AdaBoost Code :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
import scipy.io as sio
import math

data = sio.loadmat('mnist.mat')


Xtrain = data['trainX'][:10000,:].astype(int)
Xtest = data['testX'].astype(int)
ytrain =data['trainY'][0,:10000].astype(int)
ytest =  data['testY'][0,:].astype(int)

idx = np.logical_or(np.equal(ytrain,4), np.equal(ytrain,9))
Xtrain = Xtrain[idx,:]
ytrain = ytrain[idx]
ytrain[np.equal(ytrain,4)] = 1
ytrain[np.equal(ytrain,9)] = -1

idx = np.logical_or(np.equal(ytest,4), np.equal(ytest,9))
Xtest = Xtest[idx,:]
ytest = ytest[idx]
ytest[np.equal(ytest,4)] = 1
ytest[np.equal(ytest,9)] = -1


sio.savemat('mnist_binary_small.mat',{'Xtrain':Xtrain,'ytrain':ytrain,'Xtest':Xte
st,'ytest':ytest})


data = sio.loadmat('mnist_binary_small.mat')
Xtrain = data['Xtrain']
Xtest = data['Xtest']
ytrain = data['ytrain'][0,:]
ytest = data['ytest'][0,:]
```

```
print(Xtrain.shape, Xtest.shape, ytrain.shape, ytest.shape)
def get_weighted_misclass(w,y,yhat):
    return (sum(w * (np.not_equal(y, yhat)).astype(int)))/sum(w)
    #finds the error rate of a weak classifier

def get_misclass(y,yhat):
    return (np.sum(np.not_equal(y,yhat)) / len(y))
    # see's how many you are matching and gets the % of whats that match

def get_exp_loss(y,yhat):
    return np.mean(np.exp(-y * yhat))
    #exponetional loss formula

aTree = tree.DecisionTreeClassifier(max_depth=1) #creating a decision tree
aTree.fit(Xtrain, ytrain) #X train is a matrix, y Train is a vector

predictionXTest = aTree.predict(Xtest)
predictionXTrain = aTree.predict(Xtrain)

test_misclass = get_misclass(ytest, predictionXTest)
train_misclass = get_misclass( ytrain, predictionXTrain)
train_expo = get_exp_loss(ytrain, predictionXTrain)
print("Train misclassification is " + str(round(train_misclass*100, 2)) + '%',
"test misclasification is " + str(round(test_misclass*100, 2)) +
'%',str(round(train_expo,2)) + " exponential loss")


trainRates, testRates, expLosses = [], [], []

for m in range(1, 101):
    clf = tree.DecisionTreeClassifier(max_depth=m)
    clf.fit(Xtrain, ytrain)
    trainRates.append(get_misclass(ytrain, clf.predict(Xtrain)))
    testRates.append(get_misclass(ytest, clf.predict(Xtest)))
    expLosses.append(get_exp_loss(ytrain, clf.predict(Xtrain)))

plt.plot(trainRates)
plt.plot(testRates)
plt.plot(expLosses)
```

```
training = ["training-misclassification","testing_misclassification",
"training_expLoss"]

plt.legend(training)
plt.show()

print("min train classification rate:", round(min(trainRates)*100, 2),"%")
print("min test classification rate:", round(min(testRates)*100,2),"%")
print("min exponential loss:", min(expLosses))
def get_weighted_misclass_For_Adaboost(w,y,yhat):
    return (sum(w * (np.not_equal(y, yhat)).astype(int)))/sum(w)
    #finds the error rate of a weak classifier

def update(error):
    return (1/2) * np.log((1 - error) / error)

def update_weights(w, update, y, yhat):
    return w * np.exp(update * (np.not_equal(y, yhat)).astype(int))

wetrain, wetest  = np.ones(len(ytrain)) / len(ytrain), np.ones(len(ytest)) /
len(ytest) #initializing
theExpLoss, trainingMisclassifications, testingMisclassifications,
alphaWeightTrains, alphaWeightTests = [], [], [], [], []

stump = tree.DecisionTreeClassifier(max_depth = 1)
stump.fit(Xtrain, ytrain, sample_weight = wetrain)
predicationXTrain = stump.predict(Xtrain)

stump.fit(Xtest, ytest, sample_weight = wetest)
predicationXTest = stump.predict(Xtest)

errorMisclassRateTrain = get_weighted_misclass_For_Adaboost(wetrain, ytrain,
predicationXTrain) #gets weighted misclassification error rate in training
errorMisclassRateTest = get_weighted_misclass_For_Adaboost(wetest, ytest,
predicationXTest) #gets weighted misclassification error rate in testing

trainingMisclassifications.append(errorMisclassRateTrain)
testingMisclassifications.append(errorMisclassRateTest)
```

```
alphaTrain = update(errorMisclassRateTrain)
alphaTest = update(errorMisclassRateTest)

alphaWeightTrains.append(alphaTrain)
alphaWeightTests.append(alphaTest)

theExpLoss.append(get_exp_loss(ytrain, predicationXTrain))

for m in range(1, 101):
    wetrain = update_weights(wetrain, alphaTrain, ytrain, predicationXTrain)
    wetest = update_weights(wetest, alphaTest, ytest, predicationXTest)

    stump = tree.DecisionTreeClassifier(max_depth = 1)
    stump.fit(Xtrain, ytrain, sample_weight = wetrain)
    predicationXTrain = stump.predict(Xtrain)

    stump.fit(Xtest, ytest, sample_weight = wetest)
    predicationXTest = stump.predict(Xtest)

    errorMisclassRateTrain = get_weighted_misclass_For_Adaboost(wetrain, ytrain,
predicationXTrain) #gets weighted misclassification error rate in training
    errorMisclassRateTest = get_weighted_misclass_For_Adaboost(wetest, ytest,
predicationXTest) #gets weighted misclassification error rate in testing

    trainingMisclassifications.append(errorMisclassRateTrain)
    testingMisclassifications.append(errorMisclassRateTest)

    alphaTrain = update(errorMisclassRateTrain)
    alphaTest = update(errorMisclassRateTest)

    alphaWeightTrains.append(alphaTrain)
    alphaWeightTests.append(alphaTest)

    theExpLoss.append(get_exp_loss(ytrain, predicationXTrain))

plt.plot(theExpLoss, label = "The Exponential Loss")
plt.plot(trainingMisclassifications, label = "The test misclassification")
plt.plot(testingMisclassifications, label = "The training misclassification")
```

```
plt.legend()

print("minimum Training misclassification", min(trainingMisclassifications))
print("minimum Testing misclassification", min(testingMisclassifications))
print("minimum Exponential Loss", min(theExpLoss))
#Question : the boosting decision tree did not perform as well as the deep trees
because
#deep creates create large trees up to a depth of 100 and the boosting we were
doing did not
#make up the difference in accuracy (Deep trees are more accurate) Exponential
loss is lower

plt.plot(trainingMisclassifications, label="training epsilon")
plt.plot(testingMisclassifications, label="testing epsilon")
plt.plot(alphaWeightTrains, label="training alpha")
plt.plot(alphaWeightTests, label="training alpha")
plt.legend()
#Question :
```

WeatherPolyFitting :

```
import numpy as np
import matplotlib.pyplot as plt
import datetime
import copy
import scipy.io as sio
import math
#This cell forms the mat file you were already given
data = sio.loadmat('weatherDewTmp.mat')
weeks_after_start = data['weeks'][0]
dewtemp = data['dew'][0]
N = len(dewtemp)
plt.plot(weeks_after_start, dewtemp)
plt.xlabel('Week after first reading')
plt.ylabel('Dew temperature')
print(N)
def packX(z, poly_order):
    X = np.zeros((len(z), poly_order + 1))
    for i in range(len(X)):
```

```
        for c in range(len(X[i])):
            X[i][c] = math.pow(z[i],c)
    return X

def solveLinearSystem(X,y):
    return np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))

# TEST SCRIPT. DO NOT MODIFY!
X = packX(range(100),3)
y = np.sqrt(np.array(range(100)))
theta = solveLinearSystem(X,y)
print('Check number: ', np.sum(theta))
arr = [1,2,3,10,100]

for p in arr:
    XCoord = packX(weeks_after_start, p)
    YCoord = dewtemp
    theta = solveLinearSystem(XCoord,YCoord)
    output = np.matmul(XCoord, theta) #multipliying matrices
    plt.plot(weeks_after_start, output, label = 'polyOrder ' + str(p))


plt.plot(weeks_after_start, dewtemp, label = "original")
plt.legend()
def solveRidgeRegressionSystem(X,y,rho):
    return np.linalg.solve(X.T @ X + rho * np.eye(X.shape[1]), X.T @ y)

# TEST SCRIPT. DO NOT MODIFY!
X = packX(range(100),3)
y = np.sqrt(np.array(range(100)))
theta = solveRidgeRegressionSystem(X,y,1)
print('Check number: ', np.sum(theta))
arr = [1,2,3,10,100]

for p in arr:
    XCoord = packX(weeks_after_start, p)
    YCoord = dewtemp
    theta = solveRidgeRegressionSystem(XCoord,YCoord,1)
    output = np.matmul(XCoord,theta) #multipliying matrices
```

```
    plt.plot(weeks_after_start, output, label = 'polyOrder' + str(p))

plt.plot(weeks_after_start, dewtemp, label = "original")
plt.legend()
pVal = 50
rho = 0.0005 #bias for RidgeRegression
XCoord = packX(weeks_after_start, pVal)
theta = solveRidgeRegressionSystem(XCoord, dewtemp, rho)
output = np.matmul(XCoord, theta)

foreCastArr = []
CurrentArr = []

for i in range(0, len(weeks_after_start)):
    if weeks_after_start[i] in (weeks_after_start[data['weeks'][0] <= 1.0]):
        foreCastArr.append(output[i])
        CurrentArr.append(dewtemp[i])
plt.plot((weeks_after_start[data['weeks'][0] <= 1.0]), foreCastArr, label =
'forecastDays')
plt.plot((weeks_after_start[data['weeks'][0] <= 1.0]), CurrentArr, label =
'currentDays')

plt.legend()
#I believe my forecast as it follows the trend
```