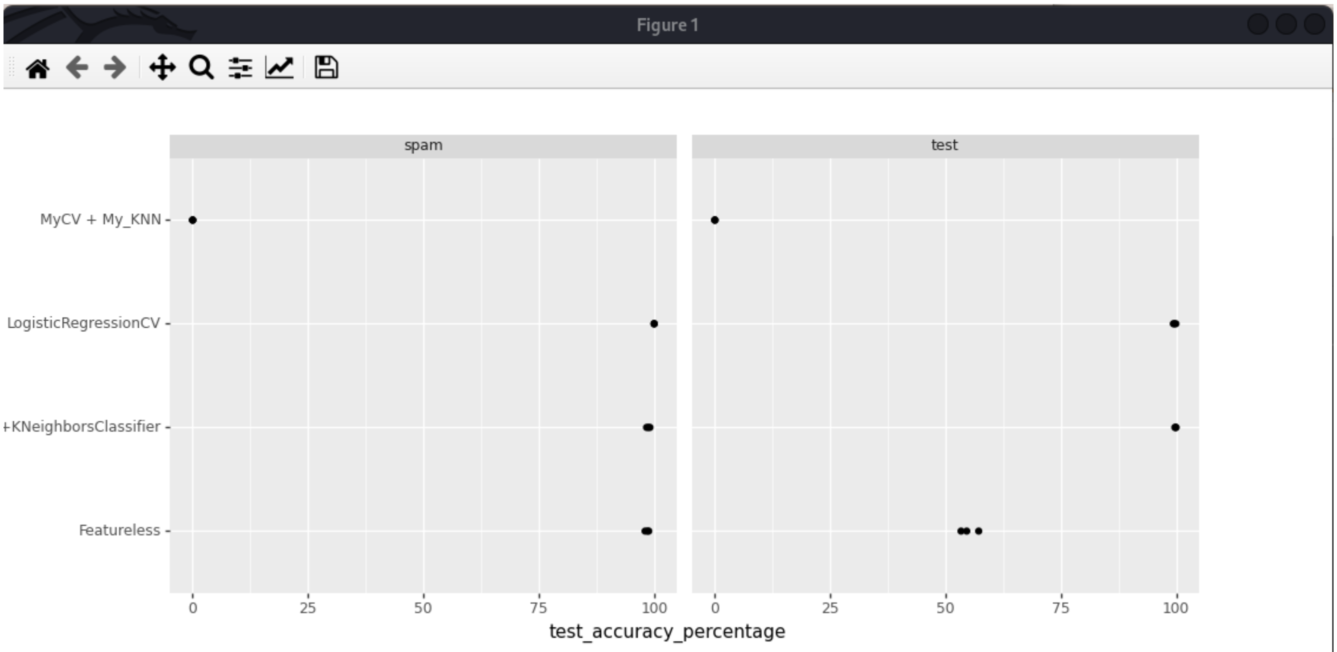

Figure 1
spam
test
MyCV + My_KNN
LogisticRegressionCV
+KNeighborsClassifier
Featureless
0
25
50
75
100
test_accuracy_percentage