Assignment Seven

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 torch

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 sklearn.model_selection import train_test_split


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)
```

```python
# - Zip data (Test) variables
ziptest_url = "https://hastie.su.domains/ElemStatLearn/datasets/zip.test.gz"
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
MAX_EPOCHS_VAR = 250
BATCH_SIZE_VAR = 32
NUM_LAYERS_VAR = 3

# MISC. VARIABLES
kf = KFold(n_splits=2, shuffle=True, random_state=1)
test_acc_df_list = []
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=1000))

#CLASS DEFINITIONS
class TorchModel(torch.nn.Module):
    def __init__(self, units_per_layer):
        super(TorchModel, self).__init__()
        seq_args = []
        for layer_i in range(len(units_per_layer)-1):
            units_in = units_per_layer[layer_i]
            units_out = units_per_layer[layer_i+1]
            seq_args.append(
                torch.nn.Linear(units_in, units_out))
            if layer_i != len(units_per_layer)-2:
                seq_args.append(torch.nn.ReLU())
        self.stack = torch.nn.Sequential(*seq_args)

    def forward(self, feature_mat):
        return self.stack(feature_mat)

    def getitem(self, item):
        weights, intercept = [p for p in self.weight_vec.parameters()]
        return weights.data[0][item]

class TorchLearner(torch.nn.Module):
    def __init__( self, **kwargs ):
        super(TorchLearner, self).__init__()
```

```python
        kwargs.setdefault("step_size", 0.0001) # trained through cv
        kwargs.setdefault("max_epochs", 10)
        kwargs.setdefault("batch_size", 2)
        kwargs.setdefault("units_per_layer")

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

        self.loss_df_list = []

        self.train_data = None
        self.train_labels = None
        self.coef_ = None
        self.intercept_ = None

        self.model = TorchModel( self.units_per_layer )
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=0.1)
        self.loss_fun = torch.nn.BCEWithLogitsLoss()

        self.loss_df = {}

    def take_step(self, X, y):
        self.optimizer.zero_grad()
        pred_tensor = self.model.forward(X.float()).reshape(len(y))
        loss_tensor = self.loss_fun(pred_tensor, y.float())
        loss_tensor.backward()
        self.optimizer.step()

    def fit(self, X, y):
        np.random.seed(1)
        n_folds = 5
        fold_vec = np.random.randint(low=0, high=n_folds, size=y.size)
        validation_fold = 0
        is_set_dict = {
            "validation":fold_vec == validation_fold,
            "subtrain":fold_vec != validation_fold,
        }

        set_features = {}
        set_labels = {}
        for set_name, is_set in is_set_dict.items():
            set_features[set_name] = X[is_set,:]
            set_labels[set_name] = y[is_set]
        {set_name:array.shape for set_name, array in set_features.items()}
```

```python
        ds = CSV(X, y)
        dl = torch.utils.data.DataLoader(
            ds, batch_size=2, shuffle=True)
        for batch_features, batch_labels in dl:
            pass

        for epoch in range(self.max_epochs):
            for batch_features, batch_labels in dl:
                self.take_step(batch_features, batch_labels)

            for set_name in set_features:
                set_X = set_features[set_name]
                set_y = set_labels[set_name]
                set_X_tensor = torch.from_numpy(set_X).float()
                set_y_tensor = torch.from_numpy(set_y)
                set_pred = self.model(set_X_tensor).reshape(len(set_y))
                set_loss = self.loss_fun(set_pred, set_y_tensor.float())

                pred_vec = set_pred.detach().numpy()
                pred_vec[pred_vec > 0] = 1
                pred_vec[pred_vec <= 0] = 0

                self.loss_df_list.append(pd.DataFrame({
                    "set_name":set_name,
                    "loss":float(set_loss),
                    "epoch":epoch,
                    "test_accuracy_percent":(
                        pred_vec == set_y).mean()*100,
                }, index=[0]))

        self.loss_df = pd.concat(self.loss_df_list)
        #self.pred_vec = pred_vec

    def decision_function(self, X):
        pred_vec = np.matmul(X,
                             self.model.stack[-1].weight[0].detach().numpy()) \
                             + self.model.stack[-1].bias[0].detach().numpy()

        return pred_vec

    def predict(self, X):
        pred_vec = self.decision_function(X)

        pred_vec[pred_vec > 0] = 1
        pred_vec[pred_vec <= 0] = 0
```

```
        pred_vec = np.stack(pred_vec, axis=0)

        return( pred_vec )

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

        kwargs.setdefault("num_folds", 3)

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

        self.estimator = self.estimator(
            units_per_layer=self.param_grid[0]['units_per_layer'])
        self.best_model = None

        self.printing_list = []

        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,
                                     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[index_vec]
                }

            # 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_list = 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

                self.printing_list.append(self.estimator.loss_df)


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

        self.plotting_df = pd.concat(self.printing_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 LinearModel(torch.nn.Module):
    def __init__(self, num_inputs):
        super(LinearModel, self).__init__()
        self.weight_vec = torch.nn.Linear(num_inputs, 1)
    def getitem(self, item):
```

```
        weights, intercept = [p for p in self.weight_vec.parameters()]
        return weights.data[0][item]
    def forward(self, feature_mat):
        return self.weight_vec(feature_mat)

class CSV(torch.utils.data.Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels
    def __getitem__(self, item):
        return self.features[item,:], self.labels[item]
    def __len__(self):
        return len(self.labels)

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,
                                     high=self.num_folds,
```

```python
                                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[index_vec]
                }

            # 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)
```

```python
        # 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)

        # 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 )

class InitialNode:
    def __init__(self, value, name):
        self.value = value
        self.name = name
    def backward(self):
        pass
        #print("backward from "+self.name)

class Operation:
    def __init__(self, *node_list):
        for index in range(len(node_list)):
            setattr(self, self.input_names[index], node_list[index])
        self.value = self.get_value()
    def backward(self):
        grad_tuple = self.gradient()
        # store each gradient in the corresponding parent_node.grad
        for index in range(len(grad_tuple)):
            parent_node = getattr(self, self.input_names[index])
            parent_node.grad = grad_tuple[index]
            parent_node.backward()

class mean_logistic_loss(Operation):
    input_names = ["pred_vec", "subtrain_labels"]
    def get_value(self):
        self.pred_vec.value = torch.reshape(torch.tensor(self.pred_vec.value), (-
1,)).detach().numpy()
        self.subtrain_labels.value = self.subtrain_labels.value.detach().numpy()
```

```
        return np.mean(np.log(
            1+np.exp(-self.subtrain_labels.value * self.pred_vec.value)))
    def gradient(self):
        return [
            -self.subtrain_labels.value/(
                1+np.exp(self.subtrain_labels.value * self.pred_vec.value)
            )/len(self.subtrain_labels.value),
            "gradient with respect to labels"]

class mm(Operation):
    input_names = ["features", "weights"]
    def get_value(self):
        return np.matmul(self.features.value, self.weights.value)
    def gradient(self):
        self.grad = np.asarray(self.grad).reshape((-1, 1))
        self.weights.value = np.asarray(self.weights.value).reshape((-1, 1))

        return [
            np.matmul(self.grad, self.weights.value.T),
            np.matmul(self.features.value.T, self.grad)]

class relu(Operation):
    input_names = ["features_before_activation"]
    def get_value(self):
        return np.where(
            self.features_before_activation.value > 0,
            self.features_before_activation.value, 0)
    def gradient(self):
        return [
            np.where(self.features_before_activation < 0, 0, self.grad)]

class AutoMLP:
    def __init__(self, max_epochs, batch_size, step_size,
                 units_per_layer, num_layers):
        """Store hyper-parameters as attributes, then initialize
        weight_node_list attribute to a list of InitialNode instances."""
        self.max_epochs = max_epochs
        self.batch_size = batch_size
        self.step_size = step_size
        self.units_per_layer = units_per_layer
        self.num_layers = num_layers

        self.lowest_loss = 10000
        self.best_epochs = 0
```

```
    self.weight_node_list = \
     np.repeat([InitialNode(torch.tensor(np.random.randn(units_per_layer)),
                "layer")], self.num_layers)

def get_pred_node(self, X):
    """return node of predicted values for feature matrix X"""
    feature_node = InitialNode(X, "feature")
    for weight_node in self.weight_node_list:
        prediction_node = mm(feature_node, weight_node)

    return prediction_node

def take_step(self, X, y):
    """call get_pred_node, then instantiate logistic_loss, call its
    backward method to compute gradients, then for loop over
    weight_node_list (one iteration of gradient descent).
    """
    pred_node = self.get_pred_node(X)
    label_node = InitialNode(y, "label")
    self.loss_node = mean_logistic_loss(pred_node, label_node)
    self.loss_node.backward()

    for weight_node in self.weight_node_list:
        weight_node.value -= self.step_size * np.asarray(weight_node.grad)

def fit(self, X, y):
    """Gradient descent learning of weights"""
    ds = CSV(X, y)
    dl = torch.utils.data.DataLoader(ds, batch_size=2, shuffle=True)
    loss_df_list = []
    for epoch in range(self.max_epochs):
        for batch_features, batch_labels in dl:
            self.take_step(batch_features, batch_labels)

            loss_df_list.append(
                pd.DataFrame(
                    {
                        "epoch":epoch,
                        "loss":float(self.loss_node.value),
                    }, index=[0])
            )#subtrain/validation loss using current weights.

            if (self.loss_node.value < self.lowest_loss):
                self.lowest_loss = self.loss_node.value
```

```python
                self.best_epochs = epoch

        try:
            self.loss_df = pd.concat(loss_df_list)
        except:
            pass

    def decision_function(self, X):
        """Return numpy vector of predicted scores"""
        pred_vec = self.get_pred_node(X)
        pred_vec = relu(pred_vec)

        return pred_vec

    def predict(self, X):
        """Return numpy vector of predicted classes"""
        pred_vec = self.decision_function(X).value

        pred_vec[pred_vec > 0] = 1
        pred_vec[pred_vec <= 0] = 0

        pred_vec = pred_vec.flatten()

        return( pred_vec )

class LinearAutoMLP:
    def __init__(self, max_epochs, batch_size, step_size, units_per_layer):
        """Store hyper-parameters as attributes, then initialize
        weight_node_list attribute to a list of InitialNode instances."""
        self.max_epochs = max_epochs
        self.batch_size = batch_size
        self.step_size = step_size
        self.units_per_layer = units_per_layer

        self.lowest_loss = 10000
        self.best_epochs = 0

        self.weight_node_list = \
         [InitialNode(torch.tensor(np.random.randn(units_per_layer)), "layer1")]

    def get_pred_node(self, X):
        """return node of predicted values for feature matrix X"""
        feature_node = InitialNode(X, "feature")
        for weight_node in self.weight_node_list:
            prediction_node = mm(feature_node, weight_node)
```

```python
        return prediction_node

    def take_step(self, X, y):
        """call get_pred_node, then instantiate logistic_loss, call its
        backward method to compute gradients, then for loop over
        weight_node_list (one iteration of gradient descent).
        """
        pred_node = self.get_pred_node(X)
        label_node = InitialNode(y, "label")
        self.loss_node = mean_logistic_loss(pred_node, label_node)
        self.loss_node.backward()

        for weight_node in self.weight_node_list:
            weight_node.value -= self.step_size * np.asarray(weight_node.grad)

    def fit(self, X, y):
        """Gradient descent learning of weights"""
        ds = CSV(X, y)
        dl = torch.utils.data.DataLoader(ds, batch_size=2, shuffle=True)
        loss_df_list = []
        for epoch in range(self.max_epochs):
            for batch_features, batch_labels in dl:
                self.take_step(batch_features, batch_labels)

                loss_df_list.append(
                    pd.DataFrame(
                        {
                            "epoch":epoch,
                            "loss":float(self.loss_node.value),
                        }, index=[0])
                )#subtrain/validation loss using current weights.

                if (self.loss_node.value < self.lowest_loss):
                    self.lowest_loss = self.loss_node.value
                    self.best_epochs = epoch

        try:
            self.loss_df = pd.concat(loss_df_list)
        except:
            pass

    def decision_function(self, X):
        """Return numpy vector of predicted scores"""
        pred_vec = self.get_pred_node(X)
```

```python
        pred_vec = relu(pred_vec)

        return pred_vec

    def predict(self, X):
        """Return numpy vector of predicted classes"""
        pred_vec = self.decision_function(X).value

        pred_vec[pred_vec > 0] = 1
        pred_vec[pred_vec <= 0] = 0

        pred_vec = pred_vec.flatten()

        return( pred_vec )

class LinearAutoGradLearnerCV:
    def __init__(self, max_epochs, batch_size, step_size, units_per_layer):
        self.model = LinearAutoMLP(max_epochs, batch_size, step_size,
                                   units_per_layer)

    def fit(self, X, y):
        """cross-validation for selecting the best number of epochs"""
        print("(Linear AutoCV)")

        best_epochs = self.model.best_epochs

        self.model.validation_data = X
        self.model.fit(X, y)
        self.model.max_epochs = best_epochs
        self.model.fit(X, y)

        self.loss_df = self.model.loss_df

    def predict(self, X):
        return self.model.predict(X)

class AutoGradLearnerCV:
    def __init__(self, max_epochs, batch_size, step_size, units_per_layer,
                 num_layers):
        self.model = AutoMLP(max_epochs, batch_size, step_size, units_per_layer,
                             num_layers)

    def fit(self, X, y):
        """cross-validation for selecting the best number of epochs"""
        print("(Deep AutoCV)")
```

```python
        best_epochs = self.model.best_epochs

        self.model.validation_data = X
        self.model.fit(X, y)
        self.model.max_epochs = best_epochs
        self.model.fit(X, y)

        self.loss_df = self.model.loss_df

    def predict(self, X):
        return self.model.predict(X)

# <-- 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 7 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])]
```

```python
# Drop empty col from zip dataframe
zip_df = zip_df.drop(columns=[zip_empty_col])

spam_features = spam_df.iloc[:,:-1].to_numpy()
spam_labels = spam_df.iloc[:,-1].to_numpy()

# 1. feature scaling.
spam_mean = spam_features.mean(axis=0)
spam_sd = np.sqrt(spam_features.var(axis=0))
spam_features = (spam_features-spam_mean)/spam_sd
spam_features.mean(axis=0)
spam_features.var(axis=0)

zip_features = zip_df.iloc[:,:-1].to_numpy()
zip_labels = zip_df[0].to_numpy()

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

final_df_list = []
final_deep_print_list = []
final_linear_print_list = []

final_deep_df = pd.DataFrame()
final_linear_df = pd.DataFrame()

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

    #torchLean = TorchLearner(units_per_layer=(ncol, 100, 10, 100, 1) )
    #torchLean.fit( input_data, output_array )

    # 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

        nrow, ncol = input_data.shape
        index_dict = dict(zip(["train", "test"], indicies))
```

```python
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 [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]]

logreg_param_nosteps_dicts = \
    [{'max_iterations':max_it} \
        for max_it in [100, 1000, 2000]]

deep_param_dict = \
    [{'units_per_layer': (ncol, 100, 10, 100, ncol, 1),
      'max_epochs':max_ep} \
         for max_ep in [10]]

linear_param_dict = \
    [{'units_per_layer': (ncol, 1),
      'max_epochs':max_ep} \
         for max_ep in [10]]

if data_set == "spam":
    STEP_SIZE_VAR = 0.001
if data_set == "zip":
    STEP_SIZE_VAR = 0.0001

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)
linearAutoCV = LinearAutoGradLearnerCV(MAX_EPOCHS_VAR,
                                       BATCH_SIZE_VAR,
                                       STEP_SIZE_VAR, ncol)
deepAutoCV = AutoGradLearnerCV(MAX_EPOCHS_VAR,
                               BATCH_SIZE_VAR,
                               STEP_SIZE_VAR, ncol,
                               NUM_LAYERS_VAR)
DeepTorchCV = TorchLearnerCV(estimator=TorchLearner,
                    param_grid=deep_param_dict,
                    max_epochs=25)
LinearTorchCV = TorchLearnerCV(estimator=TorchLearner,
                    param_grid=linear_param_dict,
                    cv=CV_VAL)
```

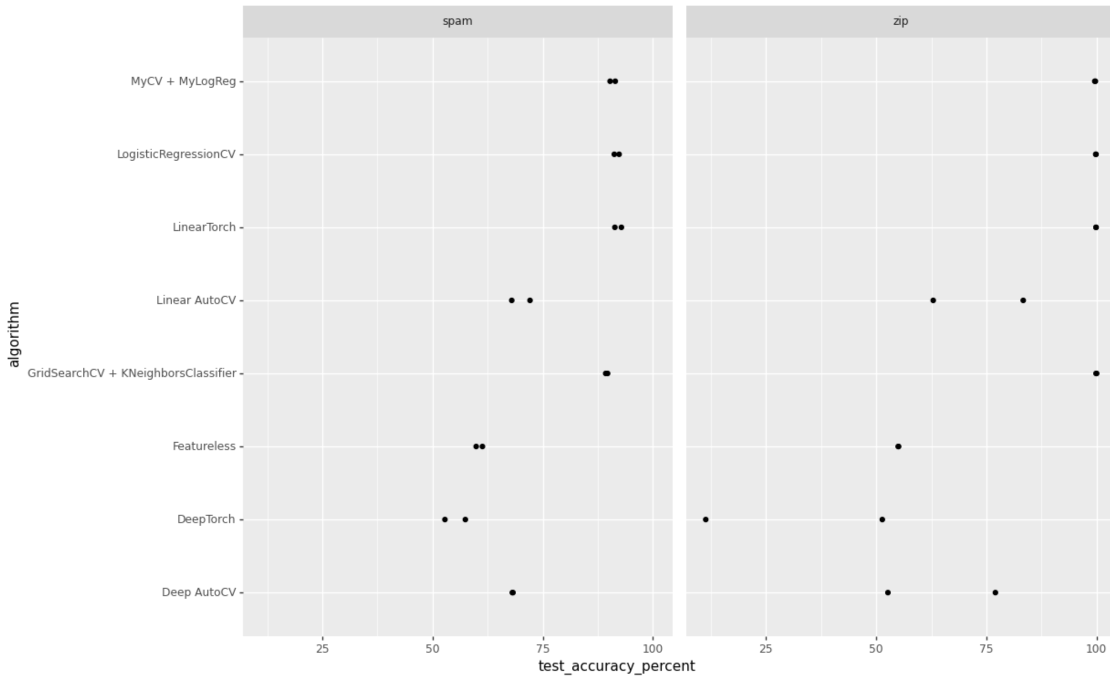
```python
        # Creating dictionary with input and outputs
        set_data_dict = {}
        for set_name, index_vec in index_dict.items():
            set_data_dict[set_name] = {
                "X":input_data[index_vec],
                "y":output_array[index_vec]
            }

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

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

        buffer_df = deepAutoCV.loss_df.groupby("epoch").mean()
        buffer_df['subfold'] = foldnum
        buffer_df['set'] = data_set
        final_deep_print_list.append(buffer_df)

        buffer_df = linearAutoCV.loss_df.groupby("epoch").mean()
        buffer_df['subfold'] = foldnum
        buffer_df['set'] = data_set
        final_linear_print_list.append(buffer_df)

        # 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"]),
            "Linear AutoCV": \
                linearAutoCV.predict(set_data_dict["test"]["X"]),
            "Deep AutoCV": \
                deepAutoCV.predict(set_data_dict["test"]["X"]),
            "DeepTorch": \
                DeepTorchCV.predict(set_data_dict["test"]["X"]),
            "LinearTorch": \
```

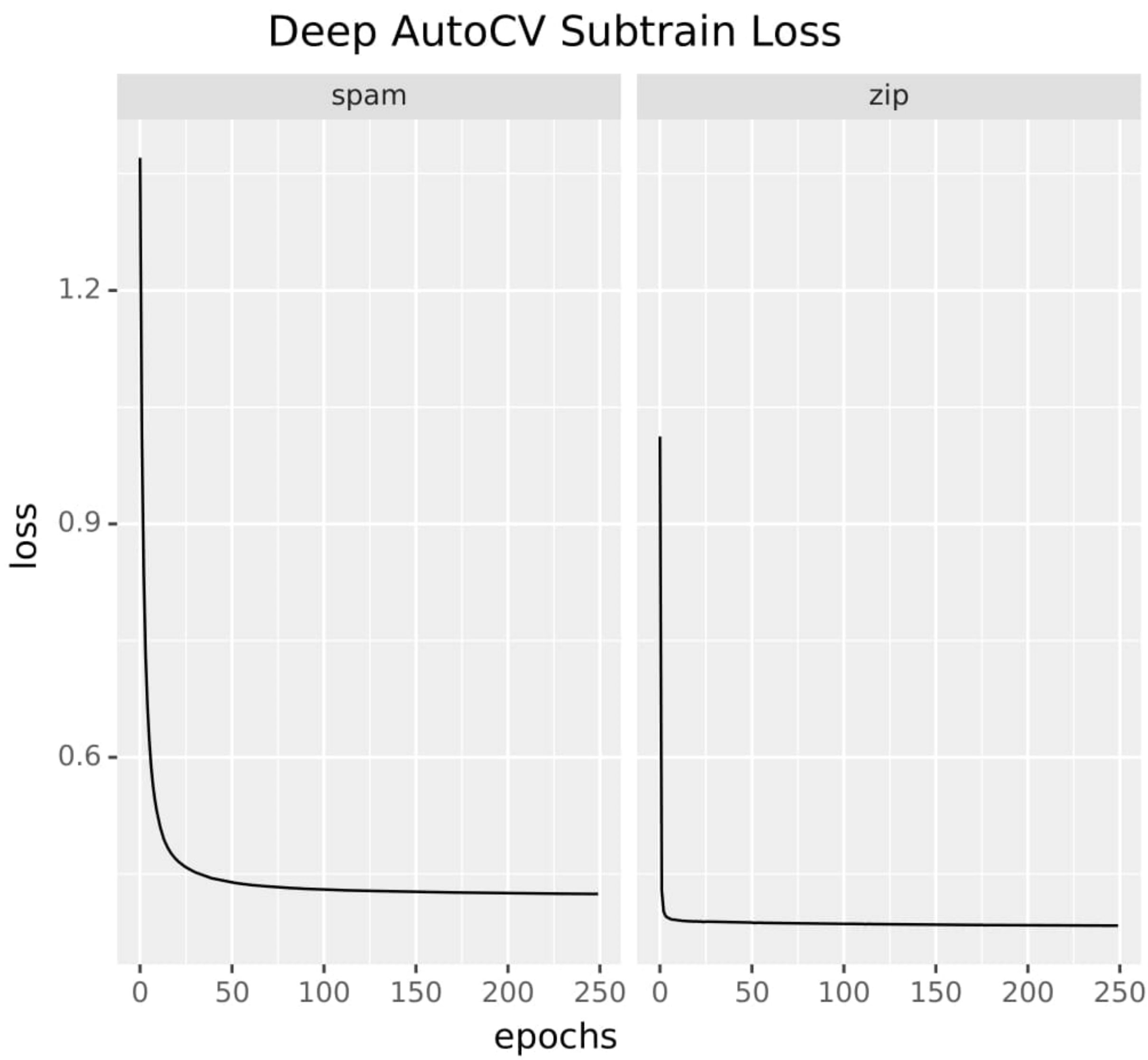
```
                LinearTorchCV.predict(set_data_dict["test"]["X"]),
            "MyCV + MyLogReg": \
                RegressionCV.predict(set_data_dict["test"]["X"]),
            "Featureless":most_common_element
        }

        # 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]))

    final_deep_df = pd.concat(final_deep_print_list)
    final_linear_df = pd.concat(final_deep_print_list)

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

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

    # Plot results
    plot = (p9.ggplot(test_acc_df,
                    p9.aes(x='test_accuracy_percent',
                    y='algorithm'))
                + p9.facet_grid('. ~ data_set')
                + p9.geom_point()
                + p9.theme(subplots_adjust={'left': 0.2}))

    epoch_vec = np.arange(MAX_EPOCHS_VAR)
    epoch_vec = np.tile(epoch_vec, 2)
    epoch_vec = epoch_vec.flatten()

    final_deep_df = final_deep_df.groupby(['set', 'epoch'],
as_index=False).mean()
    final_deep_df['epochs'] = epoch_vec
    deepplot = (p9.ggplot(final_deep_df,
                    p9.aes(x='epochs',
```

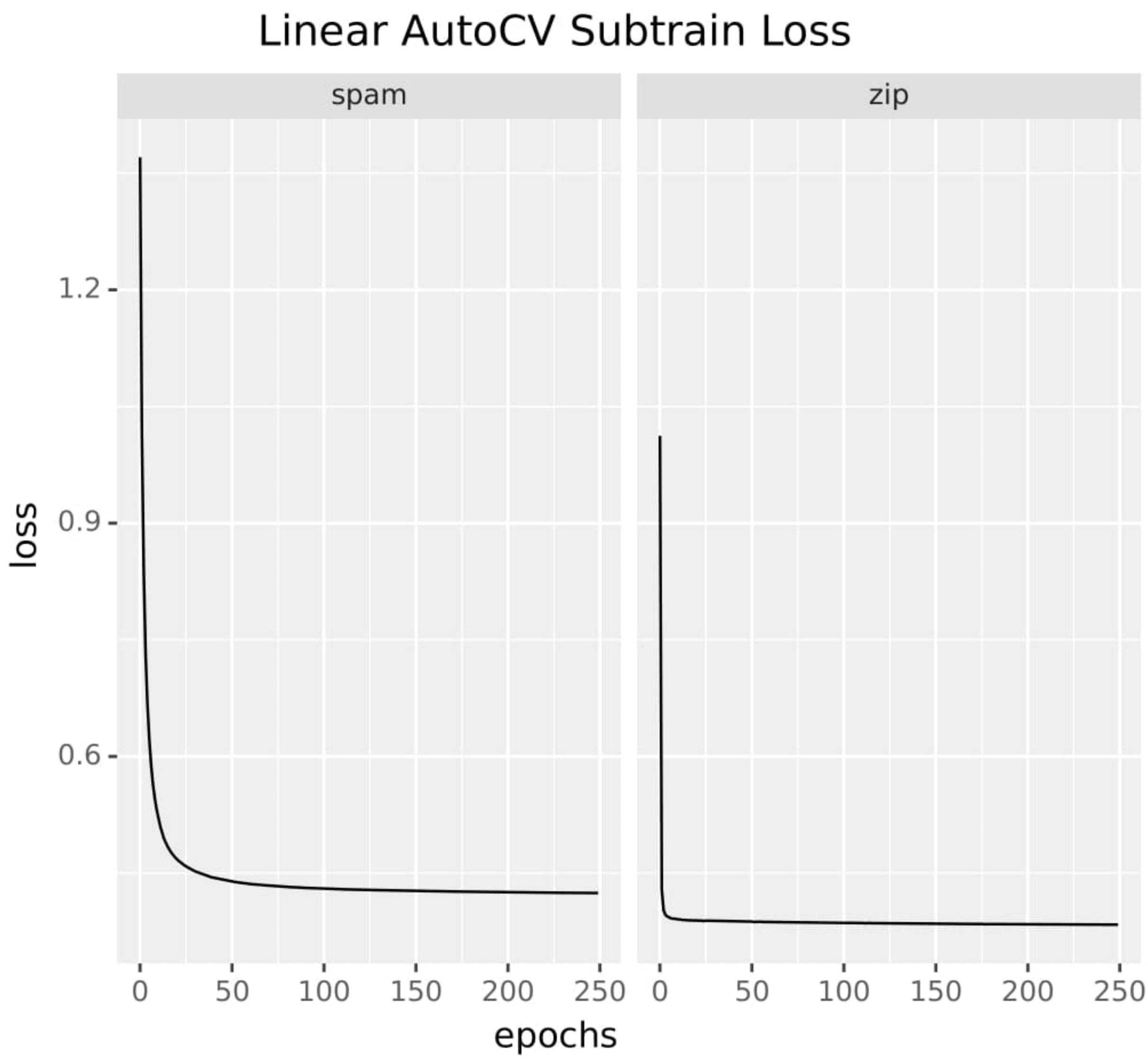
```
                              y='loss'))
                  + p9.facet_grid('. ~ set', scales='free')
                  + p9.geom_line()
                  + p9.theme(subplots_adjust={'left': 0.2})
                  + p9.ggtitle("Deep AutoCV Subtrain Loss"))

    final_linear_df = final_linear_df.groupby(['set', 'epoch'],
as_index=False).mean()
    final_linear_df['epochs'] = epoch_vec
    linearplot = (p9.ggplot(final_linear_df,
                        p9.aes(x='epochs',
                               y='loss'))
                  + p9.facet_grid('. ~ set', scales='free')
                  + p9.geom_line()
                  + p9.theme(subplots_adjust={'left': 0.2})
                  + p9.ggtitle("Linear AutoCV Subtrain Loss"))

    print(plot)
    deepplot.save()
    linearplot.save()

    print("\nCS 499: Homework 7 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:




Deep AutoCV Subtrain Loss
spam
zip
loss
1.2
0.9
0.6
0
50
100
150
200
250
epochs



Linear AutoCV Subtrain Loss
spam
zip
loss
1.2
0.9
0.6
0
50
100
150
200
250
epochs

	test_accuracy_percent	data_set	fold_id	algorithm
0	89.265537	spam	0	GridSearchCV + KNeighborsClassifier
0	92.307692	spam	0	LogisticRegressionCV
0	72.099087	spam	0	Linear AutoCV
0	68.231204	spam	0	Deep AutoCV
0	52.803129	spam	0	DeepTorch
0	92.829205	spam	0	LinearTorch
0	90.265102	spam	0	MyCV + MyLogReg
0	59.887006	spam	0	Featureless
0	89.695652	spam	1	GridSearchCV + KNeighborsClassifier
0	91.217391	spam	1	LogisticRegressionCV
0	67.956522	spam	1	Linear AutoCV
0	68.086957	spam	1	Deep AutoCV
0	57.434783	spam	1	DeepTorch
0	91.347826	spam	1	LinearTorch
0	91.434783	spam	1	MyCV + MyLogReg
0	61.304348	spam	1	Featureless
0	99.858257	zip	0	GridSearchCV + KNeighborsClassifier
0	99.858257	zip	0	LogisticRegressionCV
0	83.345145	zip	0	Linear AutoCV
0	77.037562	zip	0	Deep AutoCV
0	51.381999	zip	0	DeepTorch
0	99.858257	zip	0	LinearTorch
0	99.716513	zip	0	MyCV + MyLogReg
0	55.067328	zip	0	Featureless
0	100.000000	zip	1	GridSearchCV + KNeighborsClassifier
0	99.787385	zip	1	LogisticRegressionCV
0	62.934089	zip	1	Linear AutoCV
0	52.657690	zip	1	Deep AutoCV
0	11.339476	zip	1	DeepTorch
0	99.858257	zip	1	LinearTorch
0	99.574770	zip	1	MyCV + MyLogReg
0	54.996456	zip	1	Featureless

Question Answers / Commentary:

For Assignment 7, I was able to implement a passably acceptable version of a neural network and linear regression model, using the code demos and code skeleton provided in class.

While the models themselves are far from optimal, they do show a level of learning that is able to produce a test accuracy better than a featureless model. I believe that while the underlying network and models are functional, this lack of accuracy comes from a lack of training hyper-parameters such as step size and batch size. Manually tweaking these parameters has only led to marginal increases in accuracy.

My Deep Learning AutoCV implements 3 layers of weight nodes, whereas the Linear Learning AutoCV implements only one. There is a measurable increase in accuracy of about 10% minimum with the deep learning model. Each model was trained using 250 epochs, and step sizes were manually selected for each dataset.

For the extra credit portion of this assignment, I also implemented these new models alongside the models from Assignment 6 and Assignment 4. While there was some improvement over my Torch model, the Assignment 4 Logistic Regression model significantly outperformed this new attempt.