* attempt 2

I had spent the time since submitting attempt one trying to wrap my head around passing variables in and out of functions. I was finally able to get the program working after realizing the df_init function is returning a tuple so our values passed should be as such. After assigning the tuple I run the train() method on the last param passed in, c. from there our train() method will populate the test_acc_df, populated into variable d. Finally d is passed into plot() where we are plotting out points in facet_grid type format. Here is the code with some changes from attempt 1 highlighted in yellow:

```
# 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
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

# 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

"""
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.LogisticRegressionCV(cv=5)
            linear_model = sklearn.linear_model.LogisticRegressionCV(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))
            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 test.
    retrieve(test_file, test_url)
    retrieve(spam_file, spam_url)
    # call method to initialize our data frames, convert to numpy arrays,
    (a, b, c) = df_init(test_file, spam_file, test_cols, spam_cols, data_dict)
    # call method to perform our algorithm shown in class and the demo
    d = train(c)
    # plot our values by passing in the previously filled variable d
    plot(d)

if __name__ == '__main__':
    # run main
    main()
```

* attempt 1

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()
```