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()
```