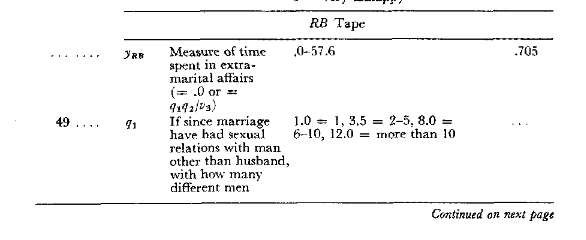
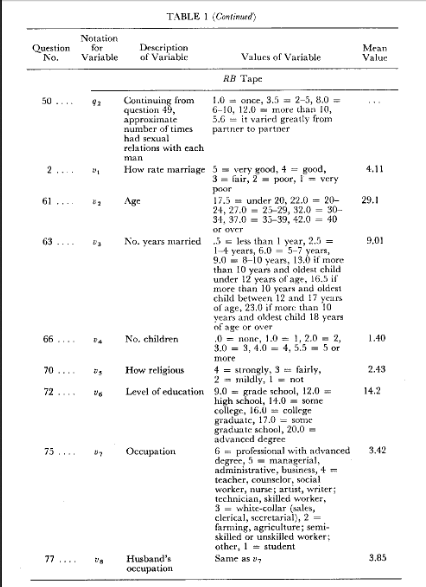
**McMaster University**

**SFWR TECH 4DM3 Course Project**

**Comparing Classifiers**

**Matthew Fischer, Yury Stanev**

1. ***DataSet***

*The dataset we used for this project is taken directly from a study conducted at Yale University by Ray C. Fair in 1978. The purpose of the report was to analyze two separate questionnaires in order to find trends that increased a person’s chances of infidelity. The first survey conducted by Psychology Today(PT) includes both male and female participants. The second survey by Redbook(RB) in 1974 only consists of women. The Redbook study was chosen for our data mining experiment, since this study contained 6366 entries in data, while the Psychology Today study only contained about 600 results. Yrb seen as a category in the left figure describes how much time the subject spent cheating and was adjusted from 0 for “not cheated,” and 1 for “cheated.” This was achieved in all of our programs through an if statement categorizing all values above 0 as 1 and all values equal to 0 as 0.*

*The RB data was separated manually from the PT data. They resided together within AffairData\affairs.txt . The RB data was copied without headings or annotation into AffairData\RbTapeData.*

*RB data: 14 variables per line; they are:*

*identifier, constant, v1, v2, v3, v4, v5, v6, not used, v7, v8, yRB, not used, not used*

*3. 1. 3. 32.0 9.0 3.0 3. 17.0 40.0 2. 5. 0.1111111 0. 1. -> Entry one from RbTapeData*

*Above shows the first entry of the Rb tape along with the explanation of columns provided in the original affairs.txt file. Space was chosen as a delimiter, but we had to solve the problem that periods appeared at the end of some column entries. We could not use dot delimiters as there are floating point numbers in these results. Due to this we created a function with NearestNeighborsAffair.py called “cleanData(filename, filename2).”*

*def cleanData(filename, filename2): ##Remove periods at the end of objects [ we are using space as a delimeter]*

*infile = open(filename, 'r')*

*outfile = open(filename2, 'w')*

*data = infile.read()*

*listdat = list(data)*

*# print listdat*

*i = 0*

*for a in listdat:*

*if listdat[i] == "." and listdat[i+1] == " ":*

*listdat[i] = " "*

*i+=1*

*i = 0*

*for a in listdat:*

*outfile.write(listdat[i])*

*i+=1*

*This function accepts two variables an input and output text file. The input file is the “dirty data” to be “cleaned” of periods that are not used as decimal points and to repost the data to a clean text file. The function creates and iterates through a list of every character and space in the dirty text file. Every time a period is found it than checks the next character for a space and if it finds one deletes the period from the list. This list is than written to the new file. In our case this was AffairData\CleanRbTapeData.txt. The file was then converted to a matrix and after the class labels were converted to 0’s and 1’s for not cheated and cheated, unused columns were deleted along with the old Yrb, appending the classLabelVector of 1’s and 0’s to the end of the matrix. Column 0 which contained labels was kept in order to seek out old test points and delete them after the test matrix was chosen.*

*25% of the 6366 data entries were used for testing amounting to about 1591 test points. The tape data was organized in such a way that the first 30% of the data consisted of class 1 or cheated, while the rest was class 0 (did not cheat). In order to circumvent this and ensure that we had an equal amount of class 1’s and 0’s, we split the matrix we collected from the file into two separate matrices for each class. We then randomized these, took 25% of the size of the original matrix and divided it by 2 to receive a length of 12.5% the original size of the matrix. A test matrix was then constructed using the 12.5% of the size of the original matrix as an index for both the class 1 and 0 halves in order to make 25%.*

*returnMat= np.delete(returnMat, [1,8,11,12,13],axis=1) #Delete unused columns, but hold on to ID in order to subtract test enteries from main data*

*returnMat = np.c\_[returnMat, classLabelVector] #Append new vector adjusted to binary values*

*#Create a test data matrix...*

*i=0*

*#Split matrix into two matrices one for true and one for false*

*for a in classLabelVector:*

*if classLabelVector[i] == 0:*

*break*

*i+=1*

*testMatTrue = returnMat[:i,:]*

*testMatFalse = returnMat[i:,:]*

*np.take(testMatTrue, np.random.permutation(testMatTrue.shape[0]), axis=0, out=testMatTrue)*

*np.take(testMatFalse, np.random.permutation(testMatFalse.shape[0]), axis=0, out=testMatFalse)*

*sizetestMat = int(len(classLabelVector))\*0.25*

*p = int(sizetestMat/2)*

*testMat = np.r\_[testMatTrue[:p,:], testMatFalse[:p,:]]*

*np.take(testMat, np.random.permutation(testMat.shape[0]), axis=0, out=testMat)*

*lengthT = int(testMat.shape[0])*

*lengthDat = int(returnMat.shape[0])*

*dataMat = returnMat*

*i=0*

*q=0*

*#Delete Entries Reserved for testing*

*for i in range(lengthT):*

*for q in range(lengthDat):*

*if testMat[i,0] == returnMat[q,0]:*

*dataMat = np.delete(dataMat, [q], axis=0)*

*#The appending of the classLabelVector along with the creation of the test matrix. Entries that are in the testMatrix are then deleted from the trainingMatrix. The code is within the function file2matrixAffair and a similar version of this function is found within every classifier we used.*

*At this point in the code the handling of the training and testing data begins to diverge between the different types of classifiers. K Nearest Neighbors begins to cross-validate for k, creating a new set of CV-test data in the same manner as the testMatrix, however not deleting the training entries as it is not used outside of the crossvalidation function. KNN also requires the matrices to be split into nominal and numeric matrices which will be described in a later section. The matrix data was conditioned before use so that all the nominal data points would range from 0 onward in increments of one. This was especially important for v6, the parameter that gauged the study subjects level of education ranging from 9.0 =grade school, 12.0 = high school, 14.0 = some college, 16.0 = college grad, 17.0 = some graduate school, and 20.0 = advance degree . Since there were no numeric values in between these, this attribute was scaled from 0-5 while other values like v1(rate marriage 1-5) and v5(How religious 1-4) just had their values negated by 1.*

*Adaboost and regression both required their train and test matrices to be converted into vectors after the initial matrix manipulation in order to work with their classifiers. Adaboost also required the classes to be converted to -1 for not cheated and 1 for cheated( as opposed to 0 and 1).*

1. ***Classifiers***

*The data we classified ended up being extremely partisan and all of our classifiers consistently achieved around 100%. The 3 classifiers chosen for this project are: K Nearest Neighbors, Logistical Regression and Adaboost. Unfortunately only Logistical Regression could score low enough to show a comparable ROC curve, while Adaboost consistently achieved 100% accuracy displaying ROC curves with an AUC = 1. That being said, the function required to generate these ROC curves is still shown, along with a few extra for regression to show the difference gradient ascension has over a couple of iterations. The code for generating ROC is present within the AdaboostAffair.py file, and a new one is generated every time it is run.*

***2.1 K-Nearest Neighbors Classifier***

***Cross-Validation***

def dfoldCrossValid(testMat, foldNum):

i=0

errRate = zeros((foldNum,2))

totalerrRate=0

for i in range(foldNum):

for q in range(foldNum):

testVal = int(testMat.shape[0]/foldNum) #Calculate the range of the testdata points

trainDat1 = testMat[(testVal\*(q+1)):,:] #Later section of train data after test indices

trainDat2 = testMat[:(testVal\*q), :] #train dat before test dat indices

testDat = testMat[(q\*testVal):((q+1)\*testVal),:] #Testdat ranges between the end of trainDat2 and the beginning of trainDat1

trainDat = np.r\_[trainDat1, trainDat2] #Append trainDat1&2

#Create Labels

trainDatLabels = trainDat[:,8]

testDatLabels = testDat[:,8]

a,b,c,d,e,f,g,h,z = classTest(trainDat, trainDatLabels, testDat, testDatLabels, ((i+1)\*5)+i\*15)

#Only value required is a at this point, the rest are dummy variables

totalerrRate += a

errRate[i,0 ] = (totalerrRate/double(q+1))

errRate[i,1] = ((i+1)\*5+i\*15)

totalerrRate = 0

errRatesort = errRate[errRate[:,0].argsort()] ##Sort values to select best k value

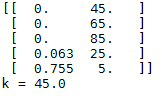
print errRatesort

k = errRatesort[0,1]

print "k = %r" %(k)

return int(k)

Cross-Validation in the KNN program works through passing a CvTestMat from the file2matrixAffair function. The CvTestMat is created in the same way that the test matrix is created, however the entries that the CvTestMat represent from the original data matrix are not deleted, as cross-validation is a separate isolated event from the actual training and testing. The test matrix is also created before the CvTestMat to ensure the test points are not present within the data matrix while chosing cross-validation points.

In order to describe how this function works, let’s use an example of foldNum = 5. foldNum divides the size of the data by 5 so if we have a CvTestMat sized 1000 columns we will have a test point size of 1000/5=200 and a training data point of 1000-(1000/5) = 800. The first for loop will iterate from 0-4 and will test for 5 k values, in this case they will be k=5,k=25, k=45, k=65 and k = 85. The second for loop will also run for 5 iterations from 0-4 changing the test data and train data so that all sections of 200 are tested on all other data points within the CvTestMat. After the nested loop, the error rates are averaged together and placed in a matrix. All k values are placed within a matrix when the the k value calculation is complete. The first column is the averaged error score and the second column is the k value. These values are sorted using argsort by the errorRate from lowest to highest. If there is a tie, the lowest k will be chosen.

* An example of the output on the Ipython prompt in spyder after cross-validation is completed. This test has chosen k= 45

The K value is than automatically entered into the classifier.

**Computational Times**

We will rate testing and training together for this algorithm as these operations seem synonymous within k-Nearest Neighbors. Distance calculation occurs between the training points and the testing points, so this could be seen as both training and testing.

def classifyMixed(norminX, normDataSet,nominX,nomDataSet, labels, k): \*n from classTest

#Distances Euclidean

dataSetSize = normDataSet.shape[0] 2ops \*n

diffMat = tile(norminX, (dataSetSize, 1)) – normDataSet 3ops\*n

sqDiffMat = diffMat\*\*2 2ops \*n

sqDistances = sqDiffMat.sum(axis=1) 2ops \*n

distances = sqDistances\*\*0.5 2ops \*n

#Distances Nominal

metric = np.array([]) 2ops \*n

Row = int(dataSetSize) 3ops \*n

Column = double(len(nomDataSet[0,:])) 4ops \*n

for m in range(Row): m = 3n train data is 3\*testdata \*n

entity = np.where((nomDataSet[m,:] != nominX)) 3ops \*3n \*n

metric = np.append(metric,(len(entity[0])/Column)) 5ops\*3n \*n

#Mixed Dissimalairities

mixedD = np.mean(np.array([distances, metric]), axis=0) 3ops \*n

sortedDistIndices = mixedD.argsort() n2 \*n

voteIlabel = [] #VoteIlabel converted to vector in order to save voters for ROC graph 1op \*n

classCount={} 1op \*n

for i in range(k): \*k \*n

voteIlabel.append(labels[sortedDistIndices[i]]) 3ops\*k \*n

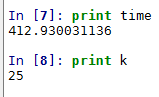
classCount[voteIlabel[i]] = classCount.get(voteIlabel[i], 0) + 1 8ops \* k\* n

sortedClassCount = sorted(classCount.iteritems(),key=operator.itemgetter(1), reverse=True) k2 ops \*n

#For ROC collect NN information those who supported a and b out of k {threshold} append to vector within classtest

Majority = (sum(float(num) == 1 for num in voteIlabel))/float(k) 5ops \*n

return sortedClassCount[0][0], Majority 4ops \*n



-> Time is in seconds for k=25 and represents the time it takes for the above function to run while testing.

**Complexity**

After counting primitive operations, a figure of **ops = n3 + nk2 +24n + 8k +32** came as the final result. There was a value m representing the loops iterating through all of the train data, but this however could be represented as 3n as the test data is a third the size of the training data given it was created from ¼ the original datamatrix, while the training data was the other ¾ . K represents the variable for nearest neighbors calculations and sorting. Alltogether the classifyMixed function is multiplied by the number of test vectors. This tallies the complexity to approximately **O(n3)** for both the training and testing. Testing seems indistinguishable from training in k-NearestNeighbors, so we compared them together.

**Confusion Matrix**



* **TP, FP, FN, TN after k=25 run with error rate of 0.943%**



* **Error Rate after k=45 run = 2.075%**

|  |  |
| --- | --- |
| **FP = 0** | **TP = 780** |
| **TN = 795** | **FN = 15** |

**Confusion Matrix KNN for K=25**

**2.2 Logistic Regression**

**Computational Times**

**Training Algorithm:**

**def stocGradAscent1(dataMatrix, classLabels, numIter=150):**

**m,n = shape(dataMatrix) 3ops**

**weights = ones(n) 2ops**

**strengths = mat(zeros((m,1))) 2ops**

**for j in range(numIter): I**

**dataIndex = range(m) 2ops \* I**

**for i in range(m):\* nI**

**alpha = 4/(1.0+j+i)+0.01 5ops \*nI**

**randIndex = int(random.uniform(0,len(dataIndex))) 2ops \*nI**

**strengths[i] = (sum(dataMatrix[randIndex]\*weights)) 4ops \*nI**

**h = sigmoid(sum(dataMatrix[randIndex]\*weights)) 8ops \*nI**

**error = classLabels[randIndex] – h 3ops \*nI**

**weights = weights + alpha \* error \* dataMatrix[randIndex] 5ops \* nI**

**del(dataIndex[randIndex]) 2ops \* nI**

**return weights 1op**

**def sigmoid(inX):**

**return 1.0/(1+exp(-inX)) 4ops**

**Testing Algorithm(within regClassify function):**

**testStart = time.clock()**

**for i in range(sizeTrainMat):**

**classifyAns = classifyVector(array(testMat[i]), trainWeights)**

**strengths.append(sum(array(testMat[i])\*trainWeights))**

**print "The classifier came back with %r, the actual answer is %r" %(classifyAns, testLabels[i])**

**if int(classifyAns)!= int(testLabels[i]):**

**errorCount+=1**

**if (classifyAns == 1 and testLabels[i]==1):**

**TP +=1**

**elif (classifyAns == 1 and testLabels[i] == 0):**

**FP +=1**

**elif (classifyAns == 0 and testLabels[i] == 1):**

**FN +=1**

**elif (classifyAns == 0 and testLabels[i] == 0):**

**TN +=1**

**errorRate = (float(errorCount)/numTestVecs)**

**print "the error rate of this test is: %f" % errorRate**

**testStop = time.clock()**

**testTime = testStop-testStart**

**def classifyVector(inX, weights): 12 ops \*n**

**prob = sigmoid(sum(inX\*weights))**

**if prob > 0.5: return 1.0**

**else: return 0.0**

**def sigmoid(inX):**

**return 1.0/(1+exp(-inX))**

**Error Rate will improve with the number of iterations of Gradient Ascension, we will log the time it takes to achieve no error and show the difference from 1, 2 and 20 iterations.**



**1 Iteration in Seconds:**



**2 Iterations in Seconds:**



**20 Iterations in Seconds:**

**As expected the iterations take around the same amount of time to execute and have no effect on the testing time.**

**Complexity**

**Training Data:**

Counting primitive operations on the training algorithm gave a result of 29NI+2I + 8 where N is the size of the training data and I is the number of iterations. This can all be simplified by ignoring constants and saying that our training sequence for regression is O(n2). It is unlikely that the number of iterations will ever exceed the number of training points with any amount of data, so NI<=n2.

**Testing Data:**

The testing algorithm only consists of one loop spanning the size of the training matrix. 12n operations have been counted as the function classifyVector inside of the loop just compares the strengths generated through training with the testMatrix, generating an answer between 1 and 0 with the sigmoid function. Testing can be simplified to O(n).

**Confusion Matrices:**

**1 Iteration of Gradient Ascent**



|  |  |
| --- | --- |
| **FP = 491** | **TP = 795** |
| **TN = 304** | **FN = 0** |

**2nd Iteration of Gradient Ascent**



|  |  |
| --- | --- |
| **FP = 114** | **TP = 795** |
| **TN = 681** | **FN = 0** |

**20th Iteration of Gradient Ascent**



|  |  |
| --- | --- |
| **FP = 0** | **TP = 795** |
| **TN = 795** | **FN = 0** |

**It seems this classifier does an extremely well job of classifying whether the person did cheat, but does have a tendency to misclassify others in the same category. This problem is alleviated with more iterations of gradient ascent at the cost of processing time.**

**ROC Graphs**

**Function for plotting ROC graphs Regression and Adaboost:**

**def plotROC(predStrengths, classLabels, title):**

**import matplotlib.pyplot as plt**

**cur = (1.0,1.0) #cursor**

**ySum = 0.0 #variable to calculate AUC**

**numPosClas = sum(array(classLabels)==1.0)**

**yStep = 1.0/float(numPosClas); xStep = 1.0/float(len(classLabels)-numPosClas)**

**sortedIndicies = predStrengths.argsort()#get sorted index, it's reverse**

**fig = plt.figure()**

**fig.clf()**

**ax = plt.subplot(111)**

**#loop through all the values, drawing a line segment at each point**

**for index in sortedIndicies.tolist()[0]:**

**if classLabels[index] == 1.0:**

**delX = 0; delY = yStep;**

**else:**

**delX = xStep; delY = 0;**

**ySum += cur[1]**

**#draw line from cur to (cur[0]-delX,cur[1]-delY)**

**ax.plot([cur[0],cur[0]-delX],[cur[1],cur[1]-delY], c='b')**

**cur = (cur[0]-delX,cur[1]-delY)**

**ax.plot([0,1],[0,1],'b--')**

**plt.xlabel('False positive rate'); plt.ylabel('True positive rate')**

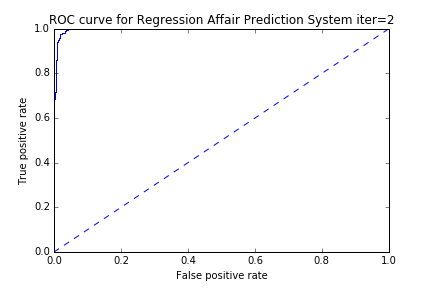
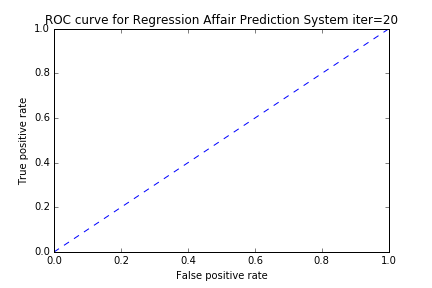
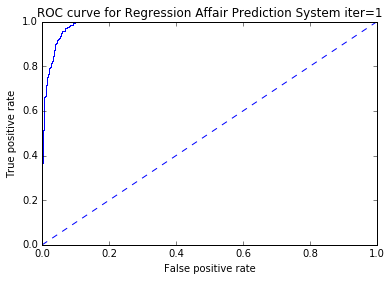
**plt.title(title)**

**ax.axis([0,1,0,1])**

**plt.show()**

**fig.savefig("OutputData\ROC\Regression\RegGradAsc20Iter.png")**

**print "the Area Under the Curve is: ",ySum\*xStep**



**Strengths used = the inputs to the sigmoid function from the classifier.( sum(array(testMat[i])\*trainWeights))**

**Labels used = Test labels**

**Function Call:**

**plotROC(strengths, testLabels, 'ROC curve for Regression Affair Prediction System iter=20') Iter 1 the Area Under the Curve is: 0.986872354733**

Iter 2 the Area Under the Curve is: 0.999636090345

**Iter 20 Area Under the Curve isL 1.0**

**2.3 Adaboost**

**Computational Times**

**Training Algorithm**

def adaBoostTrainDS(dataArr,classLabels,numIt=40):

weakClassArr = [] 1op

m = shape(dataArr)[0] 3ops

D = mat(ones((m,1))/m) #init D to all equal 3ops

aggClassEst = mat(zeros((m,1))) 2ops

for i in range(numIt): \*I

bestStump,error,classEst = buildStump(dataArr,classLabels,D)#build Stump (21+5771C) \* I

#print "D:",D.T

alpha = float(0.5\*log((1.0-error)/max(error,1e-16)))#calc alpha, throw in max(error,eps) to account for error=0 6ops \*I

bestStump['alpha'] = alpha 3ops \*I

weakClassArr.append(bestStump) #store Stump Params in Array 2ops \* I

#print "classEst: ",classEst.T

expon = multiply(-1\*alpha\*mat(classLabels).T,classEst) #exponent for D calc, getting messy 7ops \* I

D = multiply(D,exp(expon)) #Calc New D for next iteration 3ops \* I

D = D/D.sum() 3ops \* I

#calc training error of all classifiers, if this is 0 quit for loop early (use break)

aggClassEst += alpha\*classEst 2ops

#print "aggClassEst: ",aggClassEst.T

aggErrors = multiply(sign(aggClassEst) != mat(classLabels).T,ones((m,1))) 7ops \* I

errorRate = aggErrors.sum()/m 3ops \* I

print "total error: ",errorRate 1op \* I

if errorRate == 0.0: break 1op \* I

return weakClassArr, aggClassEst 2ops

def stumpClassify(dataMatrix,dimen,threshVal,threshIneq):#just classify the data 14ops

retArray = ones((shape(dataMatrix)[0],1))

if threshIneq == 'lt':

retArray[dataMatrix[:,dimen] <= threshVal] = -1.0

else:

retArray[dataMatrix[:,dimen] > threshVal] = -1.0

return retArray

def buildStump(dataArr,classLabels,D):

dataMatrix = mat(dataArr); labelMat = mat(classLabels).T 7ops

m,n = shape(dataMatrix) 3ops

numSteps = 10.0; bestStump = {}; bestClasEst = mat(zeros((m,1))) 5ops

minError = inf #init error sum, to +infinity 3ops

for i in range(n):#loop over all dimensions \*c

rangeMin = dataMatrix[:,i].min(); rangeMax = dataMatrix[:,i].max(); 8ops \* c

stepSize = (rangeMax-rangeMin)/numSteps 3ops \* c

for j in range(-1,int(numSteps)+1):#loop over all range in current dimension \*c\*12

for inequal in ['lt', 'gt']: #go over less than and greater than \*c\*122

threshVal = (rangeMin + float(j) \* stepSize) 4ops \*c\*122

predictedVals = stumpClassify(dataMatrix,i,threshVal,inequal)#call stump classify with i, j, lessThan 14ops \*c\*122

errArr = mat(ones((m,1))) 3ops \*c\*122

errArr[predictedVals == labelMat] = 0 3ops \*c\*122

weightedError = D.T\*errArr #calc total error multiplied by D 3ops \*c\*122

#print "split: dim %d, thresh %.2f, thresh ineqal: %s, the weighted error is %.3f" % (i, threshVal, inequal, weightedError)

if weightedError < minError: 1op \*c\*122

minError = weightedError 1op \*c\*122

bestClasEst = predictedVals.copy() 2ops \*c\*122

bestStump['dim'] = I 3ops \*c\*122

bestStump['thresh'] = threshVal 3ops\*c\*122

bestStump['ineq'] = inequal 3ops \*c\*122

return bestStump,minError,bestClasEst 3ops \*c\*122

**Testing Algorithm**

def adaClassify(datToClass,classifierArr):

dataMatrix = mat(datToClass)#do stuff similar to last aggClassEst in adaBoostTrainDS 2ops

m = shape(dataMatrix)[0] 3ops

aggClassEst = mat(zeros((m,1))) 3ops

for i in range(len(classifierArr)): \*i

classEst = stumpClassify(dataMatrix,classifierArr[i]['dim'],\

classifierArr[i]['thresh'],\

classifierArr[i]['ineq'])#call stump classify 13ops \* i

aggClassEst += classifierArr[i]['alpha']\*classEst 4ops \* i

# print aggClassEst

signArr = sign(aggClassEst) 3ops

error, TP, FP, FN, TN = errRate(signArr, testLabels) 13ops \* n

return error , aggClassEst, TP, FP, FN, TN

def stumpClassify(dataMatrix,dimen,threshVal,threshIneq):#just classify the data

retArray = ones((shape(dataMatrix)[0],1))

if threshIneq == 'lt':

retArray[dataMatrix[:,dimen] <= threshVal] = -1.0

else:

retArray[dataMatrix[:,dimen] > threshVal] = -1.0

return retArray

**def errRate(sign, testLabels): 13ops \* n**

**errRate = 0**

**i=0**

**TP=0**

**FP=0**

**FN=0**

**TN=0**

**size = int(len(sign))**

**for i in range(size):**

**print "The classifier came back with %r, the real answer is %r" %(sign[i], testLabels[i])**

**if sign[i] != testLabels[i]:**

**errRate+=1**

**if (sign[i] == 1 and testLabels[i]==1):**

**TP +=1**

**elif (sign[i] == 1 and testLabels[i] == -1):**

**FP +=1**

**elif (sign[i] == -1 and testLabels[i] == 1):**

**FN +=1**

**elif (sign[i] == -1 and testLabels[i] == -1):**

**TN +=1**

**print "The total error rate in percent is: %r" %((errRate/float(len(sign)))\*100**

**return errRate, TP, FP, FN, TN**

**Time with 1 Decision Stump:**



**Time with 2 Decision Stumps:**

**Time with 40 Decision Stumps:**



**The data is easily classified by a single weak classifier, our data was likely extremely partisan as all of our classifiers performed exceptionally well. One thing to note with adaboost is that the training time seems to improve with more weak classifiers created. Overhead appears to not increase with the number of stumps whereas the number of gradient ascent iterations does in logistic regression.**

**Complexity**

**Training Algorithm**

**Counting operations result: 5771CI +59I+11**

**N = datasetsize in rows**

**C= columns**

**I = iterations**

**N2 >=5771CI+59I+11**

While the function buildStump has a loop with two nested loops inside of it, the other two loops will be determined by the number of steps which is fixed at 10.0. The ancestor of these two loop’s computation time does depend on the number of columns set by the training data so this will be interpreted as C, whereas it’s children are known to run approximately 12 times. The adaboost training function adaBoostTrainDS runs this function I times depending on the number of iterations set. I is assumed to be always smaller than the train data size (in rows) so we can determine the training sequence to be O(n2). N in this case will be the dataset size.

**Testing Algorithm**

**Counting operations result: 13n+28**

**N =Test data size in rows**

**Counting operations result:**

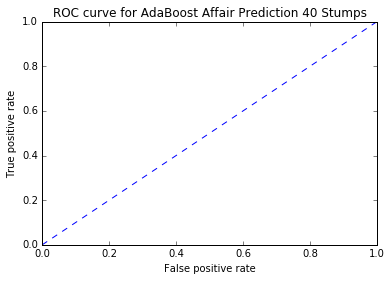
The testing function just iterates through the different classifiers. The amount of time it takes will depend on the amount of the decision stumps but is a mere multiple of the input test data. This will result in n operations by the time the error rate is calculated. The testing algorithm can therefore be classified as **O(n).**

**Confusion Matrix (all iterations >=1)**



|  |  |
| --- | --- |
| **FP = 0** | **TP = 795** |
| **TN = 795** | **FN = 0** |

**ROC Curve (all iterations>=1, used Stumps = 40)**



**the Area Under the Curve is: 1.0**

1. ***Reflections:***

*It would appear adaboost gave us the best results giving us no errors with 1 stump and taking 1.80 seconds to train and test with any amount of stumps.*

*Regression only took about .8 of a second to execute using one iteration of gradient ascent, but the error rate would often shift from 40%-20% depending on how the test and training data was randomized. Overhead in training also accumulated with extra iterations of gradient ascent, accounting for about .6 seconds for each iteration. A 0% error also took about 20 iterations to achieve which took 11 seconds to train on our data set. Adaboost took much longer to test than train, training accounting for only about .2 seconds of execution time. K Nearest Neighbors was extremely complex and inefficient taking almost 413 seconds to run. K Nearest Neighbors also gave us around 1% error regardless of which K value would be selected.*

*Our best classifier was Adaboost, and our worst was K-Nearest Neighbors*

*More ROC graphs can be found in OutputData\ROC as they are save as part of the plotROC function to this folder in png format.*