Assignment Thirteen

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 torchvision

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

import math
# <-- END IMPORTS / HEADERS -->

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

# - 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"
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

MAX_EPOCHS_VAR = 100
BATCH_SIZE_VAR = 256
STEP_SIZE_VAR = 0.01
HIDDEN_LAYERS_VAR = 10
CV_VAL = 2
N_FOLDS = 2

global ncol
global n_classes

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

#CLASS DEFINITIONS
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 TorchConvModel(torch.nn.Module):
   def __init__(self, *units_per_layer):
      super(TorchConvModel, self).__init__()
      seq_args = []

      seq_args.append( torch.nn.Conv2d( in_channels=1,
                       out_channels=32,
                       kernel_size=3,
                       stride=3 ) )

      seq_args.append( torch.nn.ReLU() )
```

```
      seq_args.append( torch.nn.Conv2d( in_channels=32,
                       out_channels=64,
                       kernel_size=3,
                       stride=3 ) )

      seq_args.append( torch.nn.ReLU() )

      seq_args.append( torch.nn.MaxPool2d(kernel_size=1, stride=1) )

      seq_args.append( torch.nn.ReLU() )

      seq_args.append( torch.nn.Flatten(start_dim=1) )

      seq_args.append( torch.nn.Linear( 64, 128 ) )

      seq_args.append( torch.nn.ReLU() )

      seq_args.append( torch.nn.Linear( 128, 2 ) )

      self.stack = torch.nn.Sequential(*seq_args)

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

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.float())

class ConvolutionalMLP:
   def __init__(self, **kwargs):
      """Store hyper-parameters, TorchConvModel instance, loss, etc."""
      kwargs.setdefault("max_epochs", 2)
      kwargs.setdefault("batch_size", BATCH_SIZE_VAR)
      kwargs.setdefault("step_size", 0.01)
```

```
        kwargs.setdefault("units_per_layer", ( ncol, 1000, 100, n_classes ) )
        kwargs.setdefault("hidden_layers", 3)
        kwargs.setdefault("opt_name", torch.optim.SGD)
        kwargs.setdefault("opt_params", {'lr':0.1})

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

        units_per_layer = [ncol]
        for L in range(self.hidden_layers):
            units_per_layer.append(100)
        units_per_layer.append(n_classes)

        self.best_epoch = -1                    # Best Epoch
        self.loss_df = pd.DataFrame()           # Dataframe of Loss per Epoch

        self.model = TorchConvModel(*self.units_per_layer)

        self.optimizer = self.opt_name(self.model.parameters(), **self.opt_params)
        self.loss_fun = torch.nn.CrossEntropyLoss()

    def take_step(self, X, y):
        """compute predictions, loss, gradients, take one step"""
        self.optimizer.zero_grad()
        pred_tensor = self.model.forward(X)#.reshape(len(y))
        loss_tensor = self.loss_fun(pred_tensor, y.long())
        loss_tensor.backward()
        self.optimizer.step()

    def fit(self, X, y):
        """Gradient descent learning of weights"""
        units_per_layer = [ncol]
        for L in range(self.hidden_layers):
            units_per_layer.append(100)
        units_per_layer.append(n_classes)

        feature_tensor = X.reshape(len(X),1,16,16)

        ds = CSV( feature_tensor, y )
        dl = torch.utils.data.DataLoader( ds, batch_size = self.batch_size,
                                          shuffle = True )
        loss_df_list = []
        best_loss_val = 10000

        for epoch in range(self.max_epochs):
```

```
        for batch_features, batch_labels in dl:
            self.take_step(batch_features, batch_labels)
            pred = self.model(batch_features)
            loss_value = self.loss_fun(pred, batch_labels.long())

            if( loss_value < best_loss_val ):
                self.best_epoch = epoch
                best_loss_val = loss_value

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

    self.loss_df = pd.concat( loss_df_list )

  def predict(self, X):
    """Return numpy vector of predictions"""
    pred_vec = []
    for row in self.model(torch.from_numpy(X.reshape(len(X),1,16,16))):
        best_label = -1
        highest_prob = -1000
        itera = 0
        for iter in row.long():
            if(iter.item() > highest_prob):
                highest_prob = iter.item()
                best_label = itera
            itera += 1
        pred_vec.append(best_label)

    return pred_vec

class TorchConvLearnerCV:
  def __init__(self, max_epochs, batch_size, step_size, units_per_layer,
**kwargs):
    self.subtrain_learner = ConvolutionalMLP( max_epochs=max_epochs,
                                              batch_size=batch_size,
                                              step_size=step_size,
                                              units_per_layer=units_per_layer )

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

    self.batch_size = batch_size
```

```
      self.step_size = step_size
      self.units_per_layer = units_per_layer

      self.plotting_df = pd.DataFrame()

   def fit(self, X, y):
      """cross-validation for selecting the best number of epochs"""
      fold_vec = np.random.randint(low=0, high=5, 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()}

      self.subtrain_learner.validation_data = set_features["validation"]
      self.subtrain_learner.fit( set_features["subtrain"],
set_labels["subtrain"], "subtrain" )
      self.plotting_df = pd.concat([self.plotting_df,
self.subtrain_learner.loss_df])

      best_epochs = self.subtrain_learner.best_epoch

      self.train_learner = ConvolutionalMLP( max_epochs=best_epochs,
                                            batch_size=self.batch_size,
                                            step_size=self.step_size,
                                            units_per_layer=self.units_per_layer )
      self.train_learner.fit( set_features["validation"],
set_labels["validation"], "validation" )
      self.plotting_df = pd.concat([self.plotting_df,
self.train_learner.loss_df])

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

class RegularizedMLP:
    def __init__(self, **kwargs):
        kwargs.setdefault("max_epochs", MAX_EPOCHS_VAR)
```

```python
        kwargs.setdefault("batch_size", BATCH_SIZE_VAR)
        kwargs.setdefault("step_size", 0.001)
        kwargs.setdefault("units_per_layer", ( ncol, 1000, 100, n_classes ) )

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

        self.estimator = estimator()

    def fit(self, X, y):
        self.estimator.fit(X, y)

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

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,
                                 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 = self.estimator.loss_df
                for param_name, param_value in parameter_entry.items():
                    printing_df[param_name] = str(param_value)
                printing_df['set'] = 'subtrain'
                printing_df['subfold'] = foldnum
                self.plotting_df = pd.concat([self.plotting_df, printing_df])

                self.estimator.fit(**set_data_dict["validation"])

                printing_df = self.estimator.loss_df
                for param_name, param_value in parameter_entry.items():
                    printing_df[param_name] = str(param_value)
                printing_df['set'] = 'validation'
                printing_df['subfold'] = foldnum
                self.plotting_df = pd.concat([self.plotting_df, printing_df])

                # 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)

        # 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)
```

```python
        # 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 OptimizerMLP:
   def __init__(self, **kwargs):
      """Store hyper-parameters, TorchModel instance, loss, etc."""
      kwargs.setdefault("max_epochs", 2)
      kwargs.setdefault("batch_size", BATCH_SIZE_VAR)
      kwargs.setdefault("step_size", 0.01)
      kwargs.setdefault("units_per_layer", ( ncol, 95, 10, 128, 2 ) )
      kwargs.setdefault("hidden_layers", 3)
      kwargs.setdefault("opt_name", torch.optim.SGD)
      kwargs.setdefault("opt_params", {'lr':0.1})

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

      units_per_layer = [ncol]
      for L in range(self.hidden_layers):
          units_per_layer.append(100)
      units_per_layer.append(n_classes)

      self.best_epoch = -1                    # Best Epoch
      self.loss_df = pd.DataFrame()           # Dataframe of Loss per Epoch

      self.model = TorchModel(*self.units_per_layer)

      self.optimizer = self.opt_name(self.model.parameters(), **self.opt_params)
      self.loss_fun = torch.nn.CrossEntropyLoss()

   def take_step(self, X, y):
      """compute predictions, loss, gradients, take one step"""
      self.optimizer.zero_grad()
      pred_tensor = self.model.forward(X)#.reshape(len(y))
      loss_tensor = self.loss_fun(pred_tensor, y.long())
      loss_tensor.backward()
      self.optimizer.step()

   def fit(self, X, y):
      """Gradient descent learning of weights"""
```

```
    units_per_layer = [ncol]
    for L in range(self.hidden_layers):
        units_per_layer.append(100)
    units_per_layer.append(n_classes)

    ds = CSV( X, y )
    dl = torch.utils.data.DataLoader( ds, batch_size = self.batch_size,
                                      shuffle = True )
    loss_df_list = []
    best_loss_val = 10000

    for epoch in range(self.max_epochs):
       for batch_features, batch_labels in dl:
          self.take_step(batch_features, batch_labels)
          pred = self.model(batch_features)
          loss_value = self.loss_fun(pred, batch_labels.long())

          if( loss_value < best_loss_val ):
              self.best_epoch = epoch
              best_loss_val = loss_value

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

    self.loss_df = pd.concat( loss_df_list )

def predict(self, X):
    """Return numpy vector of predictions"""
    pred_vec = []
    for row in self.model(torch.from_numpy(X)):
        best_label = -1
        highest_prob = -1000
        itera = 0
        for iter in row.long():
            if(iter.item() > highest_prob):
                highest_prob = iter.item()
                best_label = itera
            itera += 1
        pred_vec.append(best_label)

    return pred_vec
```

```
class TorchLearnerCV:
   def __init__(self, max_epochs, batch_size, step_size, units_per_layer,
**kwargs):
      self.subtrain_learner = OptimizerMLP( max_epochs=max_epochs,
                                            batch_size=batch_size,
                                            step_size=step_size,
                                            units_per_layer=units_per_layer )

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

      self.batch_size = batch_size
      self.step_size = step_size
      self.units_per_layer = units_per_layer

      self.plotting_df = pd.DataFrame()

   def fit(self, X, y):
      """cross-validation for selecting the best number of epochs"""
      fold_vec = np.random.randint(low=0, high=5, 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()}

      self.subtrain_learner.validation_data = set_features["validation"]
      self.subtrain_learner.fit( set_features["subtrain"],
set_labels["subtrain"], "subtrain" )
      self.plotting_df = pd.concat([self.plotting_df,
self.subtrain_learner.loss_df])

      best_epochs = self.subtrain_learner.best_epoch

      self.train_learner = OptimizerMLP( max_epochs=best_epochs,
                                         batch_size=self.batch_size,
                                         step_size=self.step_size,
```

```
                                                    units_per_layer=self.units_per_layer )
      self.train_learner.fit( set_features["validation"],
set_labels["validation"], "validation" )
      self.plotting_df = pd.concat([self.plotting_df,
self.train_learner.loss_df])

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

# <-- END INITIALIZATION -->

# <-- BEGIN FUNCTIONS -->
# FUNCTION: MAIN
#   Description  : Main driver for Assignment Ten
#   Inputs       : None
#   Outputs      : PlotNine graphs, printed and saved to directory
#   Dependencies : build_image_df_from_dataframe
def main():
    # Display the title
    print("\nCS 499: Homework 13 Program Start")
    print("================================\n")

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

    # Download data files
    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
    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])
    zip_df[0] = zip_df[0].astype(int)

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

    zip_ones = zip_df[zip_df[0].isin([0, 1])]
    zip_sevens = zip_df[zip_df[0].isin([0, 7])]

    zip_one_features = zip_ones.iloc[:,:-1].to_numpy()
    zip_one_labels = zip_ones[0].to_numpy()
```

```
zip_seven_features = zip_sevens.iloc[:,:-1].to_numpy()
zip_seven_labels = zip_sevens[0].to_numpy()


zip_seven_labels = np.where(zip_sevens[0] == 7, 1, 0)

print(zip_seven_labels)


# Create data dictionary
data_dict = {
    'zip_seven_zero' : [zip_seven_features, zip_seven_labels],
    'zip_one_zero' : [zip_one_features, zip_one_labels]
}

final_df_list = []
final_deep_print_list = []

final_deep_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)

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

        # 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]
            }

        # Finalizing variables for CV construction
```

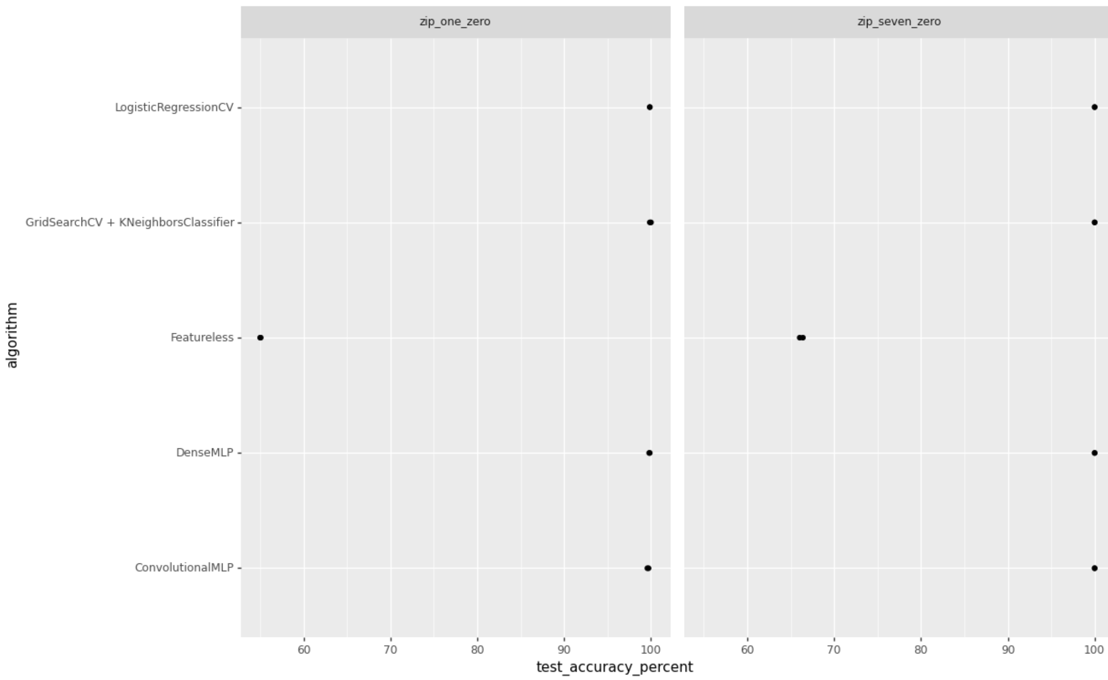
```python
param_dicts = [{'n_neighbors':[x]} for x in range(1, 21)]
global n_classes
n_classes = len( np.unique( set_data_dict['test']['y'] ) )
UNITS_PER_VAR = ( 256, 10, 10, 128, 1 )

param_grid = []

for momentum in 0.1, 0.5, 0.9:
    for lr in 0.1, 0.01, 0.001:
        param_grid.append({
            "opt_name":torch.optim.SGD,
            "opt_params":{"momentum":momentum, "lr":lr}
        })
for beta1 in 0.85, 0.9, 0.95:
    for beta2 in 0.99, 0.999, 0.9999:
        param_grid.append({
            "opt_name":torch.optim.Adam,
            "opt_params":{"betas":(beta1, beta2)}
        })

clf = GridSearchCV(KNeighborsClassifier(), param_dicts)
linear_model = sklearn.linear_model.LogisticRegressionCV(cv=CV_VAL)
DeepConvTorchCV = MyCV( max_epochs = MAX_EPOCHS_VAR,
                    batch_size = BATCH_SIZE_VAR,
                    step_size = STEP_SIZE_VAR,
                    units_per_layer = UNITS_PER_VAR,
                    estimator = ConvolutionalMLP,
                    param_grid = param_grid,
                    num_folds = CV_VAL )

print(get_n_params(DeepConvTorchCV.estimator.model))

DeepTorchCV = MyCV( max_epochs = MAX_EPOCHS_VAR,
                    batch_size = BATCH_SIZE_VAR,
                    step_size = STEP_SIZE_VAR,
                    units_per_layer = UNITS_PER_VAR,
                    estimator = OptimizerMLP,
                    param_grid = param_grid,
                    num_folds = CV_VAL )

print(get_n_params(DeepTorchCV.estimator.model))

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

```python
        DeepConvTorchCV.fit(**set_data_dict["train"])
        DeepTorchCV.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 = DeepConvTorchCV.plotting_df
        buffer_df['fold'] = foldnum
        buffer_df['data_set'] = data_set
        final_deep_print_list.append(buffer_df)

        # Get results
        pred_dict = {
            "GridSearchCV + KNeighborsClassifier": \
                clf.predict(set_data_dict["test"]["X"]),
            "LogisticRegressionCV": \
                linear_model.predict(set_data_dict["test"]["X"]),
            "ConvolutionalMLP": \
                DeepConvTorchCV.predict(set_data_dict["test"]["X"]),
            "DenseMLP": \
                DeepTorchCV.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)

# 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}))

    print(plot)

    # Epoch vector for plotting
    """epoch_vec = np.arange(MAX_EPOCHS_VAR)
    epoch_vec = np.tile(epoch_vec, 1)
    epoch_vec = epoch_vec.flatten()"""

    final_deep_df['opt_and_params'] = final_deep_df['opt_name'] +
final_deep_df['opt_params']
    final_deep_df = final_deep_df.groupby(['set', 'data_set', 'epoch',
'opt_and_params'], as_index=False).mean()
    #final_deep_df['epochs'] = epoch_vec
    print(final_deep_df)
    deepplot = (p9.ggplot(final_deep_df,
                      p9.aes(x='epoch',
                             y='loss',
                             color='set'))
                 + p9.facet_grid('opt_and_params ~ data_set', scales='free',
shrink=True)
                 + p9.geom_line()
                 + p9.theme(subplots_adjust={'left': 0.2},
                            strip_text_y = p9.element_text(angle = 0,ha =
'left'),
                            figure_size=(25,25))
                 + p9.ggtitle("Loss vs. Epochs"))

    print(plot)
    deepplot.save("DeepTorch Loss Graph.png")

    print("\nCS 499: Homework 13 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
```

```python
#       - 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")

# FUNCTION : DOWNLOAD_DATA_FILE
#   Description: Downloads file from source, if not already downloaded
#   Inputs:
#       - file      : Name of file to download
#   Outputs: None
def get_n_params(module):
    return sum(
        [math.prod(list(p.shape)) for p in module.parameters()])

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

Program Output:




```
  test_accuracy_percent        data_set  fold_id                            algorithm
0            100.000000  zip_seven_zero        0  GridSearchCV + KNeighborsClassifier
0            100.000000  zip_seven_zero        0                 LogisticRegressionCV
0            100.000000  zip_seven_zero        0                      ConvolutionalMLP
0            100.000000  zip_seven_zero        0                             DenseMLP
0             66.410912  zip_seven_zero        0                          Featureless
0            100.000000  zip_seven_zero        1  GridSearchCV + KNeighborsClassifier
0            100.000000  zip_seven_zero        1                 LogisticRegressionCV
0            100.000000  zip_seven_zero        1                      ConvolutionalMLP
0            100.000000  zip_seven_zero        1                             DenseMLP
0             66.040956  zip_seven_zero        1                          Featureless
0             99.858257    zip_one_zero        0  GridSearchCV + KNeighborsClassifier
0             99.858257    zip_one_zero        0                 LogisticRegressionCV
0             99.716513    zip_one_zero        0                      ConvolutionalMLP
0             99.858257    zip_one_zero        0                             DenseMLP
0             55.067328    zip_one_zero        0                          Featureless
0            100.000000    zip_one_zero        1  GridSearchCV + KNeighborsClassifier
0             99.858257    zip_one_zero        1                 LogisticRegressionCV
0             99.574770    zip_one_zero        1                      ConvolutionalMLP
0             99.787385    zip_one_zero        1                             DenseMLP
0             54.996456    zip_one_zero        1                          Featureless
```

Question Answers / Commentary:

For this assignment I implemented 2 convolutional layers into a new Torch model. Alongside this convolutional network, I also implemented a deep fully connected network with roughly the same number of layers (27,394 vs. 27,041). Overall, it appears that there was extremely high accuracy with little variation between the two models. I did implement a Max Pooling layer, but with a functional kernel size of 1 it has no effect.