Assignment Ten

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
# <-- 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"
```

```python
ziptest_file_path = os.path.join(download_directory, ziptest_file)

# CONSTANT VARIABLES
spam_label_col = 57
zip_empty_col = 257

MAX_EPOCHS_VAR = 500
BATCH_SIZE_VAR = 256
STEP_SIZE_VAR = 0.001
CV_VAL = 5
N_FOLDS = 3

# 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 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 TorchLearner:
   def __init__(self, max_epochs, batch_size, step_size, units_per_layer):
      """Store hyper-parameters, TorchModel instance, loss, etc."""
```

```
        self.max_epochs = max_epochs            # Max Epochs
        self.best_epoch = -1                    # Best Epoch
        self.batch_size = batch_size            # Batch Size
        self.step_size = step_size              # Step Size
        self.units_per_layer = units_per_layer  # Units Per Layer (tuple)
        self.loss_df = pd.DataFrame()           # Dataframe of Loss per Epoch

        self.model = TorchModel(*units_per_layer)

        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=0.1)
        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, set_name):
        """Gradient descent learning of weights"""
        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 )
```

```python
  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
        iter = 0
        while(iter < 10):
            if(row[iter].item() > highest_prob):
                highest_prob = row[iter].item()
                best_label = iter
            iter += 1
        pred_vec.append(best_label)

    return pred_vec

class TorchLearnerCV:
  def __init__(self, max_epochs, batch_size, step_size, units_per_layer):
    self.subtrain_learner = TorchLearner( max_epochs, batch_size,
                                          step_size, units_per_layer )

    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()}
```

```python
      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 = TorchLearner( best_epochs, self.batch_size,
                                          self.step_size, 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 10 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)
```

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

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

    ds = torchvision.datasets.MNIST(
        root="~/teaching/cs499-599-fall-2022/data",
        download=True,
        transform=torchvision.transforms.ToTensor(),
        train=False)
    dl = torch.utils.data.DataLoader(ds, batch_size=len(ds), shuffle=False)
    for mnist_features, mnist_labels in dl:
        pass
    mnist_features = mnist_features.flatten(start_dim=1).numpy()
    mnist_labels = mnist_labels.numpy()

    # Create data dictionary
    data_dict = {
        'mnist' : [mnist_features, mnist_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)

        # 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))

            # Creating dictionary with input and outputs
```

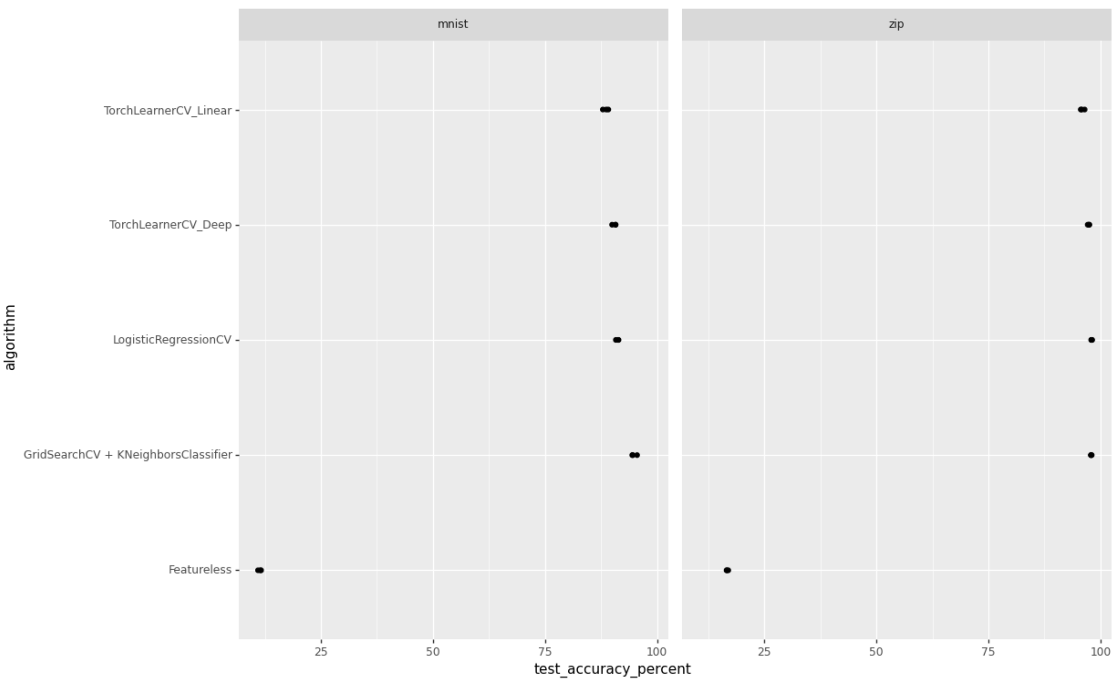
```
set_data_dict = {}
for set_name, index_vec in index_dict.items():
    set_data_dict[set_name] = {
        "X":input_data[index_vec],
        "y":output_array[index_vec]
    }

# Finalizing variables for CV construction
param_dicts = [{'n_neighbors':[x]} for x in range(1, 21)]
n_classes = len( np.unique( set_data_dict['test']['y'] ) )
UNITS_PER_VAR = ( ncol, 1000, 100, n_classes )

if( current_set == 'zip' ):
    STEP_SIZE_VAR = 0.001
if( current_set == 'mnist' ):
    STEP_SIZE_VAR = 0.00001

clf = GridSearchCV(KNeighborsClassifier(), param_dicts)
linear_model = sklearn.linear_model.LogisticRegressionCV(cv=CV_VAL)
DeepTorchCV = TorchLearnerCV( MAX_EPOCHS_VAR, BATCH_SIZE_VAR,
                              STEP_SIZE_VAR, UNITS_PER_VAR )
UNITS_PER_VAR = ( ncol, n_classes )
LinearTorchCV = TorchLearnerCV( MAX_EPOCHS_VAR, BATCH_SIZE_VAR,
                                STEP_SIZE_VAR, UNITS_PER_VAR )

# Train the models with given data
clf.fit(**set_data_dict["train"])
linear_model.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 = DeepTorchCV.plotting_df
buffer_df['subfold'] = foldnum
buffer_df['set'] = data_set
final_deep_print_list.append(buffer_df)

buffer_df = LinearTorchCV.plotting_df
buffer_df['subfold'] = foldnum
buffer_df['set'] = data_set
final_linear_print_list.append(buffer_df)

# Get results
```

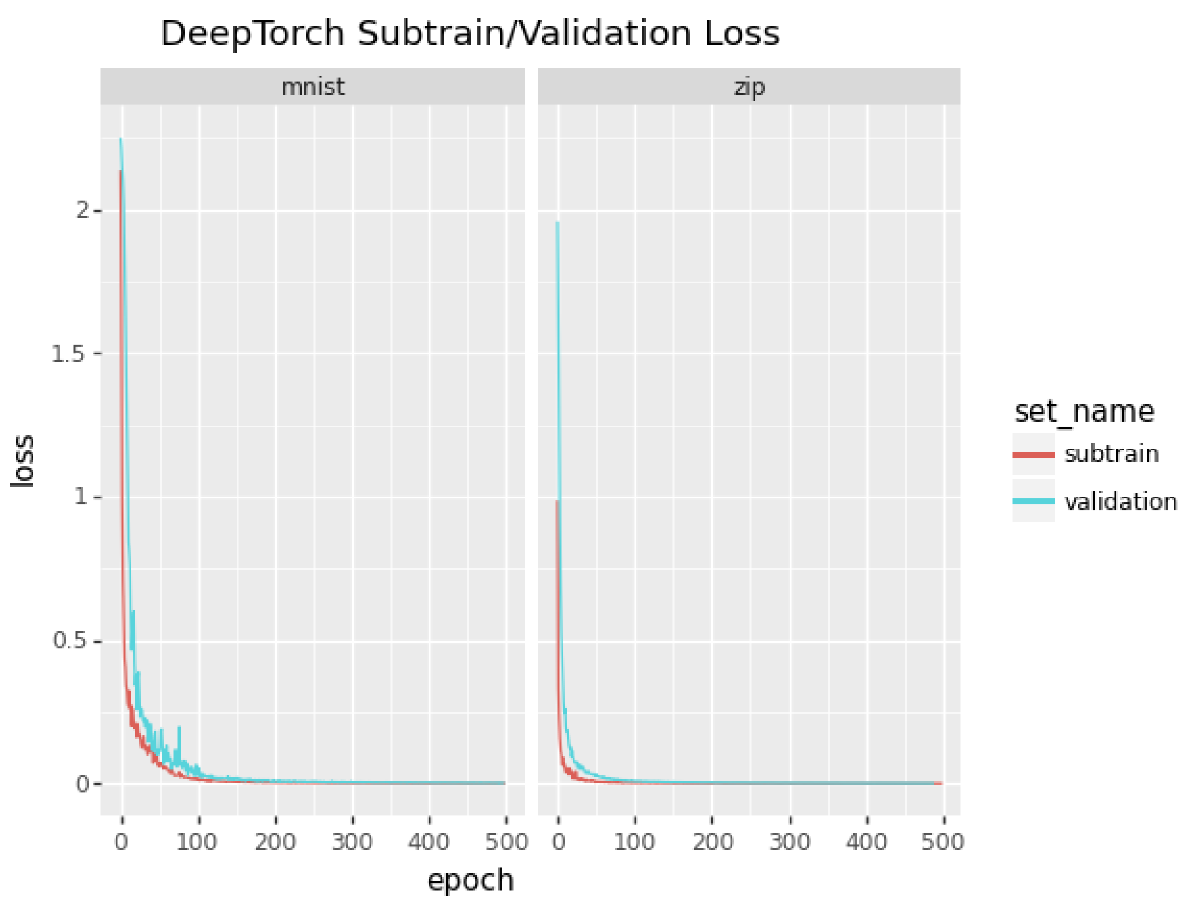
```
        pred_dict = {
            "GridSearchCV + KNeighborsClassifier": \
                clf.predict(set_data_dict["test"]["X"]),
            "LogisticRegressionCV": \
                linear_model.predict(set_data_dict["test"]["X"]),
            "TorchLearnerCV_Deep": \
                DeepTorchCV.predict(set_data_dict["test"]["X"]),
            "TorchLearnerCV_Linear": \
                LinearTorchCV.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 vector for plotting
"""epoch_vec = np.arange(MAX_EPOCHS_VAR)
epoch_vec = np.tile(epoch_vec, 1)
```

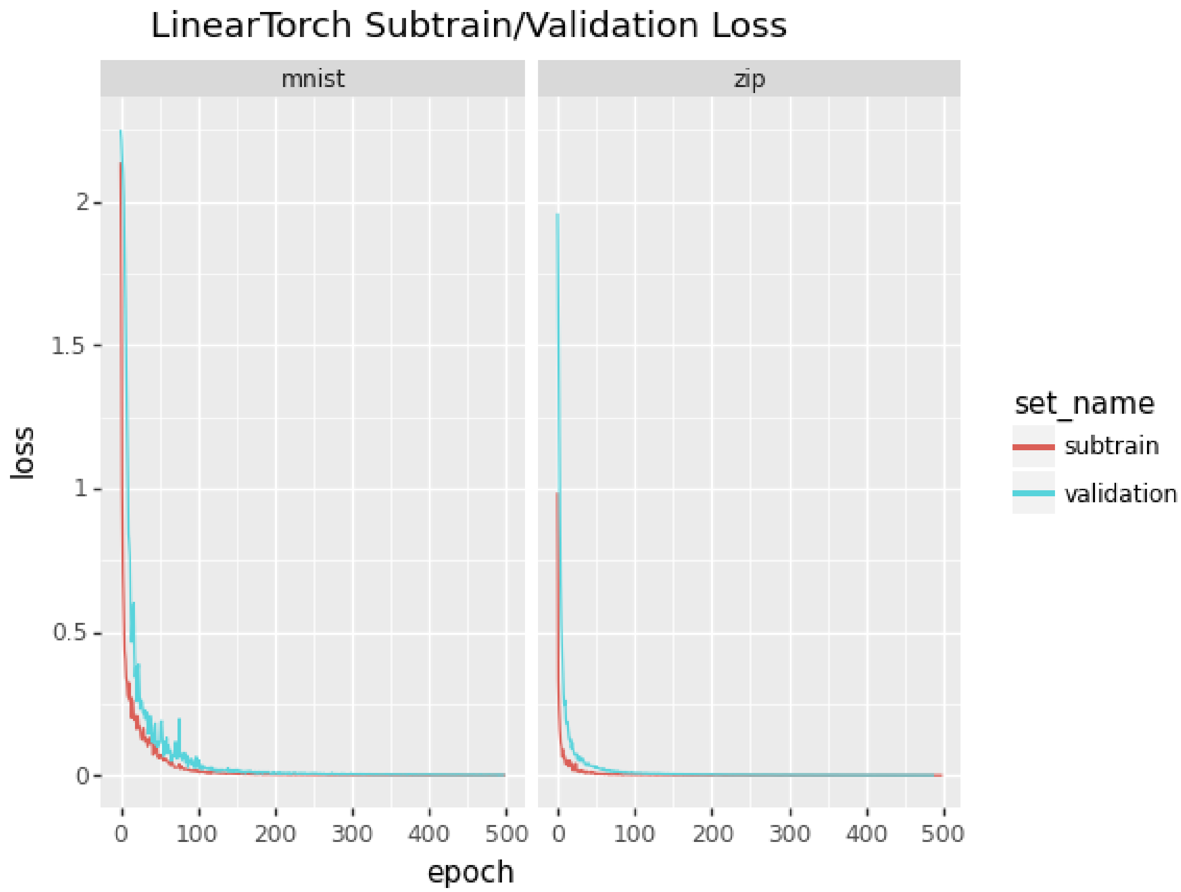
```python
    epoch_vec = epoch_vec.flatten()"""

    final_deep_df = final_deep_df.groupby(['set', 'epoch', 'set_name'],
as_index=False).mean()
    #final_deep_df['epochs'] = epoch_vec

    deepplot = (p9.ggplot(final_deep_df,
                        p9.aes(x='epoch',
                               y='loss',
                               color='set_name'))
                  + p9.facet_grid('. ~ set', scales='free')
                  + p9.geom_line()
                  + p9.theme(subplots_adjust={'left': 0.2})
                  + p9.ggtitle("DeepTorch Subtrain/Validation Loss"))

    final_linear_df = final_linear_df.groupby(['set', 'epoch', 'set_name'],
as_index=False).mean()
    #final_linear_df['epochs'] = epoch_vec

    linearplot = (p9.ggplot(final_linear_df,
                        p9.aes(x='epoch',
                               y='loss',
                               color='set_name'))
                  + p9.facet_grid('. ~ set', scales='free')
                  + p9.geom_line()
                  + p9.theme(subplots_adjust={'left': 0.2})
                  + p9.ggtitle("LinearTorch Subtrain/Validation Loss"))
    print(plot)
    deepplot.save("DeepTorch Loss Graph.png")
    linearplot.save("LinearTorch Loss Graph.png")

    print("\nCS 499: Homework 10 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):
```

```python
        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:


mnist
zip
TorchLearnerCV_Linear
TorchLearnerCV_Deep
LogisticRegressionCV
GridSearchCV + KNeighborsClassifier
Featureless
algorithm
25
50
75
100
test_accuracy_percent


DeepTorch Subtrain/Validation Loss
mnist
zip
loss
2
1.5
1
0.5
0
0
100
200
300
400
500
epoch
set_name
subtrain
validation


LinearTorch Subtrain/Validation Loss
mnist
zip
loss
epoch
set_name
subtrain
validation
0
0.5
1
1.5
2
0
100
200
300
400
500

	test_accuracy_percent	data_set	fold_id	algorithm
0	95.560888	mnist	0	GridSearchCV + KNeighborsClassifier
0	91.301740	mnist	0	LogisticRegressionCV
0	90.761848	mnist	0	TorchLearnerCV_Deep
0	88.662268	mnist	0	TorchLearnerCV_Linear
0	11.637672	mnist	0	Featureless
0	94.599460	mnist	1	GridSearchCV + KNeighborsClassifier
0	91.419142	mnist	1	LogisticRegressionCV
0	90.759076	mnist	1	TorchLearnerCV_Deep
0	89.138914	mnist	1	TorchLearnerCV_Linear
0	10.951095	mnist	1	Featureless
0	94.479448	mnist	2	GridSearchCV + KNeighborsClassifier
0	90.819082	mnist	2	LogisticRegressionCV
0	89.978998	mnist	2	TorchLearnerCV_Deep
0	87.908791	mnist	2	TorchLearnerCV_Linear
0	11.461146	mnist	2	Featureless
0	98.064516	zip	0	GridSearchCV + KNeighborsClassifier
0	98.032258	zip	0	LogisticRegressionCV
0	97.419355	zip	0	TorchLearnerCV_Deep
0	95.838710	zip	0	TorchLearnerCV_Linear
0	16.935484	zip	0	Featureless
0	97.902549	zip	1	GridSearchCV + KNeighborsClassifier
0	97.934818	zip	1	LogisticRegressionCV
0	97.547596	zip	1	TorchLearnerCV_Deep
0	95.611488	zip	1	TorchLearnerCV_Linear
0	16.618264	zip	1	Featureless
0	97.870281	zip	2	GridSearchCV + KNeighborsClassifier
0	98.160697	zip	2	LogisticRegressionCV
0	97.160374	zip	2	TorchLearnerCV_Deep
0	96.482736	zip	2	TorchLearnerCV_Linear
0	16.553727	zip	2	Featureless

Question Answers / Commentary:

For this assignment, I was able to implement a multi-class classification solution which can make predictions at relatively high accuracies. While not quite as good as the SciKit tools, my solution is still relatively on par, being only a few percent lower in accuracy. Overall, I believe that my solution shows that it can accurately make predictions on multi-class data.

For my final product, I used a maximum epoch of 500, with varying step sizes for each data set. The step sizes were manually selected for each data set based on experimentation.

There was relatively little difference in accuracy between the two data sets. While the MNIST data had lower accuracy overall, my implementation of the Torch Learner CV still performed on-par with the SciKit tools. There is a small difference in accuracy between my solution and the best of the SciKit tools of around 3% for both data sets.