Assignment Four

CS 499

Richard McCormick (RLM443)

Python Program:

```
# <-- BEGIN IMPORTS / HEADERS -->
import os
import urllib
import urllib.request
import pandas as pd
import numpy as np
import plotnine as p9

import sklearn
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

from statistics import mode
import inspect
import warnings
# <-- END IMPORTS / HEADERS -->

# <-- BEGIN INITIALIZATION -->
# FILE VARIABLES
download_directory = "."

# - Spam data variables
spam_data_url = "https://hastie.su.domains/ElemStatLearn/datasets/spam.data"
spam_data_file = "spam.data"
spam_file_path = os.path.join(download_directory, spam_data_file)

# - Zip data (Training) variables
ziptrain_url = "https://hastie.su.domains/ElemStatLearn/datasets/zip.train.gz"
ziptrain_file = "zip.train.gz"
ziptrain_file_path = os.path.join(download_directory, ziptrain_file)

# - Zip data (Test) variables
ziptest_url = "https://hastie.su.domains/ElemStatLearn/datasets/zip.test.gz"
```

```python
ziptest_file = "zip.test.gz"
ziptest_file_path = os.path.join(download_directory, ziptest_file)

# CONSTANT VARIABLES
spam_label_col = 57
zip_empty_col = 257
MyKNN_N_NEIGHBORS_VAL = 20
CV_VAL = 5

# MISC. VARIABLES
kf = KFold(n_splits=3, shuffle=True, random_state=1)
test_acc_df_list = []
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))

#CLASS DEFINITIONS
class MyCV():
    def __init__(self, **kwargs):
        # Initialize parameters and setup variables
        self.train_features = []
        self.train_labels = []
        self.training_data = None

        kwargs.setdefault("num_folds", 5)
        for key, value in kwargs.items():
            setattr(self, key, value)

        self.estimator = self.estimator()
        self.best_model = None

        self.plotting_df = pd.DataFrame()

    def fit(self, X, y):
        # Populate internal data structures
        self.train_features = X
        self.train_labels = y
        self.training_data = {'X':self.train_features, 'y':self.train_labels}

        # Create a dataframe to temporarily hold results from each fold
        best_paramter_df = pd.DataFrame()

        # Calculate folds
        fold_indicies = []

        # Pick random entries for validation/subtrain
        fold_vec = np.random.randint(low=0,
```

```python
                                  high=self.num_folds,
                                  size=self.train_labels.size)

        # for each fold,
        for fold_number in range(self.num_folds):
            subtrain_indicies = []
            validation_indicies = []
            # check if index goes into subtrain or validation list
            for index in range(len(self.train_features)):
                if fold_vec[index] == fold_number:
                    validation_indicies.append(index)
                else:
                    subtrain_indicies.append(index)

            fold_indicies.append([subtrain_indicies, validation_indicies])


        printing_df = pd.DataFrame()
        # Loop over the folds
        for foldnum, indicies in enumerate(fold_indicies):
            print("(MyCV) Subfold #" + str(foldnum))

            # Get indicies of data chosen for this fold
            index_dict = dict(zip(["subtrain", "validation"], indicies))
            set_data_dict = {}

            # Dictionary for test and train data
            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)
                }

            # Create a dictionary to hold the results of the fitting
            results_dict = {}

            parameter_index = 0
            # Loop over each parameter in the param_grid
            for parameter_entry in self.param_grid:
                for param_name, param_value in parameter_entry.items():
                    setattr(self.estimator, param_name, param_value)

                # Fit fold data to estimator
                self.estimator.fit(**set_data_dict["subtrain"])
```

```
#                 printing_df = printing_df.append({'loss':
self.estimator.avg_loss, 'iterations': self.estimator.max_iterations,
'step_size': self.estimator.step_size, 'fold':foldnum}, ignore_index=True)

                # Make a prediction of current fold's test data
                prediction = \
                    self.estimator.predict(set_data_dict["validation"]['X'])

                # Determine accuracy of the prediction
                results_dict[parameter_index] = \
                (prediction == set_data_dict["validation"]["y"]).mean()*100

                # index only serves to act as key for results dictionary
                parameter_index += 1

            # Store the results of this param entry into dataframe
            best_paramter_df = best_paramter_df.append(results_dict,
                                                       ignore_index=True)

        # all of this stuff is for plotting loss vs iterations...
#        printing_df = printing_df.groupby(['step_size',
'iterations']).loss.apply(list)
#        printing_df = printing_df.to_frame().reset_index()
#        printing_df['iteration_list'] = ""
#        for index, row in printing_df.iterrows():
#            new_loss_row = row['loss']
#            new_loss_row = np.mean(new_loss_row, axis=0)
#            printing_df.at[index, 'loss'] = new_loss_row

#            new_iter_row = row['iterations']
#            new_iter_row = np.arange(new_iter_row)
#            printing_df.at[index, 'iteration_list'] = new_iter_row
#
#        printing_df = printing_df.explode(['loss', 'iteration_list'])

        # Average across all folds for each parameter
        averaged_results = dict(best_paramter_df.mean())

        # From the averaged data, get the single best model
        best_result = max(averaged_results, key = averaged_results.get)

        # Store best model for future reference
        self.best_model = self.param_grid[best_result]

    def predict(self, test_features):
```

```
        # Load best model into estimator
        for param_name, param_value in self.best_model.items():
            setattr(self.estimator, param_name, param_value)

        # Fit estimator to training data
        self.estimator.fit(**self.training_data)

        # Make a prediction of the test features
        prediction = self.estimator.predict(test_features)

        return(prediction)

class MyLogReg():
    def __init__(self, **kwargs):
        kwargs.setdefault("num_folds", 5)
        kwargs.setdefault("max_iterations", 10) # trained through cv
        kwargs.setdefault("step_size", 0.0001) # trained through cv

        self.train_data = None
        self.train_labels = None

        self.coef_ = None
        self.intercept_ = None

        self.plotting_df = {}

        #self.pipe = \
        #    make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))

        for key, value in kwargs.items():
            setattr(self, key, value)

    def fit(self, X, y):
        self.train_data = X
        self.train_labels = y

        self.avg_loss = []

        # Create a dictionary to hold the results of the fitting
        results_dict = {}
        best_weights = {}

        # If input labels are 0/1 then make sure to convert labels to -1 and 1
        # for learning with the logistic loss.
        self.train_labels = np.where(self.train_labels==1, 1, -1)
```

```
        # Calculate folds
        fold_indicies = []

        self.plotting_dict = {
            "max_iterations": [],
            "avg_loss": []
        }

        # Pick random entries for validation/subtrain
        fold_vec = np.random.randint(low=0,
                                     high=self.num_folds,
                                     size=self.train_labels.size)

        # for each fold,
        for fold_number in range(self.num_folds):
            subtrain_indicies = []
            validation_indicies = []
            # check if index goes into subtrain or validation list
            for index in range(len(self.train_data)):
                if fold_vec[index] == fold_number:
                    validation_indicies.append(index)
                else:
                    subtrain_indicies.append(index)

            fold_indicies.append([subtrain_indicies, validation_indicies])

        # Loop over the folds
        for foldnum, indicies in enumerate(fold_indicies):
            # Get indicies of data chosen for this fold
            index_dict = dict(zip(["subtrain", "validation"], indicies))
            set_data_dict = {}

            # Dictionary for test and train data
            for set_name, index_vec in index_dict.items():
                set_data_dict[set_name] = {
                    "X":self.train_data[index_vec],
                    "y":self.train_labels[index_vec]
                }



            # Define a variable called scaled_mat which has
            subtrain_data = set_data_dict["subtrain"]['X']
            subtrain_labels = set_data_dict["subtrain"]['y']
```

```
        scaled_mat = subtrain_data

        # (1) filtered/removed any zero variance features
        #non_variant_indicies = \
    #        np.argwhere(np.all(scaled_mat[..., :] == 0, axis=0))

        #scaled_mat = np.delete(scaled_mat,
        #                       non_variant_indicies,
        #                       axis=1)

        # (2) scaled any other features
        # self.pipe.fit(scaled_mat, self.train_labels)

        # (3) and an extra column of ones (for learning the intercept).
        #intercept_col = np.ones((len(scaled_mat), 1))
        #scaled_mat = np.append(scaled_mat,
        #                       intercept_col,
        #                       axis=1)

        # Initialize an weight vector with size equal to the number of
columns
        # in scaled_mat.
        nrow, ncol = scaled_mat.shape

        learn_features = np.column_stack([
            np.repeat(1, nrow),
            scaled_mat
        ])

        weight_vec = np.zeros(ncol+1)

        #learn_features = learn_features[:,0]

        subtrain_mean = subtrain_data.mean(axis=0)
        subtrain_sd = np.sqrt(subtrain_data.var(axis=0))

        # Then use a for loop from 0 to max_iterations to iteratively compute
        # linear model parameters that minimize the average logistic loss
over
        #the subtrain data.
        min_loss = np.array([10])
        best_iter = 0
        best_coef = weight_vec
```

```
        avg_iter_loss = []

        # Loop for each of the max iterations
        for index in range(self.max_iterations):
            # Calculate prediction and log loss
            pred_vec = np.matmul(learn_features, weight_vec)
            log_loss = np.ma.log(1+np.exp(-subtrain_labels * pred_vec))
            #print("iteration=%d log_loss=%s"%(index,log_loss.mean()))
            grad_loss_pred = -subtrain_labels / \
                                (1+np.exp(subtrain_labels * pred_vec))
            grad_loss_pred = grad_loss_pred
            grad_loss_weight_mat = grad_loss_pred * learn_features.T
            grad_vec = grad_loss_weight_mat.sum(axis=1)
            weight_vec -= self.step_size * grad_vec
            # get the smallest log loss
            if( not np.isinf(log_loss.mean()) <= min_loss.mean() ):
                min_loss = log_loss
                best_iter = index
                best_coef = weight_vec
            # build list of loss values
            avg_iter_loss.append(log_loss.mean())

        # save best stuff from each pass
        results_dict[best_iter] = min_loss.mean()
        best_weights[best_iter] = best_coef
        self.avg_loss.append(avg_iter_loss)

    # get single best weight and intercept
    best_result = max(results_dict, key = results_dict.get)
    self.coef_ = best_weights[best_result][1:]
    self.intercept_ = best_weights[best_result][0]

    # these get saved for plotting
    self.avg_loss = np.asarray(self.avg_loss)
    self.avg_loss = self.avg_loss.mean(axis=0)

    # At the end of the algorithm you should save the learned
    # weights/intercept (on original scale) as the coef_ and intercept_
    # attributes of the class (values should be similar to attributes of
    # LogisticRegression class in scikit-learn).

def decision_function(self, X):
    # Implement a decision_function(X) method which uses the learned weights
    # and intercept to compute a real-valued score (larger for more likely
    # to be predicted positive)
```

```python
        # use best coef and inter to build result
        pred_vec = np.matmul(X, self.coef_) + self.intercept_

        return pred_vec

    def predict(self, test_features):
        # Implement a predict(X) method which uses np.where to threshold the
        # predicted values from decision_function, and obtain a vector of
        # predicted classes (1 if predicted value is positive, 0 otherwise).
        pred_vec = self.decision_function(test_features)

        # positive values are 1, anything else is 0
        pred_vec[pred_vec > 0] = 1
        pred_vec[pred_vec <= 0] = 0

        # predicted values using either scaled or unscaled features agree:
        # print(pred_vec)
        return( pred_vec )

# <-- END INITIALIZATION -->

# <-- BEGIN FUNCTIONS -->
# FUNCTION: MAIN
#   Description  : Main driver for Assignment Three
#   Inputs       : None
#   Outputs      : PlotNine graphs saved to program directory
#   Dependencies : build_image_df_from_dataframe
def main():
    # Display the title
    print("\nCS 499: Homework 4 Program Start")
    print("================================\n")

    # Suppress annoying plotnine warnings
    warnings.filterwarnings('ignore')

    # Download data files
    download_data_file(spam_data_file, spam_data_url, spam_file_path)
    download_data_file(ziptrain_file, ziptrain_url, ziptrain_file_path)
    download_data_file(ziptest_file, ziptest_url, ziptest_file_path)

    # Open each dataset as a pandas dataframe
    spam_df = pd.read_csv(spam_data_file, header=None, sep=" ")
    zip_train_df = pd.read_csv(ziptrain_file, header=None, sep=" ")
    zip_test_df = pd.read_csv(ziptest_file, header=None, sep=" ")
```

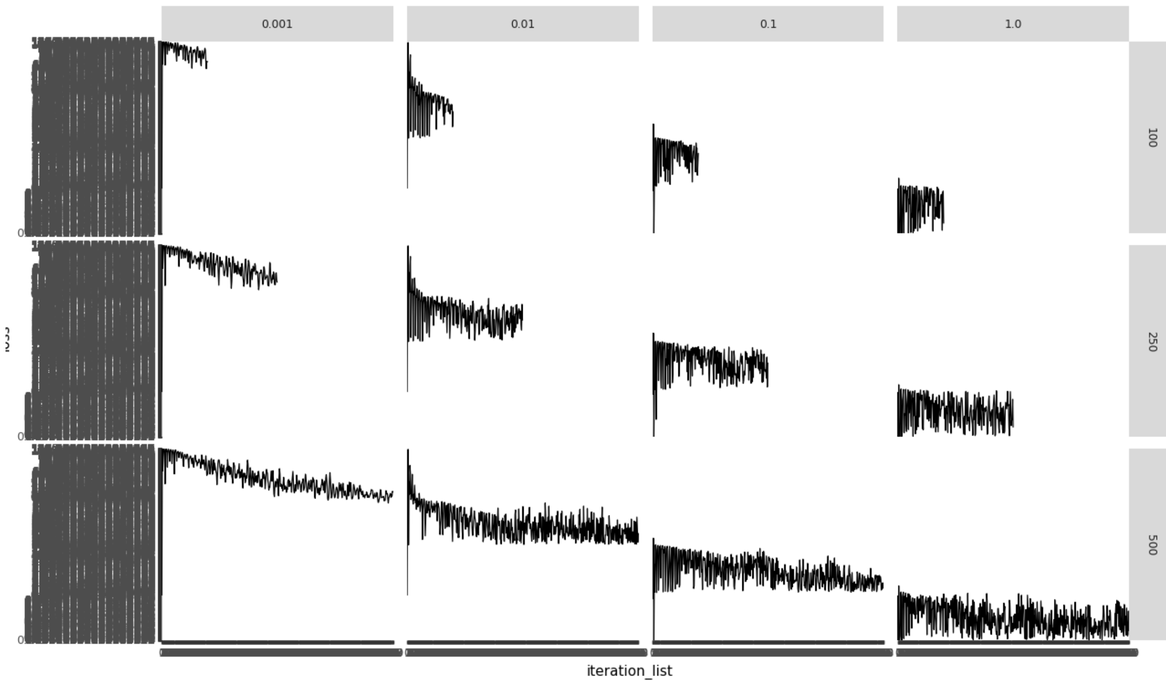
```
# Concat the two zip dataframes together
zip_df = pd.concat([zip_train_df, zip_test_df])

# Drop rows of dataframes where the label is not ( 0 or 1)
zip_df[0] = zip_df[0].astype(int)
zip_df = zip_df[zip_df[0].isin([0, 1])]
# Drop empty col from zip dataframe
zip_df = zip_df.drop(columns=[zip_empty_col])

# Create label vectors
zip_labels = zip_df[0]
spam_labels = spam_df[spam_label_col]

# Create numpy data
zip_data = zip_df.iloc[:, 1:256].to_numpy()
spam_data = spam_df.iloc[:, :56].to_numpy()

pipe.fit(spam_data, spam_labels)


# Create data dictionary
data_dict = {
    'spam' : [spam_data, spam_labels],
    'zip' : [zip_data, zip_labels]
}

# Loop through each data set
for data_set, (input_data, output_array) in data_dict.items():
    # Output message for logging
    print("Working on set: " + str(data_set))
    current_set = str(data_set)
    # Scale the data set

    # Loop over each fold for each data set
    for foldnum, indicies in enumerate(kf.split(input_data)):
        print("Fold #" + str(foldnum))
        # Set up input data structs
        index_dict = dict(zip(["train", "test"], indicies))
        param_dicts = [{'n_neighbors':[x]} for x in range(1, 21)]
        logreg_param_dicts = \
            [{'max_iterations':max_it, 'step_size':steps} \
                for max_it in [100, 1000, 2000] \
                for steps in [1, 0.1, 0.01, 0.001]]

        logreg_param_nosteps_dicts = \
```

```python
    [{'max_iterations':max_it} \
        for max_it in [100, 1000, 2000]]

# Establish different models
clf = GridSearchCV(KNeighborsClassifier(), param_dicts)
linear_model = sklearn.linear_model.LogisticRegressionCV(cv=5)
RegressionCV = MyCV(estimator=MyLogReg,
                    param_grid=logreg_param_dicts,
                    cv=CV_VAL)
RegressionCVNoSteps = MyCV(estimator=MyLogReg,
                    param_grid=logreg_param_nosteps_dicts,
                    cv=CV_VAL)

# Creating dictionary with input and outputs
set_data_dict = {}
for set_name, index_vec in index_dict.items():
    set_data_dict[set_name] = {
        "X":input_data[index_vec],
        "y":output_array.iloc[index_vec].reset_index(drop=True)
    }

# Train the models with given data
clf.fit(**set_data_dict["train"])
linear_model.fit(**set_data_dict["train"])
RegressionCV.fit(**set_data_dict["train"])
RegressionCVNoSteps.fit(**set_data_dict["train"])

# Get most common output from outputs for featureless set
most_common_element = mode(set_data_dict["train"]['y'])

# Get results
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 + MyLogReg (No Step Size Training)": \
        RegressionCVNoSteps.predict(set_data_dict["test"]["X"]),
    "MyCV + MyLogReg": \
        RegressionCV.predict(set_data_dict["test"]["X"]),
    "Featureless":most_common_element
}
```

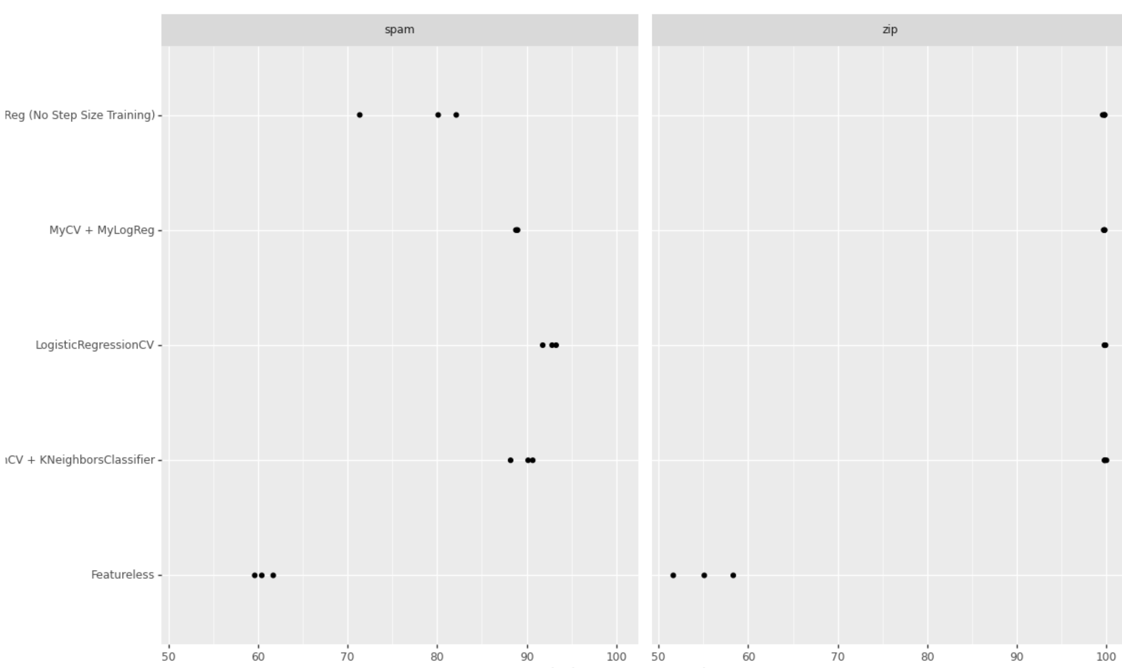
```python
            # Build results dataframe for each algo/fold
            for algorithm, pred_vec in pred_dict.items():
                test_acc_dict = {
                    "test_accuracy_percent":(
                        pred_vec == set_data_dict["test"]["y"]).mean()*100,
                    "data_set":data_set,
                    "fold_id":foldnum,
                    "algorithm":algorithm
                }
                test_acc_df_list.append(pd.DataFrame(test_acc_dict, index=[0]))

    # Build accuracy results dataframe
    test_acc_df = pd.concat(test_acc_df_list)

    # Print results
    print("\n")
    print(test_acc_df)

    # Plot results
    plot = (p9.ggplot(test_acc_df,
                        p9.aes(x='test_accuracy_percent',
                        y='algorithm'))
                + p9.facet_grid('. ~ data_set')
                + p9.geom_point())

    print(plot)

    print("\nCS 499: Homework 4 Program End")
    print("==============================\n")

# FUNCTION : DOWNLOAD_DATA_FILE
#   Description: Downloads file from source, if not already downloaded
#   Inputs:
#       - file      : Name of file to download
#       - file_url  : URL of file
#       - file_path : Absolute path of location to download file to.
#                     Defaults to the local directory of this program.
#   Outputs: None
def download_data_file(file, file_url, file_path):
    # Check for data file. If not found, download
    if not os.path.isfile(file_path):
        try:
            print("Getting file: " + str(file) + "...\n")
            urllib.request.urlretrieve(file_url, file_path)
            print("File downloaded.\n")
```

```
        except(error):
            print(error)
    else:
        print("File: " + str(file) + " is already downloaded.\n")

# Launch main
if __name__ == "__main__":
    main()
```

Program Output:



(Figure 1) Attempt at plotting loss vs iterations, facetted by step size and max iteration



(Figure 2) Test accuracy for Zip and Spam training set. Step size training clearly increases the training accuracy.

```
CS 499: Homework 4 Program Start
================================

File: spam.data is already downloaded.

File: zip.train.gz is already downloaded.

File: zip.test.gz is already downloaded.

Working on set: spam
Fold #0
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
Fold #1
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
Fold #2
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
```

```
Working on set: zip
Fold #0
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
Fold #1
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
Fold #2
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
(MyCV) Subfold #0
(MyCV) Subfold #1
(MyCV) Subfold #2
(MyCV) Subfold #3
(MyCV) Subfold #4
```

```
   test_accuracy_percent data_set  fold_id                               algorithm
0              90.156454     spam        0      GridSearchCV + KNeighborsClassifier
0              92.829205     spam        0                     LogisticRegressionCV
0              82.138201     spam        0  MyCV + MyLogReg (No Step Size Training)
0              88.983051     spam        0                          MyCV + MyLogReg
0              59.647979     spam        0                              Featureless
0              88.200782     spam        1      GridSearchCV + KNeighborsClassifier
0              91.786180     spam        1                     LogisticRegressionCV
0              80.117340     spam        1  MyCV + MyLogReg (No Step Size Training)
0              88.787484     spam        1                          MyCV + MyLogReg
0              60.430248     spam        1                              Featureless
0              90.671885     spam        2      GridSearchCV + KNeighborsClassifier
0              93.281148     spam        2                     LogisticRegressionCV
0              71.363340     spam        2  MyCV + MyLogReg (No Step Size Training)
0              88.910633     spam        2                          MyCV + MyLogReg
0              61.709067     spam        2                              Featureless
0              99.787460      zip        0      GridSearchCV + KNeighborsClassifier
0              99.787460      zip        0                     LogisticRegressionCV
0              99.574920      zip        0  MyCV + MyLogReg (No Step Size Training)
0              99.681190      zip        0                          MyCV + MyLogReg
0              58.342189      zip        0                              Featureless
0              99.787460      zip        1      GridSearchCV + KNeighborsClassifier
0              99.787460      zip        1                     LogisticRegressionCV
0              99.787460      zip        1  MyCV + MyLogReg (No Step Size Training)
0              99.787460      zip        1                          MyCV + MyLogReg
0              51.647184      zip        1                              Featureless
0             100.000000      zip        2      GridSearchCV + KNeighborsClassifier
0              99.893617      zip        2                     LogisticRegressionCV
0              99.787234      zip        2  MyCV + MyLogReg (No Step Size Training)
0              99.787234      zip        2                          MyCV + MyLogReg
0              55.106383      zip        2                              Featureless


CS 499: Homework 4 Program End
==============================
```

Question Answers / Commentary:

I was unable to complete some of the major requirements of this assignment. While the Logistic Regression model and accompanying CV were able to correctly predict the correct outcomes with great accuracy, I was not able to create the plots of Loss vs. Iterations. Additionally, I had difficulty configuring GGPlot to correctly display axis labels and information in an aesthetic manner.

For the extra credit portion of the assignment, I added the step size training parameters into my CV. Because it was correctly built last assignment to allow for this functionality, it can take any number of parameters without having to be rebuilt, and has K-fold cross validation built in. I plotted the accuracy of the CV with and without step size training, and step size training clearly increases accuracy.

Overall, I would cold the model a success. Despite not being able to correctly plot the training/validation loss over time, the end results speak for themselves.