STAT W4400 | Haoyang Chen | hc2812 | HW3

1.

(1) Implement of train, classify and agg\_class function.

Train a decision stamp using the weighted train data.

First, define a function called learning\_rule to calculate the error rate. Then use optimize function to get the best split. However, the optimize function only do one-dimension optimization, we have three dimensions: the location of the axis j, the split tj in axis j, and the m, thus we use two loops in j and m, getting the optimized tj for each j and m.

# The train function

train <- function(X, w, y){

if (!is.vector(y)){

y <- as.vector(y$V1)

}

n <- dim(X)[1] # the number of observations

d <- dim(X)[2] # the number of features

theta <- c()

error\_rate <- c()

m\_list <- c()

# compute the error rate for a weak learner

learning\_rule <- function(theta\_j, x\_j, weight, m\_stump){

predict\_y <- c()

# predict the classification based on the split point

for (i in c(1: n)){

if (x\_j[i] > theta\_j){

predict\_y[i] <- m\_stump

} else {

predict\_y[i] <- -m\_stump

}

}

error\_rate <- 0

for (i in c(1: n)){

if (y[i] != predict\_y[i]){

error\_rate <- error\_rate + weight[i]

}

}

error\_rate <- error\_rate / (sum(weight))

return(error\_rate)

}

# compute the optimal parameter theta\_j and m for each x

for (j in c(1: d)){

test\_parameter <- c()

test\_error\_rate <- c()

k <- 1

# compute the arg min of cost function

for (m in c(-1, 1)){

optimization <- optimize(learning\_rule, interval = c(min(X[,j]), max(X[,j])), x\_j = X[, j], weight = w, m\_stump = m)

test\_parameter[k] <- optimization$minimum

test\_error\_rate[k] <- optimization$objective

k <- k + 1

}

if(test\_error\_rate[1] < test\_error\_rate[2]){

theta[j] <- test\_parameter[1]

error\_rate[j] <- test\_error\_rate[1]

m\_list[j] <- c(-1)

} else {

theta[j] <- test\_parameter[2]

error\_rate[j] <- test\_error\_rate[2]

m\_list[j] <- c(1)

}

}

location\_j <- which.min(error\_rate)

result\_list = list(j = location\_j, theta = theta[location\_j], mode = m\_list[location\_j])

return(result\_list)

}

Use the definition of the decision stamp to do classification.

# The classify function

classify <- function(X, pars){

label <- (2\*(X[, pars$j] > pars$theta) - 1) \* pars$mode

return(label)

}

Aggregate the weak learners.

# agg\_class function

agg\_class <- function(X, alpha, allPars) {

n <- dim(X)[1]

B <- length(alpha)

labels <- matrix(0, nrow = n, ncol = B)

for (i in c(1: B)){

labels[,i] <- classify(X, allPars[[i]])

}

sum\_label <- labels %\*% alpha

classifier <- as.vector(sign(sum\_label))

return(classifier)

}

Implement AdaBoost using train, classify and agg\_class function.

# The adaBoost function

adaBoost <- function(X, y, B){

if (!is.vector(y)){

y <- as.vector(y$V1)

}

n <- dim(X)[1]

d <- dim(X)[2]

weight\_list <- rep(1/n, d)

alpha <- c()

allPars <- list()

# train routin

for (i in c(1: B)){

allPars[[i]] <- train(X, weight\_list, y)

predict\_y <- classify(X, allPars[[i]])

error\_rate <- sum((predict\_y != y) \* weight\_list) / (sum(weight\_list))

# compute alpha and new weights

alpha[i] <- log((1 - error\_rate) / error\_rate)

weight\_list <- weight\_list \* exp(alpha[i] \* (predict\_y != y))

}

return(list(alpha = alpha, allPars = allPars))

}

As we need to perform k-fold cross-validation in USPS data, thus I defined a cross\_validation function

# cross validation function

cross\_validation <- function(X, y, B\_max, k\_fold){

n <- dim(X)[1]

train\_error\_rate <- matrix(0, nrow = B\_max, ncol = k\_fold)

test\_error\_rate <- matrix(0, nrow = B\_max, ncol = k\_fold)

# split data into k group

num\_of\_group <- round(n/k\_fold)

k\_fold\_groups <- list()

for(k in 1:k\_fold){

ini\_point <- (k - 1) \* num\_of\_group

stop\_point <- k \* num\_of\_group

if(stop\_point < n){

k\_fold\_groups[[k]] <- c(ini\_point: stop\_point)

} else {

k\_fold\_groups[[k]] <- c(ini\_point : n)

}

}

# for each iteration in cross validation, choose train data and test data

for(k in 1:k\_fold){

train.x <- X[-k\_fold\_groups[[k]],]

train.y <- y[-k\_fold\_groups[[k]],]

test.x <- X[k\_fold\_groups[[k]],]

test.y <- y[k\_fold\_groups[[k]],]

ada <- adaBoost(train.x, train.y, B\_max)

allPars <- ada$allPars

alpha <- ada$alpha

# compute test error rate and train error rate for each b

for (b in 1: B\_max){

test\_predict\_y <- agg\_class(test.x, alpha[1:b], allPars = allPars[1:b])

test\_error\_rate[b, k] <- mean(test.y != test\_predict\_y)

train\_predict\_y <- agg\_class(train.x, alpha[1:b], allPars = allPars[1:b])

train\_error\_rate[b, k] <- mean(train.y != train\_predict\_y)

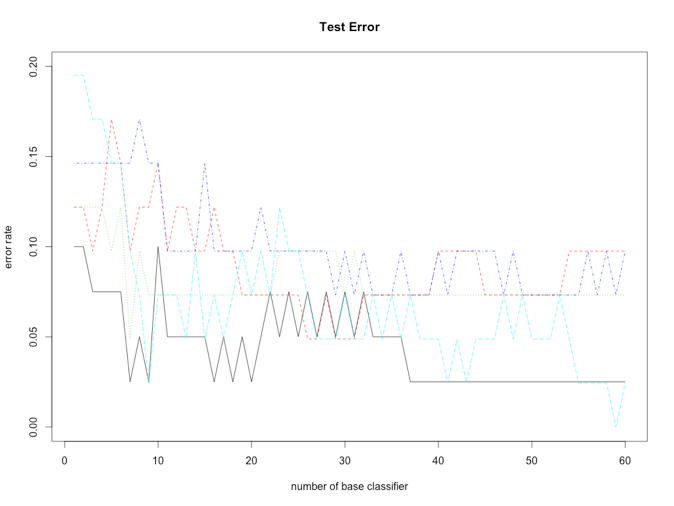
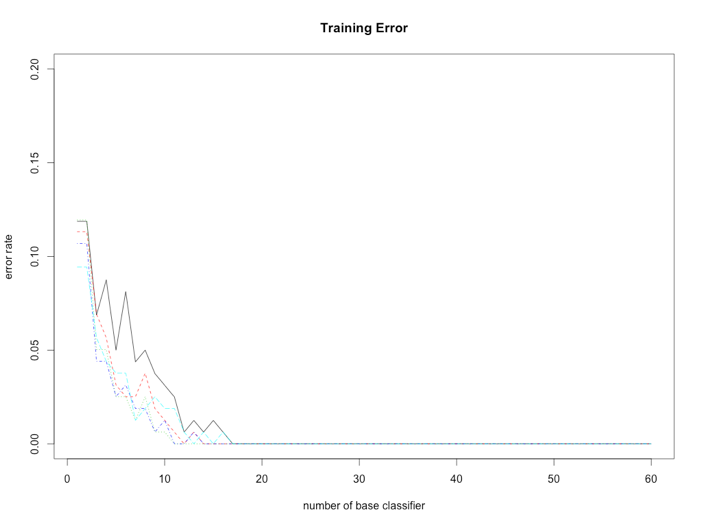
}

}

return(list(train\_error\_rate = train\_error\_rate, test\_error\_rate = test\_error\_rate))

}

The Plot of train error and test error:



As the weak learner increase, train error vanishes quickly. When b = 18, the train error is almost zero. However, when b = 18, although the train error is minimal, increasing the number of weak learner would still decrease the test error. According to the test error graph, I would choose b > 30 and b < 40, as when b > 40, increasing b does not have effect in reducing test error.

2.

(1)

The left one (q = 0.5) encourages sparse solutions, the right one (q = 4) does not encourage sparse solutions.

For q = 0.5, the intersections of the ellipse iso-line of the square-loss and the edge of the penalty are on axis, which means the entry for the respective other axis is zero. In contrast, when q = 4, the entries are even size, not zero.

(2)

q = 0.5: x3 would achieve the smallest cost, since it is intersecting the edge of the penalty and locate at the beta-2 axis.

q = 4: x4 would achieve the smallest cost, since it has the same square-loss with other x’s, while it has the smallest penalty.