Assignment 4: Linear Regression, Fully Connected Neural Networks, and Stacked Models

Part 1: With scikit-learn, and sing your choice of hyperparameter tuning method, run a simple ordinary least squares (OLS) linear regression model on the Housing Prices Dataset. Then, train a lasso regression model and a ridge regression model on the dataset as well. Tune the hyper-parameters on the validation set and report the mean absolute error on the test set for these three methods. In addition, provide plots of the MAE vs. regularization constant for the lasso regression model and the ridge regression model (two plots total) as you vary the regularization constant using six or more reasonable values.

Python file name on github: linear_regression.py

```
import pandas as pd
import numpy as np

from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_absolute_error

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

import matplotlib.pyplot as plt

def load_data(dataset):
    """
    Load a pair of data X,y

    Params
    ------
    dataset:    train/valid/test

    Return
    ------
    X:          shape (N, 240)
    y:          shape (N, 1)
    """
    X = pd.read_csv(f"../../Data/housing_data/{dataset}_x.csv", header=None).to_numpy()
    y = pd.read_csv(f"../../Data/housing_data/{dataset}_y.csv", header=None).to_numpy()

    return X,y

def score(model, X, y):
    """
    Score the model with X, y

    Params
    ------
    model:  the model to predict with
    X:      the data to score on
    y:      the true value y

    Return
    ------
    mae:    the mean absolute error
    """
    #generate predictions on X
    y_pred = model.predict(X)
    #calculate the mean_absolute_error
    mae = mean_absolute_error(y, y_pred)
    #return the mean absolute error
    return mae

#function that will run the ordinary least square linear regression model
def ols_lin_reg(train, valid, test):
    """
    Fit OLS model and calculate mean absolute error

    Params
    ------
    train:    training dataset
```

```
    valid:    validation dataset
    test:     test data set

    Return
    ------
    mae:    the mean absolute error of OLS model
    """
    #concatenate training and validation data. There are no hyperparameters to tune
    #with ordinary least squares because there is no regularization term
    train_X = np.concatenate([train[0], valid[0]], axis=0)
    train_y = np.concatenate([train[1], valid[1]], axis=0)

    #generate matrices for test data
    test_X, test_y = test

    #fit the model with training and test set data
    model = LinearRegression().fit(train_X, train_y)

    #get mean absolute error on test dataset
    mae_ols = score(model, test_X, test_y)

    return mae_ols

def hyper_parameter_tuning(model_class, param_grid, train, valid):
    """
    Tune the hyper-parameter using training and validation data

    Params
    ------
    model_class:    the model class
    param_grid:     the hyper-parameter grid, dict
    train:          the training data (train_X, train_y)
    valid:          the validatation data (valid_X, valid_y)

    Return
    ------
    model:          model fit with best params
    best_param:     the best params
    """
    train_X, train_y = train
    valid_X, valid_y = valid

    #initialize an arbitrarily large value of mean absolute error
    best_mae = 1000000

    # Set up the parameter grid
    param_grid = list(ParameterGrid(param_grid))

    # train the model with each parameter setting in the grid
    #loop over each combination of the parameters, alpha and max_iter
    for i in param_grid:
        #fit the model with the parameters and training data
        model = model_class(alpha=i.get('alpha'), max_iter=i.get('max_iter'), tol=i.get('tol')).fit(train_X,
train_y)
        #calculate the mean absolute error on the validation set
        mae = score(model, valid_X, valid_y)
        # choose the model with lowest MAE on validation set
        #if the current mae is lower then the best_mae, overwrite the best_mae
        if mae < best_mae:
            #overwrite the best_mae score
            best_mae = mae
            #overwrite the best_params
            best_params = i

    # then fit the model with the training and validation set (refit)
    #concatenate training data and labels into new training set
    new_train_X = np.concatenate((train_X, valid_X))
    new_train_y = np.concatenate((train_y,valid_y))
    #refit the model with the best parameters
    final_model = model_class(alpha=best_params.get('alpha'),
max_iter=best_params.get('max_iter')).fit(new_train_X, new_train_y)

    # return the fitted model, the best parameter setting, and best mae
    return final_model, best_params, best_mae

def plot_mae_alpha(model_class, params, train, valid, test, title="Model"):
    """
    Plot the model MAE vs Alpha (regularization constant)

    Params
    ------
```

```
    model_class:    The model class to fit and plot
    params:         The best params found
    train:          The training dataset
    valid:          The validation dataest
    test:           The testing dataset
    title:          The plot title

    Return
    ------
    None
    """
    train_X = np.concatenate([train[0], valid[0]], axis=0)
    train_y = np.concatenate([train[1], valid[1]], axis=0)

    #generate matrices for test data
    test_X, test_y = test

    # set up the list of alphas to train on
    alphas = [0.001, 0.01, 0.1, 0.2, 0.5, 1, 10, 100]

    #initialize an empty list for storing MAE values
    mae_vals = []

    # train the model with each alpha, log MAE
    for n in range(len(alphas)):
        #fit the model with each alpha value, and the best max_iter from hyper_parameter_tuning
        model = model_class(alpha=alphas[n], max_iter=params.get('max_iter')).fit(train_X, train_y)
        #calculate the mean absolute error
        mae = score(model, test_X, test_y)
        #append the list of mean absolute errors
        mae_vals.append(mae)

    # plot the MAE - Alpha
    plt.plot(alphas, mae_vals)
    #plot title
    plt.title('MAE for Various Values of the Regularization\nConstant, Alpha, for the '+title+' Model')
    #label x and y axes
    plt.xlabel('Regularization Constant (Alpha)')
    plt.ylabel('Mean Absolute Error')
    #save to Submission/Figures folder with appropriate name
    plt.savefig(('../Figures/'+title+'.png'))
    #clear figure
    plt.clf()


def main():
    """
    Load in data
    """
    train = load_data('train')
    valid = load_data('valid')
    test = load_data('test')

    """
    Define the parameter grid for each classifier
    e.g. lasso_grid = dict(alpha=[0.1, 0.2, 0.4],
                           max_iter=[1000, 2000, 5000])
    """

    #create a dictioanry to store MAE values
    models = ['OLS', 'Lasso', 'Ridge']
    models_best_mae = dict([(key, []) for key in models])

    #calculate MAE of ordinary least squares model
    ols_mae = ols_lin_reg(train, valid, test)
    models_best_mae['OLS'].append(ols_mae)

    # Tune the hyper-paramter by calling the hyper-parameter tuning function
    # e.g. lasso_model, lasso_param = hyper_parameter_tuning(Lasso, lasso_grid, train, valid)
    lasso_grid = dict(alpha=[0.1, 0.2, 0.4, 0.8, 1.0, 10], max_iter=[1000, 2000, 5000], tol=[0.001, 0.01, 0.1,
1, 10, 100])
    lasso_model, lasso_params, best_lasso_mae = hyper_parameter_tuning(Lasso, lasso_grid, train, valid)
    print('\nBest Lasso Params:')
    print(lasso_params)
    models_best_mae['Lasso'].append(best_lasso_mae)

    ridge_grid = dict(alpha=[0.1, 0.2, 0.4, 0.8, 1.0, 10], max_iter=[1000, 2000, 5000], tol=[0.001, 0.01, 0.1,
1, 10, 100])
    ridge_model, ridge_params, best_ridge_mae = hyper_parameter_tuning(Ridge, ridge_grid, train, valid)
    print('\nBest Ridge Params:')
    print(ridge_params)
```

```
    models_best_mae['Ridge'].append(best_ridge_mae)

    # Plot the MAE - Alpha plot by calling the plot_mae_alpha function
    # e.g. plot_mae_alpha(Lasso, lasso_param, train, valid, test, "Lasso")
    plot_mae_alpha(Lasso, lasso_params, train, valid, test, "Lasso")
    plot_mae_alpha(Ridge, ridge_params, train, valid, test, "Ridge")

    print('\nBest MAEs for each model')
    print(models_best_mae)

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

Part 2: Implement the neural network with the architecture outlined above using PyTorch library for classifying MNIST digits, containing 10 output labels. In particular, you need to define your layers in the init () function, and define the connectivity between the layers in the forward() function of class NNet(nn.Module). The hidden layers use ReLU activation and class probability output layer uses softmax activation. For training, use batch size of 32, and Adam optimizer (torch.optim.Adam) with learning rate of 0.002 for 15 epochs. Plot the training and test accuracy with respect to number of epochs of training.

Python file name on github: neural_network.py

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

import numpy as np
import torch.utils.data as utils

import matplotlib.pyplot as plt

# The parts that you should complete are designated as TODO
class NNet(nn.Module):
    def __init__(self):
        super(NNet, self).__init__()
        # TODO: define the layers of the network
        self.hidden_1 = nn.Linear(784, 64)
        self.hidden_2 = nn.Linear(64, 32)

    def forward(self, x):
        # TODO: define the forward pass of the network using the layers you defined in constructor
        h1 = F.relu(self.hidden_1(x))
        h2 = F.relu(self.hidden_2(h1))
        output = F.softmax(h2)

        return output

def train(model, device, train_loader, optimizer, epoch):
    model.train()
    correct = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0: #Print loss every 100 batch
            print('Train Epoch: {}\tLoss: {:.6f}'.format(
                epoch, loss.item()))
    accuracy = test(model, device, train_loader)
    return accuracy

def test(model, device, test_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
```

```
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    accuracy = 100. * correct / len(test_loader.dataset)

    return accuracy


def main():
    torch.manual_seed(1)
    np.random.seed(1)
    # Training settings
    use_cuda = False # Switch to False if you only want to use your CPU
    learning_rate = 0.002
    NumEpochs = 15
    batch_size = 32

    device = torch.device("cuda" if use_cuda else "cpu")

    train_X = np.load('../../Data/mnist/X_train.npy')
    train_Y = np.load('../../Data/mnist/y_train.npy')

    test_X = np.load('../../Data/mnist/X_test.npy')
    test_Y = np.load('../../Data/mnist/y_test.npy')

#   train_X = train_X.reshape([-1,1,28,28]) # the data is flatten so we reshape it here to get to the original
dimensions of images
#   test_X = test_X.reshape([-1,1,28,28])

    # transform to torch tensors
    tensor_x = torch.tensor(train_X, device=device)
    tensor_y = torch.tensor(train_Y, dtype=torch.long, device=device)

    test_tensor_x = torch.tensor(test_X, device=device)
    test_tensor_y = torch.tensor(test_Y, dtype=torch.long)

    train_dataset = utils.TensorDataset(tensor_x,tensor_y) # create your datset
    train_loader = utils.DataLoader(train_dataset, batch_size=batch_size) # create your dataloader

    test_dataset = utils.TensorDataset(test_tensor_x,test_tensor_y) # create your datset
    test_loader = utils.DataLoader(test_dataset) # create your dataloader if you get a error when loading test
data you can set a batch_size here as well like train_dataloader

    model = NNet().to(device)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    #initialize a list to save training accuracy
    training_accuracy = []
    #initialize a list to save testing accuracy
    testing_accuracy = []


    for epoch in range(NumEpochs):
        train_acc = train(model, device, train_loader, optimizer, epoch)
        #append training accuracy list
        training_accuracy.append(train_acc)
        print('\nTrain set Accuracy: {:.0f}%\n'.format(train_acc))
        test_acc = test(model, device, test_loader)
        #append testing accuracy list
        testing_accuracy.append(test_acc)
        print('\nTest set Accuracy: {:.0f}%\n'.format(test_acc))

    torch.save(model.state_dict(), "mnist_nn.pt")

    #TODO: Plot train and test accuracy vs epoch
    plt.plot(list(range(1,16)), training_accuracy, label='Training Accuracy')
    plt.plot(list(range(1,16)), testing_accuracy, label='Testing Accuracy')
    #plt.xticks(np.arange(list(range(1,16))))
    plt.title('Training and Testing Accuracy for\nIncreasing Number of Epochs')
    plt.xlabel('Number of Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.savefig('../Figures/nn_accuracy.png')
    plt.clf()


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

Part 3: In this question, for the Breast Cancer Wisconsin (Diagnostic) Dataset, try stacking multiple models using sklearn.ensemble.StackingClassifier module in two levels.

Python file name on github: stacked_models.py

```
#import packages
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import StackingClassifier
from sklearn.metrics import f1_score



def main():
    #import training and testing data
    train_X = np.load('../../Data/breast_cancer_data/train_x.npy')
    train_Y = np.load('../../Data/breast_cancer_data/train_y.npy')
    test_X = np.load('../../Data/breast_cancer_data/test_x.npy')
    test_Y = np.load('../../Data/breast_cancer_data/test_y.npy')

    ###BASE ESTIMATORS###
    #create a list of the base estimator models to use with the StackingClassifier function
    #Model 1 base estimators: random forest and linear support vector machine
    #estimators = [('rf', RandomForestClassifier(n_estimators=10, random_state=42)),('svr',
make_pipeline(StandardScaler(), LinearSVC(random_state=42)))]

    #Model 2 base estimators: Random Forest and KNN
    #estimators = [('rf', RandomForestClassifier(n_estimators=10, random_state=42)),('knn',
KNeighborsClassifier(n_neighbors=5))]

    #Model 3: Random Forest and Logistic Regression with final estimator as SVM
    #estimators = [('rf', RandomForestClassifier(n_estimators=10, random_state=42)),('lr',
KNeighborsClassifier(n_neighbors=5))]

    #Model 4: Random Forest and SVM (testing different kernels), final estimator = LinearRegression
    estimators = [('rf', RandomForestClassifier(n_estimators=10, random_state=42)),('svr',
make_pipeline(StandardScaler(), SVC(kernel='rbf',random_state=42)))]



    ###STACKED CLASSIFIERS###
    #stack the base estimators with the final estimator
    #Final estimator as logistic regression
    clf = StackingClassifier(estimators=estimators, final_estimator=LogisticRegression())


    #fit the model with the training data and labels
    clf.fit(train_X, train_Y)

    #generate predictions on the test data set
    y_pred = clf.predict(test_X)

    #calculate the f1 score
    score = f1_score(test_Y, y_pred)

    #print the f1 score
    print('F1 Score:')
    print(score)


if __name__ == '__main__':
    main()


#Results from different models
#Model 1: Base: Random Forest and LinearSVC, Final: Logistic Regression
#F1 Score: 0.9589

#Model 2: Base: Random Forest and KNN, Final: Logistic Regression
#F1 Score: 0.9445
```

```
#Model 4: Base: Random Forest and linear kernel SVC, final: Logistic Regression
#F1 Score: RBF Kernel: 0.9859, Linear Kernel:0.9722, Polynomial Kernel: 0.9859
```