Leonardo Gomez

May 10, 2018

CS 596

H8: Adaboost algorithm

**Introduction:**

Adaboost is a short adaptive Boosting, in which used in conjunction with many other types of learning algorithms to improve performance. For example, we can multiple ‘weak learners’ such as logistic regression or decision stumps to combine their outputs into a weighted sum that represents the final output. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. One of the downsides of this algorithm is that it is sensitive to noisy data and outliers. However, it is less susceptible to the overfitting problem. Individual ‘learners’ can be weak, but if the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner.

**Implementation:**

The assignment was based on the provided main\_clustering.py file. Some of the main changes are highlighted by the comments PLACEHOLDER 1, 2, and 3 respectably. In the first placeholder, the equation was used. This was to weight each classifier or in our case the decision stump. For the second placeholder, the formula was used to normalize the data. The formula also helps re-assign new weight depending on the selection of the weaker classifiers. This was the only formula that was broken into different parts. The first to take care of the , then multiplied by , and finally to divide by the sum of the training samples. In the final placeholder, the only thing left to do was to run it through was to add the stumpClasiffiy() and pass down all the parameters needed

**Results:**

**For 50 Iterations:**

current error: 0.43

current error: 0.43

current error: 0.395

current error: 0.42

current error: 0.395

current error: 0.4

…

…

…

current error: 0.34

current error: 0.345

current error: 0.345

current error: 0.34

current error: 0.34

current error: 0.32

**For 100 Iterations:**

current error: 0.43

current error: 0.43

current error: 0.395

current error: 0.42

current error: 0.395

current error: 0.4

…

…

…

current error: 0.305

current error: 0.305

current error: 0.305

current error: 0.305

current error: 0.31

current error: 0.3

**For 150 Iterations:**

urrent error: 0.43

current error: 0.43

current error: 0.395

current error: 0.42

current error: 0.395

current error: 0.4

…

…

…

current error: 0.29

current error: 0.285

current error: 0.285

current error: 0.285

current error: 0.29

current error: 0.285

**For 500 Iterations:**

current error: 0.43

current error: 0.43

current error: 0.395

current error: 0.42

current error: 0.395

current error: 0.4

…

…

…

current error: 0.315

current error: 0.315

current error: 0.315

current error: 0.315

current error: 0.315

current error: 0.32

**For 1000 Iterations:**

current error: 0.43

current error: 0.43

current error: 0.395

current error: 0.42

current error: 0.395

current error: 0.4

…

…

…

current error: 0.32

current error: 0.325

current error: 0.32

current error: 0.325

current error: 0.32

current error: 0.32

**Observations:**

For the first couple of iterations we notice the same pattern in terms of current error. It can be argued that the repeating results could be due to the nature of randomization. However, towards the end of the iterations we noticed that they all converge around 32% error rate. The only one that gave the best results was 150 iterations with an error rate of 28.5%. My guess is that the sample size and randomization causes the data to become a normal distribution in wish the more iterations it will eventually converge to the same number.

**Code:**

# This script is used to randomly generate a set of data points, and apply the adaboost method to classify these data points.  
  
  
# from numpy import \*  
# import data  
# import adaboost  
  
import numpy as np  
import math  
  
# Read data from Training or Testing file  
def func\_readData(filename, option):  
 if option == 'train':  
 fid = open(filename, 'r')  
  
 label = []  
 data = None  
 while True:  
 fline = fid.readline()  
 if len(fline) == 0: # EOF  
 break  
 label.append(int(fline[0:fline.find(':')]))  
  
 dataNew = []  
 i = fline.find(':') + 1  
 dataNew = [float(fline[i:fline.find(',', i, -1)])]  
 while True:  
 i = fline.find(',', i, -1) + 1  
 if not i:  
 break;  
 dataNew.append(float(fline[i:fline.find(',', i, -1)]))  
 if data is None:  
 data = np.mat(dataNew)  
 else:  
 data = np.vstack([data, np.mat(dataNew)])  
 fid.close()  
 return data, label  
 elif option == 'test':  
 fid = open(filename, 'r')  
 data = None  
 while True:  
 fline = fid.readline()  
 if len(fline) == 0: # EOF  
 break  
 dataNew = []  
 i = 0  
 while True:  
 dataNew.append(float(fline[i:fline.find(',', i, -1)]))  
 i = fline.find(',', i, -1) + 1  
 if not i:  
 break  
 if data is None:  
 data = np.mat(dataNew)  
 else:  
 data = np.vstack([data, np.mat(dataNew)])  
 fid.close()  
 return data  
 else:  
 print  
 'Wrong input parameter!'  
  
  
# function for building weak classifiers, i.e.: stump function  
  
def buildWeakStump(d, l, D): # (data, label, weight)  
 dataMatrix = np.mat(d)  
 labelmatrix = np.mat(l).T  
 m, n = np.shape(dataMatrix)  
 numstep = 10.0  
 bestStump = {}  
 bestClass = np.mat(np.zeros((5, 1)))  
 minErr = np.inf  
 for i in range(n):  
 datamin = dataMatrix[:, i].min()  
 datamax = dataMatrix[:, i].max()  
 stepSize = (datamax - datamin) / numstep  
 for j in range(-1, int(numstep) + 1):  
 for inequal in ['lt', 'gt']:  
 threshold = datamin + float(j) \* stepSize  
 predict = stumpClassify(dataMatrix, i, threshold, inequal)  
 err = np.mat(np.ones((m, 1)))  
 err[predict == labelmatrix] = 0  
 weighted\_err = D.T \* err;  
 if weighted\_err < minErr:  
 minErr = weighted\_err  
 bestClass = predict.copy()  
 bestStump['dim'] = i  
 bestStump['threshold'] = threshold  
 bestStump['ineq'] = inequal  
 return bestStump, minErr, bestClass  
  
# Boosting Algorithm  
  
def train(data, label, numIt=1000):  
 eps = 10 \*\* -16  
 weakClassifiers = []  
 # m is the number of samples  
 m = np.shape(data)[0]  
 # sample weights, 1/m at the beginning  
 D = np.mat(np.ones((m, 1)) / m)  
  
 estStrong = np.mat(np.zeros((m, 1)))  
 for i in range(numIt):  
 # bestStump: weak classifier; error: error rate  
 bestStump, error, classEstimate = buildWeakStump(data, label, D)  
  
 ##### PLACEHOLDER 1 START ###  
 # calculate the weight of the selected decision stump based on its error rate  
 alpha = float(.5 \* np.log((1.0-error)/(error + eps)))  
 ##### PLACEHOLDER 1 End ###  
  
 # add one more field to bestStump, i.e. classifier weight  
 bestStump['alpha'] = alpha  
 # add bestStump to the list of weak classifiers  
 weakClassifiers.append(bestStump)  
  
 ##### PLACEHOLDER 2 START ###  
 # calculate sample weights (of all samples)  
 # set sample weights  
 # temporary variable to hold the exponent  
 temp = np.multiply(-1 \* alpha \* np.mat(label).T, classEstimate)  
 D = np.multiply(D, np.exp(temp))  
 # normalize D  
 D = D / D.sum()  
 ##### PLACEHOLDER 2 End ###  
  
 estStrong += classEstimate \* alpha  
  
 EnsembleErrors = np.multiply(np.sign(estStrong) != np.mat(label).T, np.ones((m, 1))) # Converte to float  
  
 errorRate = EnsembleErrors.sum() / m  
  
 print("current error: {}".format(errorRate))  
  
 if errorRate == 0.0:  
 break  
 return weakClassifiers  
  
# Use a weak classifier, i.e. a decision stump, to classify data  
  
def stumpClassify(datamat, dim, threshold, inequal):  
 res = np.ones((np.shape(datamat)[0], 1))  
 if inequal == 'lt':  
 res[datamat[:, dim] <= threshold] = -1.0  
 else:  
 res[datamat[:, dim] > threshold] = -1.0  
 return res  
  
# Applying an adaboost classifier for a single data sample  
  
def adaboostClassify(dataTest, classifier):  
 dataMatrix = np.mat(dataTest)  
 m = np.shape(dataMatrix)[0]  
 estStrong = np.mat(np.zeros((m, 1)))  
 for i in range(len(classifier)):  
 ##### PLACEHOLDER 3 START ###  
 # call the function stumpClassify()  
 classEstimate = stumpClassify(dataMatrix, m, estStrong, classifier)  
 # accumulate all predictions  
 estStrong += classifier[i]['alpha'] \* classEstimate  
 ##### PLACEHOLDER 3 START ###  
 return np.sign(estStrong)  
  
  
# Applying an adaboost classifier for all testing samples  
def test(dataSet, classifier):  
 label = []  
 for i in range(np.shape(dataSet)[0]):  
 label.append(adaboostClassify(dataSet[i, :], classifier))  
 return label  
  
#############. main ##################  
# The data files "train.txt" and "test.txt" are randomly generated by the function randomData() and are used to test your developed codes.  
  
trainData, label = func\_readData('train.txt', 'train')  
testData = func\_readData('test.txt', 'test')  
  
# training  
classifier = train(trainData, label, 150)  
print('done training\n')  
# testing  
test(testData, classifier)  
print('done testing\n')