Haoyang Chen hc2812 HW2

1. (1) $f(\underline{x}_1) = sgn(\langle \underline{x}_1, \underline{v}_H \rangle - c) = sgn(-\frac{3}{\sqrt{2}} + 0 - \frac{1}{2\sqrt{2}}) = -1$

$f(\underline{x}_2) = sgn(\langle \underline{x}_2, \underline{v}_H \rangle - c) = sgn(\frac{1}{2\sqrt{2}} + \frac{1}{2\sqrt{2}} - \frac{1}{2\sqrt{2}}) = +1$

(2) The results do not change, actually, the SVM margin only affect the training process. After the classifier is developed, the prediction only related to the classifier. In this problem, the classification result for x_1 and x_2 only relate to $(\underline{v}_H, -c)$

(3) The perceptron cost function approximate the empirical risk. As the empirical risk is piece-wise constant, which means linear programming algorithm could not be applied to minimized the risk, we use perceptron cost function to linearly approximate the risk, making the optimization process work well.

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)
}
```

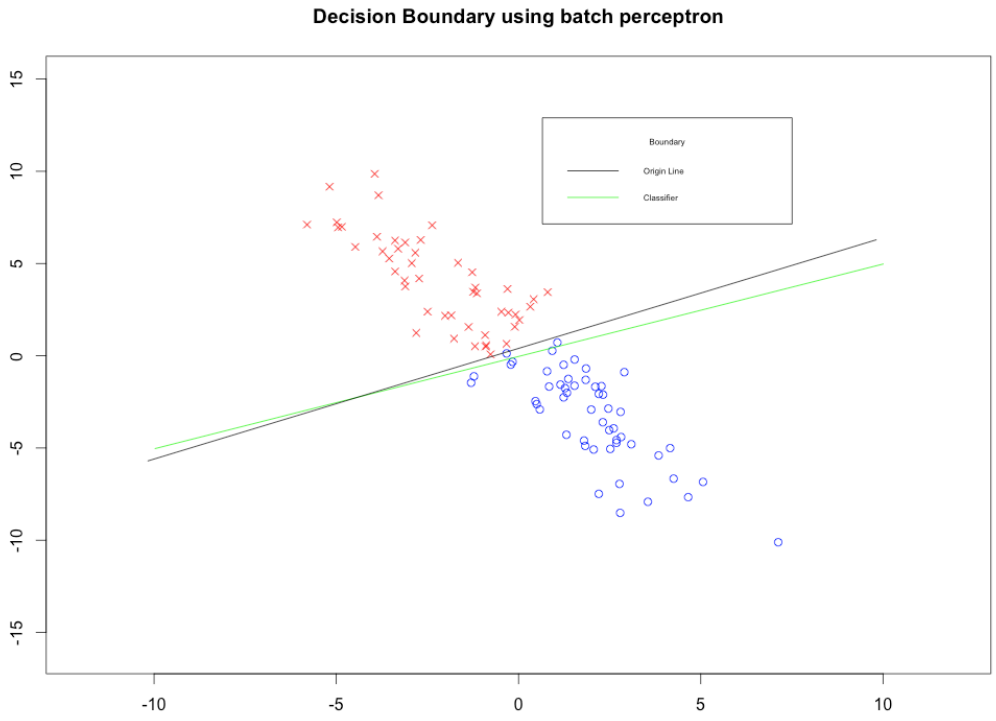
(2)

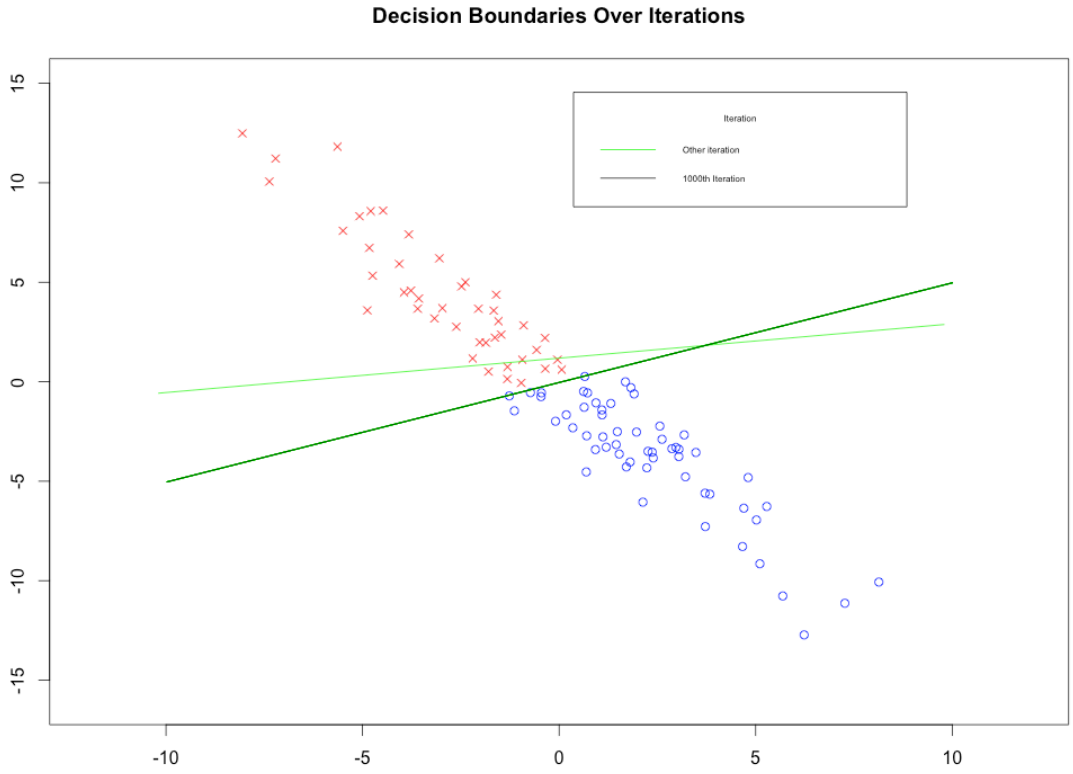
```
# Input: sample data set S, class label vector y
# Output: normal vector Z, the history of the normal vector
throught the training run Z_history
perceptrain <- function(S, y){
    dimension <- dim(S)[2] # The dimension of <x, 1>
    n <- dim(S)[1]  # number of observations
    Z <- runif(dimension, 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)

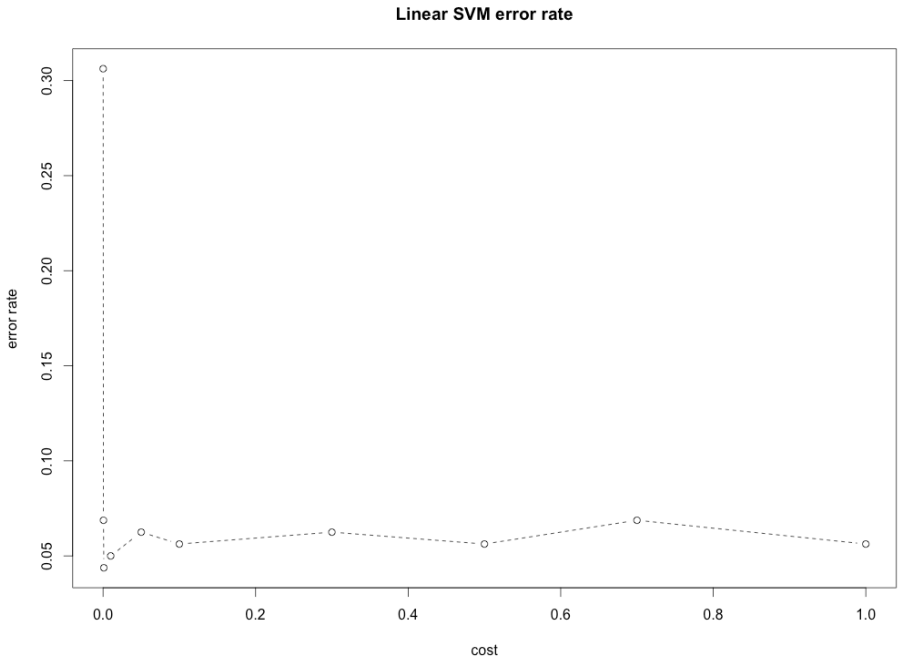


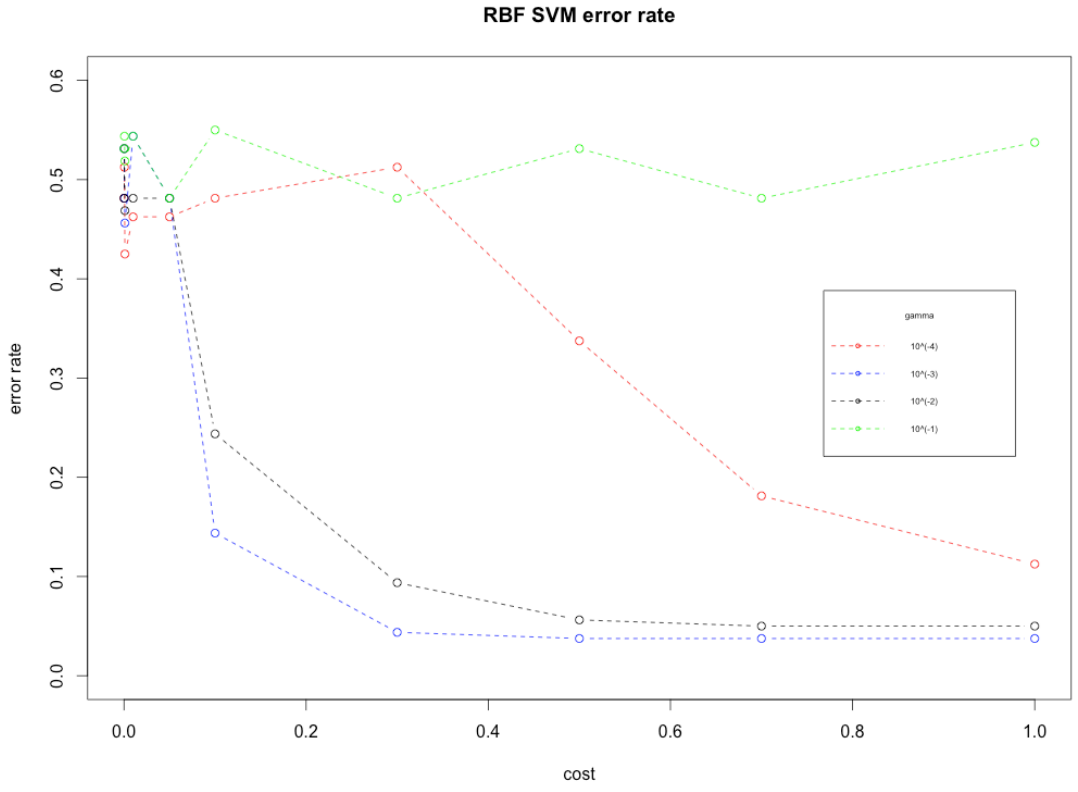



```
# Convert data to 2D corresponding representation
test.pos <- S.test$y == 1
test.data_pos <- S.test$S[test.pos, 1:2]
test.data_neg <- S.test$S[!test.pos, 1:2]
# Convert the 3D vectors into corresponding line
vector_to_line <- function (vector){
    Null_space <- Null(vector[1:2])
    offset <- (-vector[3]) * 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).


Linear SVM error rate
error rate
0.05
0.10
0.15
0.20
0.25
0.30
0.0
0.2
0.4
0.6
0.8
1.0
cost


RBF SVM error rate
error rate
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.0
0.2
0.4
0.6
0.8
1.0
cost
gamma
10^(-4)
10^(-3)
10^(-2)
10^(-1)

(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)
```