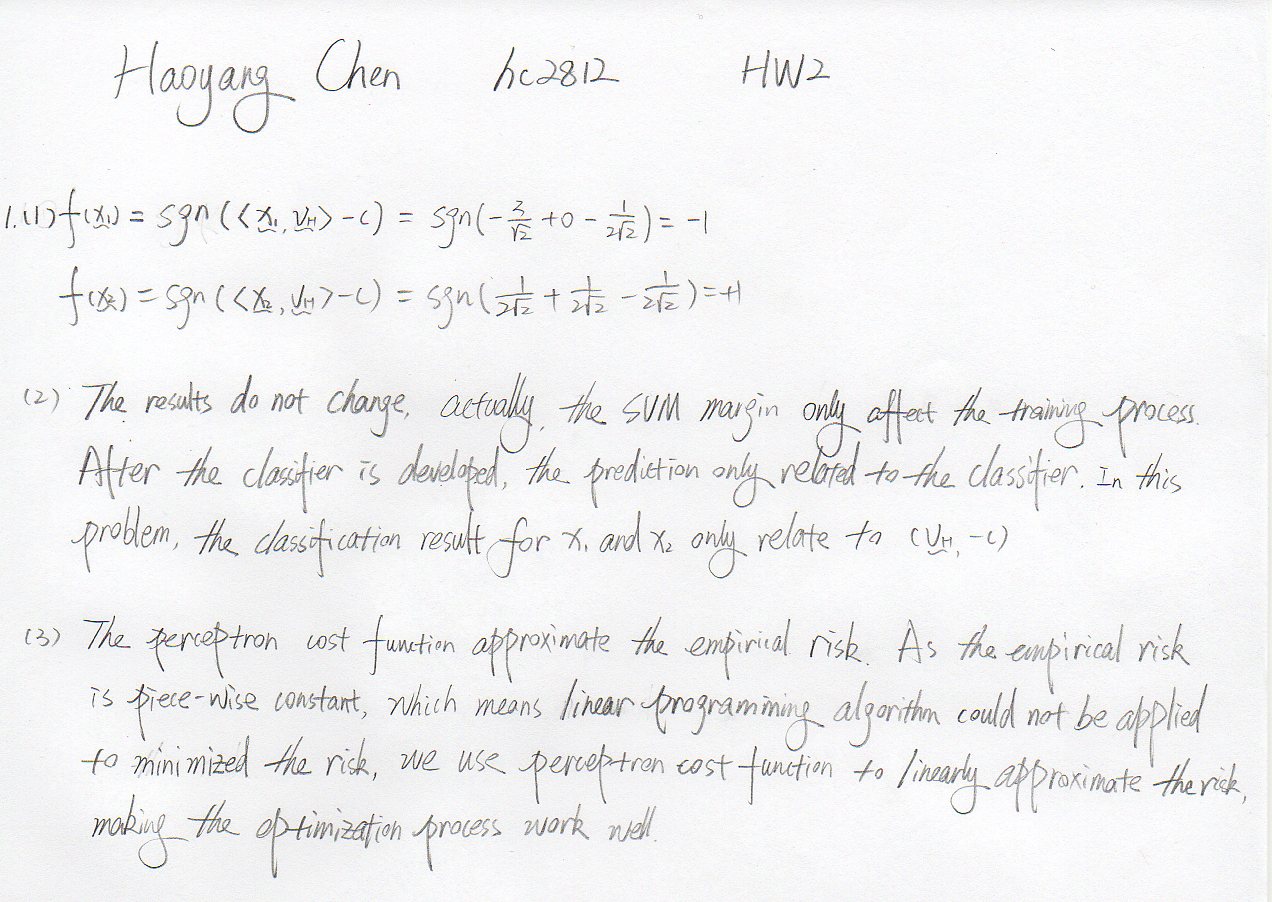
Haoyang Chen | hc2812 | W4400 | HW2



2

(1)

# Input: sample data set S, Perceptron weight vector z = (v, -c)

# Output: predict class label vector y

classify <- function(S, z){

y <- sign(S %\*% z)

y[y == 0] <- 1

return(y)

}

(2)

# Input: sample data set S, class label vector y

# Output: normal vector Z, the history of the normal vector throught the training run Z\_history

perceptrain <- function(S, y){

dimension <- dim(S)[2] # The dimension of <x, 1>

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

Z <- runif(dimension, min = -1, max = 1) # Start at a random z

k <- 1 # Initial the iteration time

Cost <- 100000 # Set initial Perceptron cost function value with a large number

Z\_history <- Z

while((Cost > 0) || (k < 1000)){

# Set Cost = 0 at the begining of each iteration

Cost <- 0

# Set Gradient of the cost function = 0 at the begining of each iteratin

Gradient\_Cost <- rep(0, dimension)

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

x\_vector <- S[i,]

predict\_y <- classify(x\_vector, Z)

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

Cost <- Cost + abs(Z %\*% x\_vector)

Gradient\_Cost <- Gradient\_Cost + (-y[i]) \* x\_vector

}

}

Z <- Z - (1 / k) \* Gradient\_Cost

Z\_history <- rbind(Z\_history, Z)

k <- k + 1

}

rownames(Z\_history) <- c()

return(list(Z = Z, Z\_history = Z\_history))

}

(3)

The error rate of the perceptron classifier on the test data is 0.02, which means the perceptron classifier has high accuracy, it works well.

# Produce a sample data set using z = c(-10, 6, -5) as train set

perceptron\_weight\_vector <- c(-3, 5, -2)

S.train <- fakedata(perceptron\_weight\_vector, 100)

# Build Classifier using perceptron

classifier <- perceptrain(S.train$S, S.train$y)

# Produce a sample data set using z = c(-10, 6, -5) as test set

S.test <- fakedata(perceptron\_weight\_vector, 100)

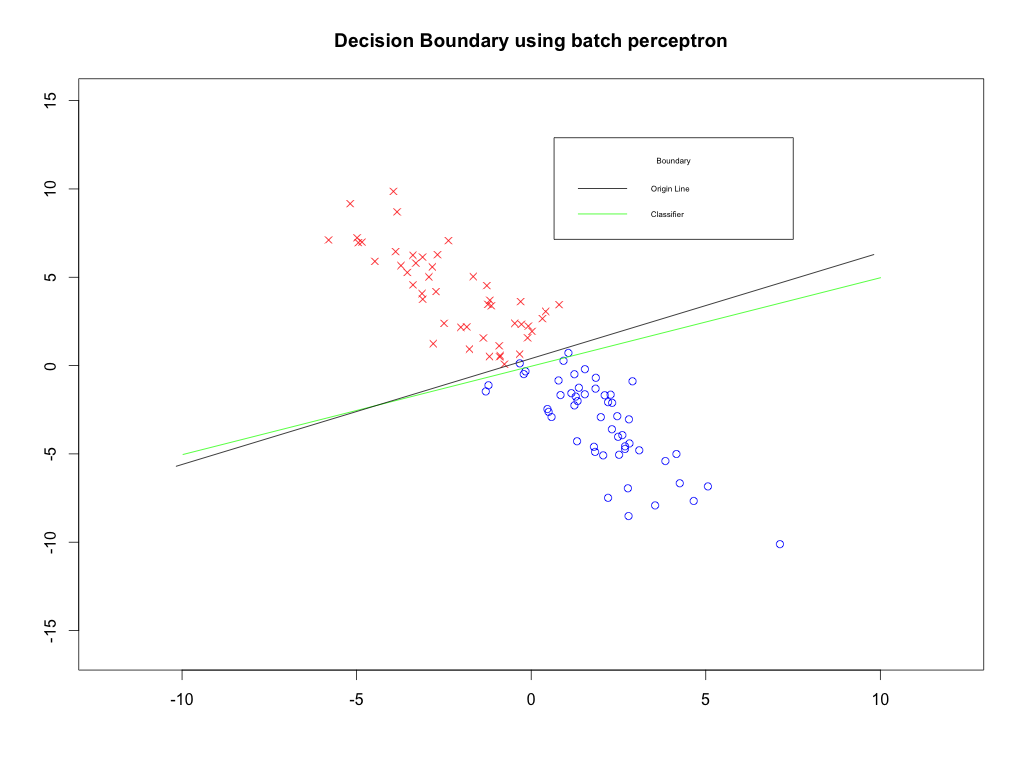
# Predict test data response

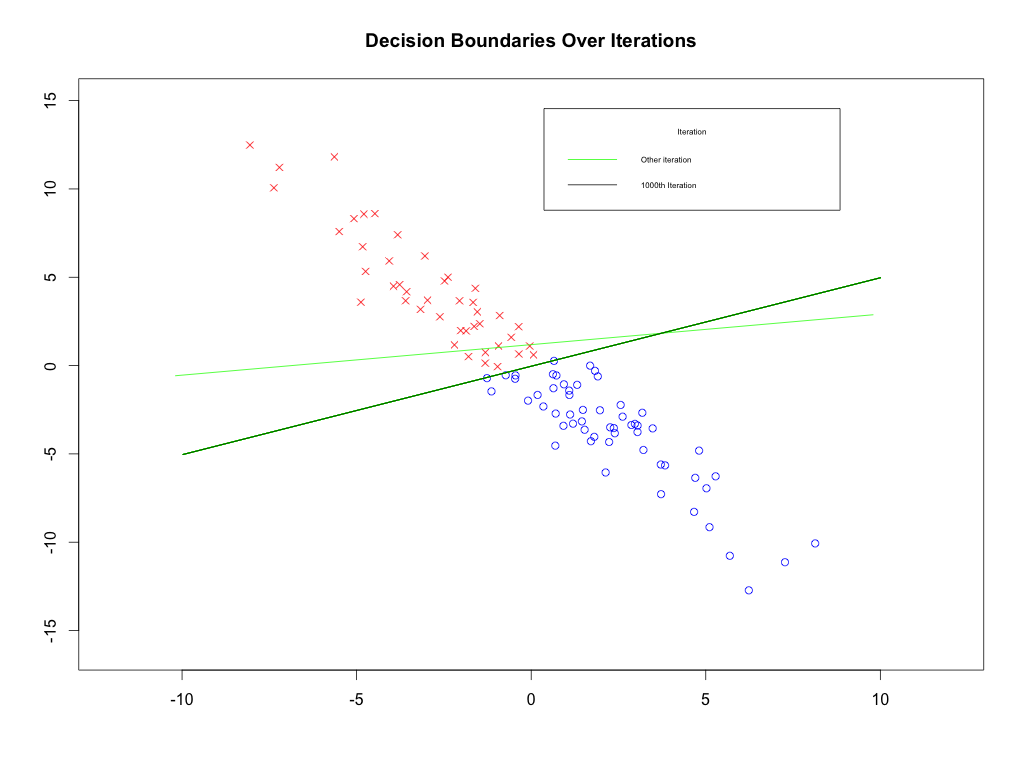
predict\_y <- classify(S.test$S, classifier$Z)

# Calculate error rate

error <- sum(predict\_y != S.test$y) / length(S.test$y)

(4)





# Convert data to 2D corresponding representation

test.pos <- S.test$y == 1

test.data\_pos <- S.test$S[test.pos, 1:2]

test.data\_neg <- S.test$S[!test.pos, 1:2]

# Convert the 3D vectors into corresponding line

vector\_to\_line <- function (vector){

Null\_space <- Null(vector[1:2])

offset <- (-vector[3]) \* vector[1:2] / (vector[1] ^ 2 + vector[2] ^ 2)

x1 <- -10 + offset [1]

x2 <- 10 + offset [1]

y1 <- -10 \* Null\_space[2] / Null\_space[1] + offset[2]

y2 <- 10 \* Null\_space[2] / Null\_space[1] + offset[2]

Hyperplane <- rbind (c(x1, y1), c(x2, y2))

return (Hyperplane)

}

# Draw dots

plot(test.data\_pos, pch = 4, col = 'red', xlim = c(-12, 12), ylim = c(-16, 15), xlab = "", ylab = '')

points(test.data\_neg, pch = 1, col = 'blue')

# The line produce the data

origin\_line <- vector\_to\_line(perceptron\_weight\_vector)

# Classifier Line

classifier\_line <- vector\_to\_line(classifier$Z)

# Draw lines

segments(classifier\_line[1, 1], classifier\_line[1, 2], classifier\_line[2, 1], classifier\_line[2, 2], col = 'green')

segments(origin\_line[1, 1], origin\_line[1, 2], origin\_line[2, 1], origin\_line[2, 2], col = 'black')

legend(locator(1), title="Boundary", c("Origin Line","Classifier"),

lty = 1, col=c('black', 'green'), cex = 0.5)

title(main = "Decision Boundary using batch perceptron")

# split the train data into two piece and transfer into 2D points

train.pos <- S.train$y == 1

train.data\_pos <- S.train$S[train.pos, 1:2]

train.data\_neg <- S.train$S[!train.pos, 1:2]

# Draw dots

plot(train.data\_pos, pch = 4, col = 'red', xlim = c(-12, 12), ylim = c(-16, 15), xlab = "", ylab = '')

points(train.data\_neg, pch = 1, col = 'blue')

# Transfer the 1st, 100th, 200th ..., 1000th iteration to 2D

history\_classifier <- classifier$Z\_history[c(1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000),]

for(i in c(1: 10)){

history\_classifier\_line <- vector\_to\_line(history\_classifier[i,])

segments(history\_classifier\_line[1, 1], history\_classifier\_line[1, 2], history\_classifier\_line[2, 1], history\_classifier\_line[2, 2], col = 'green')

}

# Draw 1000th classifier

history\_classifier\_line <- vector\_to\_line(history\_classifier[11,])

segments(history\_classifier\_line[1, 1], history\_classifier\_line[1, 2], history\_classifier\_line[2, 1], history\_classifier\_line[2, 2], col = 'black')

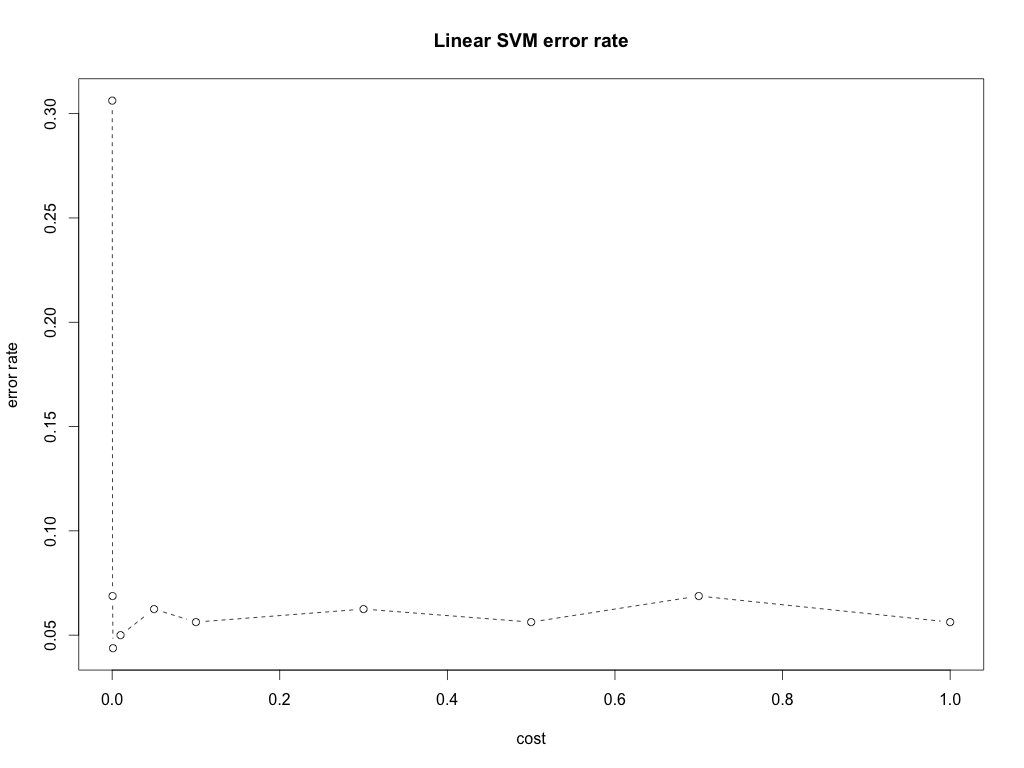
legend(locator(1), title="Iteration", c("Other iteration","1000th Iteration"),

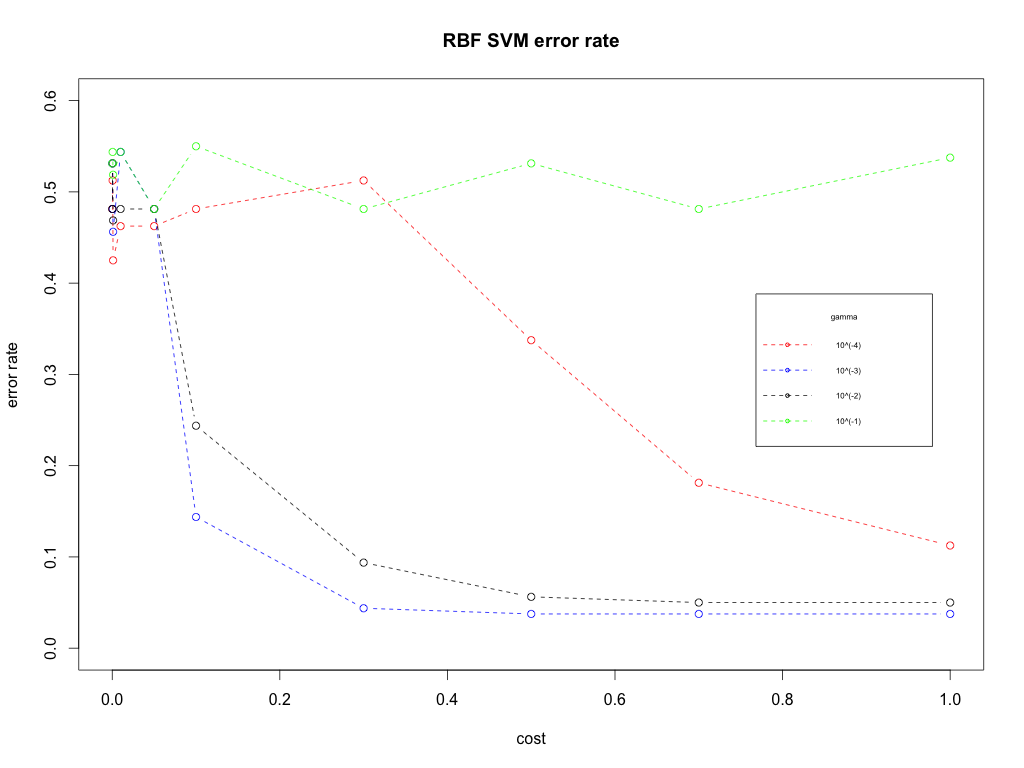
lty = 1, col=c('green', 'black'), cex = 0.5)

title('Decision Boundaries Over Iterations')

3

(a).





(b).

Based on the results above, for linear kernel SVM, we use cost 0.001, and the error rate in test set is 0. For RBF kernel SVM, we use cost 0.5 and gamma 0.001, the error rate in the test set is also 0. As the test set and cross-validation exist randomness, thus, the error rates of the two classifier are not totally exact. But, both of them have very high accuracy, more than 95%.

As the linear kernel SVM has the similar performance with the RBF kernel SVM, a linear SVM would be a good choice for this data because train a linear SVM is more efficient and fast.

library(e1071)

# read data

uspsdata <- read.table('uspsdata.txt')

labels <- read.table('uspscl.txt')

labels <- as.factor(labels$V1)

# Split the data set into train data and test data

train\_num <- sample(1:dim(uspsdata)[1], 0.8 \* dim(uspsdata)[1], replace = F)

test\_num <- setdiff(1:dim(uspsdata)[1], train\_num)

train.x <- uspsdata[train\_num,]

train.y <- labels[train\_num]

test.x <- uspsdata[test\_num,]

test.y <- labels[test\_num]

# Use grid method to set tunning parameter cost and gamma

# The range is according to wiki recommendation

costs <- c(0.0001, 0.0005, 0.001, 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 1)

gammas <- 10 ^ (-4: -1)

# train SVM classifier using linear Kernel using 10-fold cross-validation and record the error v.s each cost

linear\_SVM <- c() # record the cost and error rate

for (cost in costs){

classifier <- svm(train.x, train.y, kernel = 'linear', cost = cost, cross = 10)

error\_rate <- (100 - classifier$tot.accuracy) / 100

linear\_SVM <- rbind(linear\_SVM, c(cost, error\_rate))

}

# draw the plot

plot(linear\_SVM, type = 'b', lty = 2, xlab = 'cost', ylab = 'error rate', main = "Linear SVM error rate")

# select the tunning parameter which has the best performance

# best cost is 0.001

linear\_SVM <- as.data.frame(linear\_SVM)

names(linear\_SVM) <- c('cost', 'error\_rate')

best\_linear\_cost <- as.vector(linear\_SVM[linear\_SVM$error\_rate == min(linear\_SVM$error\_rate),]$cost)

if(length(best\_linear\_cost) > 1){

best\_linear\_cost <- best\_linear\_cost[1]

}

# Use the cost with best performance to train a tuning parameter

linear\_SVM\_model <- svm(train.x, train.y, kernel = 'linear', cost = best\_linear\_cost)

# Compute the error rate of the linear SVM model using test set

linear\_predict\_y <- predict(linear\_SVM\_model, test.x)

linear\_error\_rate <- sum(linear\_predict\_y != test.y) / length(test.y)

# train SVM classifier using RBF Kernel using 10-fold cross-validation and record the error v.s each cost and gamma

radial\_SVM <- c() # record error v.s. cost and gamma

for (cost in costs){

for (gamma in gammas){

classifier <- svm(train.x, train.y, kernel = 'radial', cost = cost, gamma = gamma, cross = 10)

error\_rate <- (100 - classifier$tot.accuracy) / 100

radial\_SVM <- rbind(radial\_SVM, c(cost, gamma, error\_rate))

}

}

# draw the plot

radial\_SVM <- as.data.frame(radial\_SVM)

names(radial\_SVM) <- c('cost', 'gamma', 'error\_rate')

cols <- c('red', 'blue', 'black', 'green')

i <- 1

for (gamma in gammas){

col <- cols[i]

if(i == 1){

plot(radial\_SVM[radial\_SVM$gamma == gamma,]$cost, radial\_SVM[radial\_SVM$gamma == gamma,]$error\_rate, type = 'b', lty = 2, col = col, xlim = c(0, 1), ylim = c(0, 0.6), xlab = 'cost', ylab = 'error rate', main = 'RBF SVM error rate')

}else{

lines(radial\_SVM[radial\_SVM$gamma == gamma,]$cost, radial\_SVM[radial\_SVM$gamma == gamma,]$error\_rate, type = 'b', lty = 2, col = col)

}

i <- i + 1

}

legend(locator(1), title="gamma", c("10^(-4)","10^(-3)", "10^(-2)", "10^(-1)"),

lty=2, pch=1, col=cols, cex = 0.5)

# select the tunning parameter which has the best performance

# best cost is 0.5, best gamma is 0.001

best\_radial\_parameter <- radial\_SVM[radial\_SVM$error\_rate == min(radial\_SVM$error\_rate),]

if(dim(best\_radial\_parameter)[1] > 1){

best\_radial\_cost <- best\_radial\_parameter$cost[1]

best\_radial\_gamma <- best\_radial\_parameter$gamma[1]

}

# Use the cost with best performance to train a tuning parameter

radial\_SVM\_model <- svm(train.x, train.y, kernel = 'radial', cost = best\_radial\_cost, gamma = best\_radial\_gamma)

# Compute the error rate of the RBF SVM model using test set

radial\_predict\_y <- predict(radial\_SVM\_model, test.x)

radial\_error\_rate <- sum(radial\_predict\_y != test.y) / length(test.y)