CSDS391 - Programming Project 2

caseid: mxd601

Muhammed Demirak

29 November, 2022

Contents

```
# loading required data and libraries
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(plotly)
```

```
## Loading required package: ggplot2

##
## Attaching package: 'plotly'

## The following object is masked from 'package:ggplot2':
##
##     last_plot

## The following object is masked from 'package:stats':
##
##     filter

## The following object is masked from 'package:graphics':
##
##     layout
```

```
library(randomcoloR)
library(neuralnet)
```

```
##
## Attaching package: 'neuralnet'

## The following object is masked from 'package:dplyr':
##
##     compute
```

```
data(iris) # irisdata.csv built into r

iris_unsupervised <- iris[ , -5] # remove species
```

```r
ir_c <- iris[ which(iris$Species != "setosa"), ]
rownames(ir_c) = seq(length = nrow(ir_c))        # reset rows

ir_c_unsupervised <- ir_c[ , -5] # remove species

set.seed(69) # ensure reproducability of results
```

1 Clustering (35P total)

1.1 a 15P

Write a program that implements the k-means clustering algorithm on the iris data set. You should use the objective function and learning rule you derived in W4 Q1 (and your implementation will help you uncover any errors, so it is helpful to do this before you turn in W4.)

```r
empty_cluster <- function() {
  df <- data.frame(Sepal.Length = numeric(),
                   Sepal.Width = numeric(),
                   Petal.Length = numeric(),
                   Petal.Width = numeric())
  return(df)
}

initial_clusters <- function(K) {
  clusters <- list()
  for (k in 1:K) {
    clusters[[k]] <- empty_cluster()
  }
  return(clusters)
}

initial_means <- function(K, dataframe) {
  means <- initial_clusters(K)
  random_initial <- dataframe[sample(nrow(dataframe), 3), ]
  for (k in 1:K) {
    means[[k]][1, ] <- random_initial[k, ]
  }
  return(means)
}

get_diffs <- function(x_n, means, K) {
  diffs <- initial_clusters(K)
    for (k in 1:K) {
      mean_k <- means[[k]][1, ]
      diff <- (abs(x_n - mean_k))^2
      diffs[[k]][1, ] <- diff
    }
  return(diffs)
}

get_min_se <- function(diffs) {
  K <- length(diffs)
  ses <- empty_cluster()

  for (k in 1:K) {
    ses[nrow(ses) + 1, ] <- diffs[[k]][1, ]
  }

  min <- (ses %>% arrange(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width))[1, ]
  return(min)
}

get_cluster_mean <- function(cluster) {
  mean <- empty_cluster()
  mean[1, 1] <- mean(cluster[ , 1])
  mean[1, 2] <- mean(cluster[ , 2])
  mean[1, 3] <- mean(cluster[ , 3])
  mean[1, 4] <- mean(cluster[ , 4])
  return(mean)
}

get_clusters <- function(K, dataframe, means) {
  clusters <- initial_clusters(K)
  all_mins <- empty_cluster()
  clusters_vector <- c()
  N <- nrow(dataframe)
  for (n in 1:N) {
    x_n <- dataframe[n, ]
    diffs <- get_diffs(x_n, means, K)
    min_se <- get_min_se(diffs)
    all_mins[n, ] <- min_se

    for (k in 1:K) {
      se <- diffs[[k]][1, ]
      if (all_equal(se, min_se) == TRUE) {
        clusters[[k]] <- rbind(clusters[[k]], x_n)
        clusters_vector <- c(clusters_vector, k)
        break
      }
    }
  }

  clusters_type <- list(clusters=clusters, mins = all_mins, vec = clusters_vector)

  return(clusters_type)
}

get_min_diffs <- function(K, dataframe, means) {

  N <- nrow(dataframe)
  for (n in 1:N) {
    x_n <- dataframe[n, ]
    diffs <- get_diffs(x_n, means, K)
    min_se <- get_min_se(diffs)
  }
  return(all_mins)
}

update_means <- function(K, clusters, means) {
  new_means <- initial_clusters(K)
  for (k in 1:K) {
    new_means[[k]][1, ] <- get_cluster_mean(clusters$clusters[[k]])
  }
  return(new_means)
}

empty_sum <- function() {
  sum <- empty_cluster()
  sum[1, ] <- c(0.00, 0.00, 0.00, 0.00)
  return(sum)
}

sum_min_diffs <- function(all_mins) {
  sum <- empty_sum()
  N <- nrow(all_mins)
  for (n in 1:N) {
    old <- sum[1, ]
    diff <- all_mins[n, ]
    sum[1, ] <- old + diff
  }

  return(sum)
}

not_zero <- function(diff_sum) {
  op <- diff_sum[ , 1] > 0.00 &
        diff_sum[ , 2] > 0.00 &
        diff_sum[ , 3] > 0.00 &
        diff_sum[ , 4] > 0.00
  return(op)
}

k_means <- function(K, dataframe, cls) {

  means <- initial_means(K, dataframe)
  clusters <- get_clusters(K, dataframe, means)
  all_mins <- clusters$mins
  prev_sum <- empty_sum()
  curr_sum <- sum_min_diffs(all_mins)

  iter <- 0

  diff_sum_hist <- empty_cluster()
  means_hist <- c(list(means))
  clusters_hist <- c(list(clusters))
  diff_sum <- abs(curr_sum[1, ] - prev_sum[1, ])
  while (not_zero(diff_sum) | iter < 10) {
    # update
    means <- update_means(K, clusters, means)
    clusters <- get_clusters(K, dataframe, means)

    means_hist <- c(means_hist, list(means))
    clusters_hist <- c(clusters_hist, list(clusters))
    all_mins <- clusters$mins

    # increment
    prev_sum[1, ] <- curr_sum[1, ]
    curr_sum[1, ] <- sum_min_diffs(all_mins)
    diff_sum <- abs(curr_sum[1, ] - prev_sum[1, ])
    diff_sum_hist <- rbind(diff_sum_hist, diff_sum)
    iter <- iter + 1
  }

  max_iter <- nrow(diff_sum_hist)
  itr_vec <- c()
  diff_hist <- c()
  for(of_iteration in 1:max_iter) {
    itr_vec <- c(itr_vec, of_iteration)
    diff_sum_sum <- rowSums(diff_sum_hist[of_iteration, ])
    diff_hist <- c(diff_hist, abs(diff_sum_sum))
  }

  petal_l <- dataframe$Petal.Length
  petal_w <- dataframe$Petal.Width
  y <- as.factor(cls)

  kmeans <- list("petal_l"=petal_l,
                 "petal_w"=petal_w,
                 "y"=y,
                 "classes"=clusters,
                 "centroids"=means,
                 "mins"=all_mins,
                 "iteration"=itr_vec,
                 "objective_func"=curr_sum,
                 "diff_hist" = diff_hist,
                 "clusts_hist" = clusters_hist,
                 "centroids_hist" = means_hist)

  return(kmeans)
}
```

```r
k_means_result_2 <- k_means(2, ir_c_unsupervised, ir_c$Species)
k_means_result <- k_means(3, iris_unsupervised, iris$Species)
```

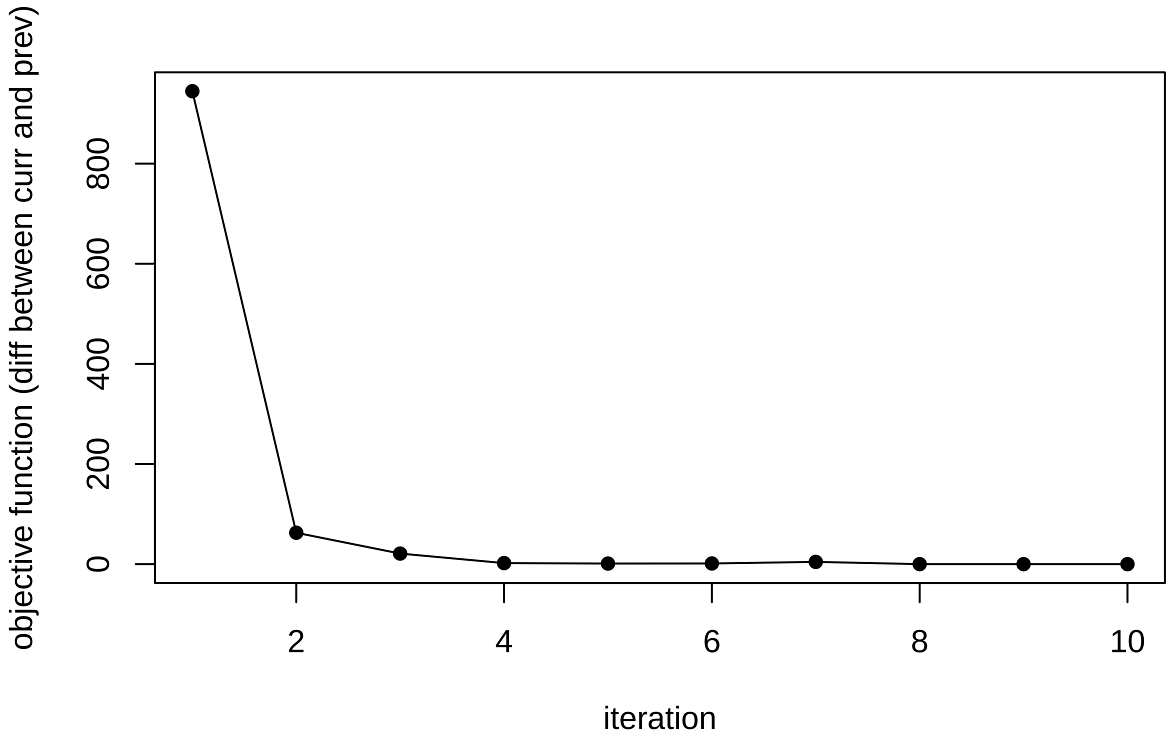
1.2 b 5P

Plot the value of the objective function as a function of the iteration

$$D = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{n,k} ||x_n - \mu_k||^2$$

to show that your learning rule and implementation minimizes this expression.

```r
plot(k_means_result$iteration, k_means_result$diff_hist,
     type = "l",
     xlab = "iteration",
     ylab = "objective function (diff between curr and prev)")
points(k_means_result$iteration, k_means_result$diff_hist,
       pch = 16)
```



1.3 c 5P

Plot the results of the learning process by showing the initial, intermediate, and converged cluster centers overlaid on the data for k = 2 and k = 3.

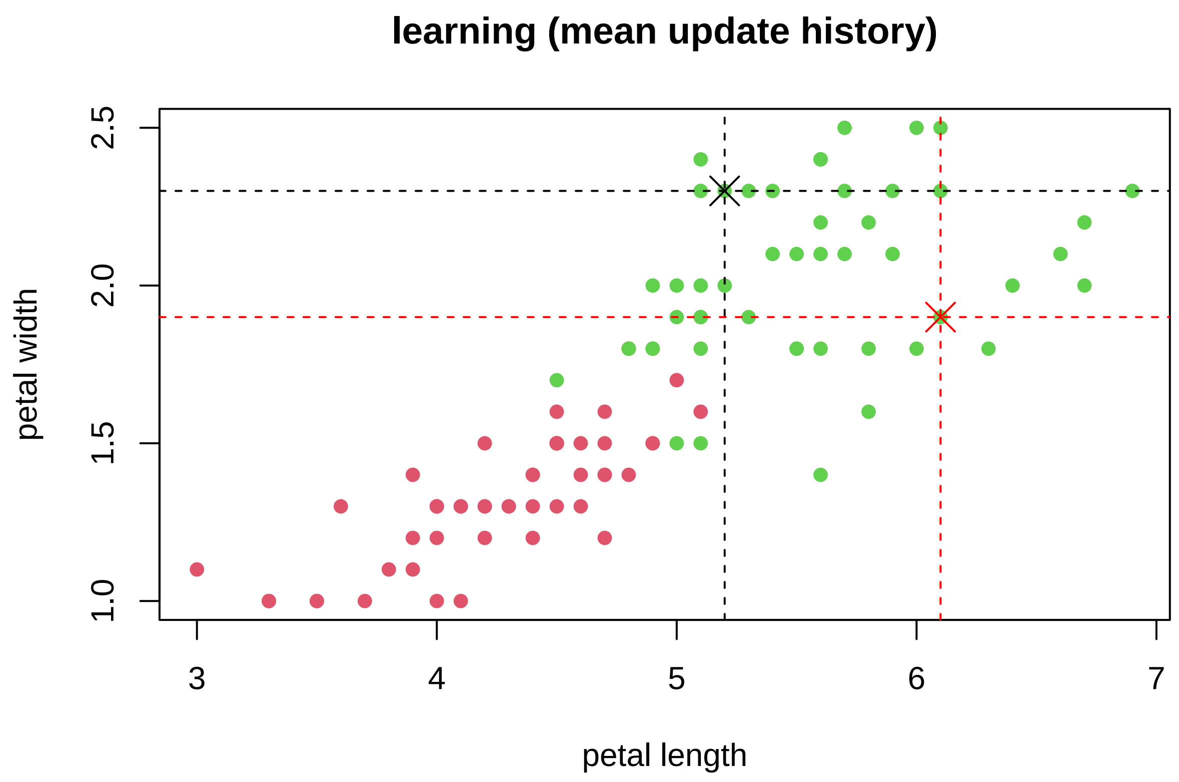
The black point is k = 1

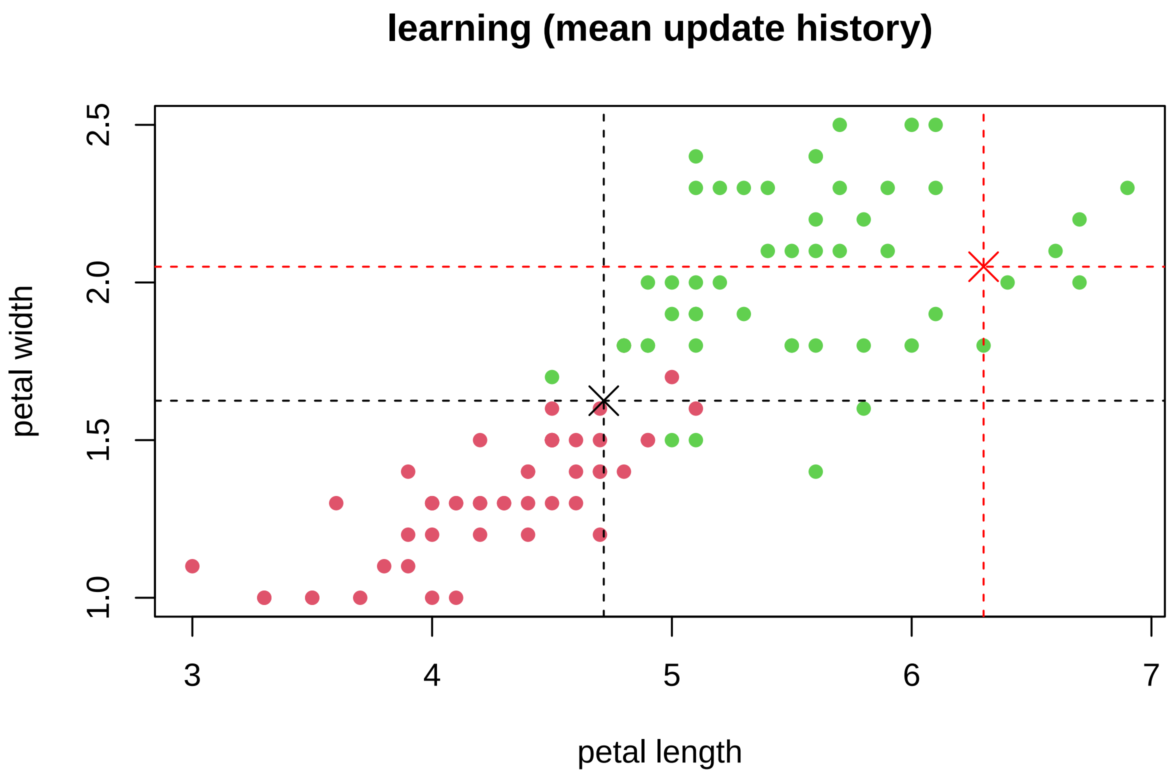
The red point is k = 2

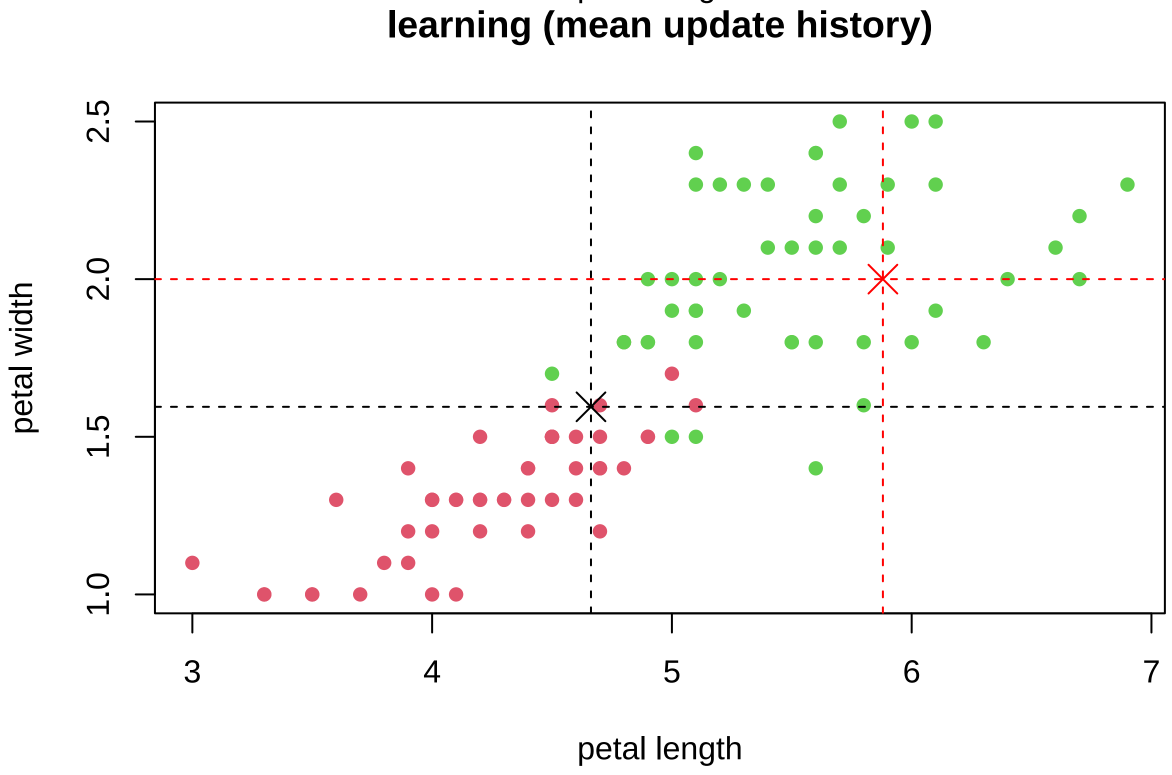
The green point is k = 3

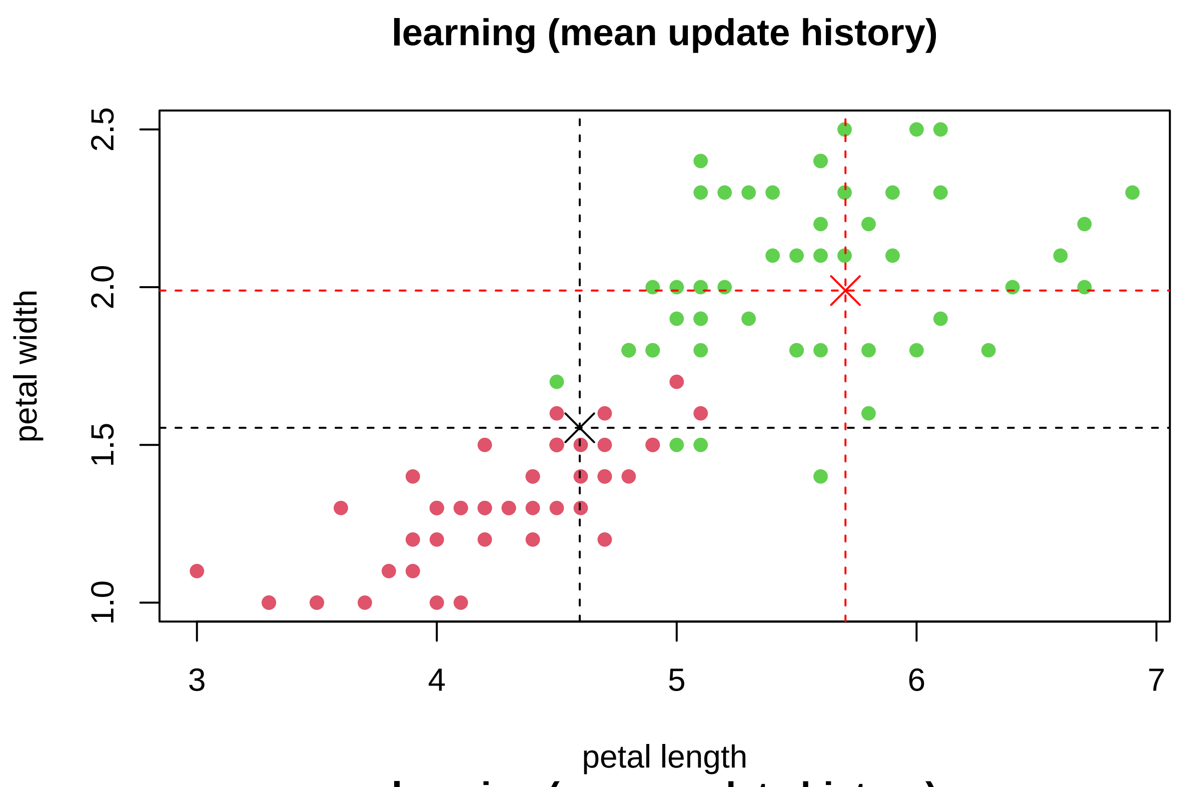
```r
plot_means <- function(iter, k_means_set) {
  x1 <- k_means_set$petal_l
  x2 <- k_means_set$petal_w
  y <- as.factor(k_means_set$y)
  K <- length(k_means_set$centroids)
  plot(x1, x2, col = y, pch = 16,
       xlab = "petal length", ylab = "petal width",
       main = "learning (mean update history)")

  colors_for_centroids <- c("black", "red", "green")
    for (k in 1:K) {
      desired_col <- colors_for_centroids[k]
      points(k_means_set$centroids_hist[[iter]][[k]]$Petal.Length,
             k_means_set$centroids_hist[[iter]][[k]]$Petal.Width,
             col = desired_col, pch = 4, cex = 2)
      abline(v = k_means_set$centroids_hist[[iter]][[k]]$Petal.Length,
             col = desired_col, lty = 2)
      abline(h = k_means_set$centroids_hist[[iter]][[k]]$Petal.Width,
             col = desired_col, lty = 2)
    }
}


plot_learning <- function(k_means_set) {
  iters <- length(k_means_result$centroids_hist)
  for (i in 1:iters) {
    plot_means(i, k_means_set)
  }
}
```

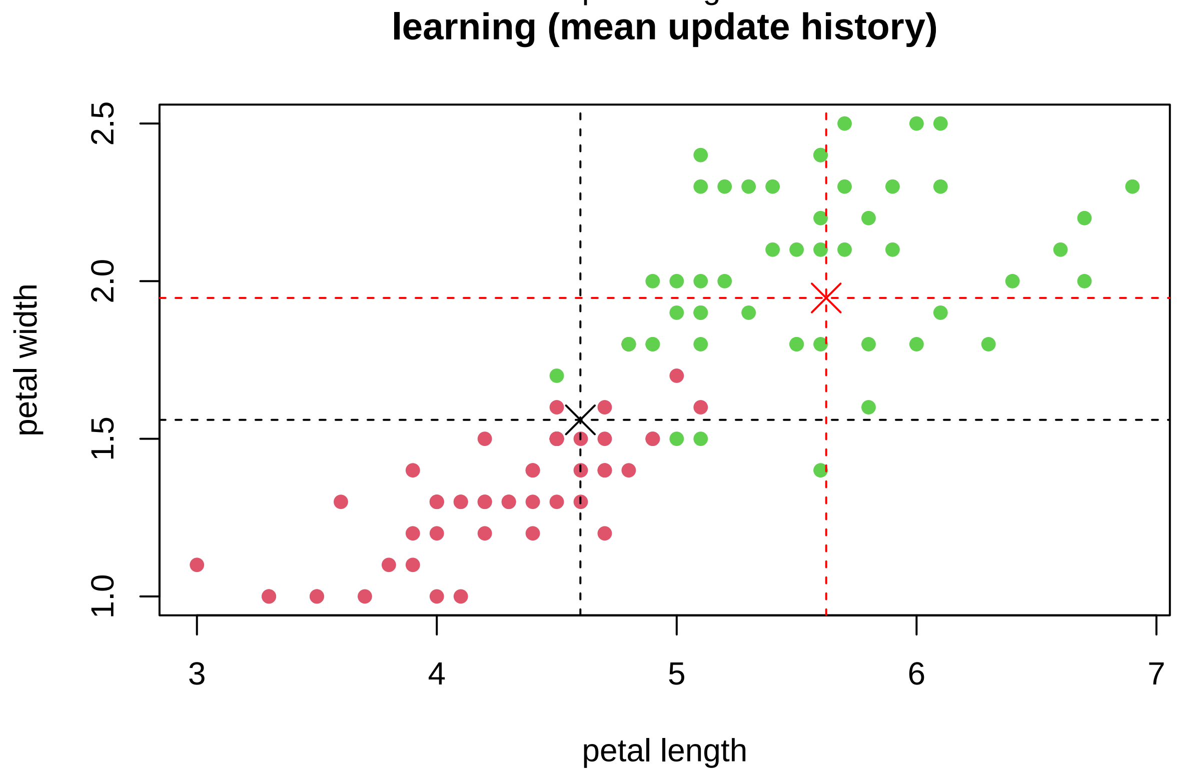
```
# k = 2
plot_learning(k_means_result_2)
```

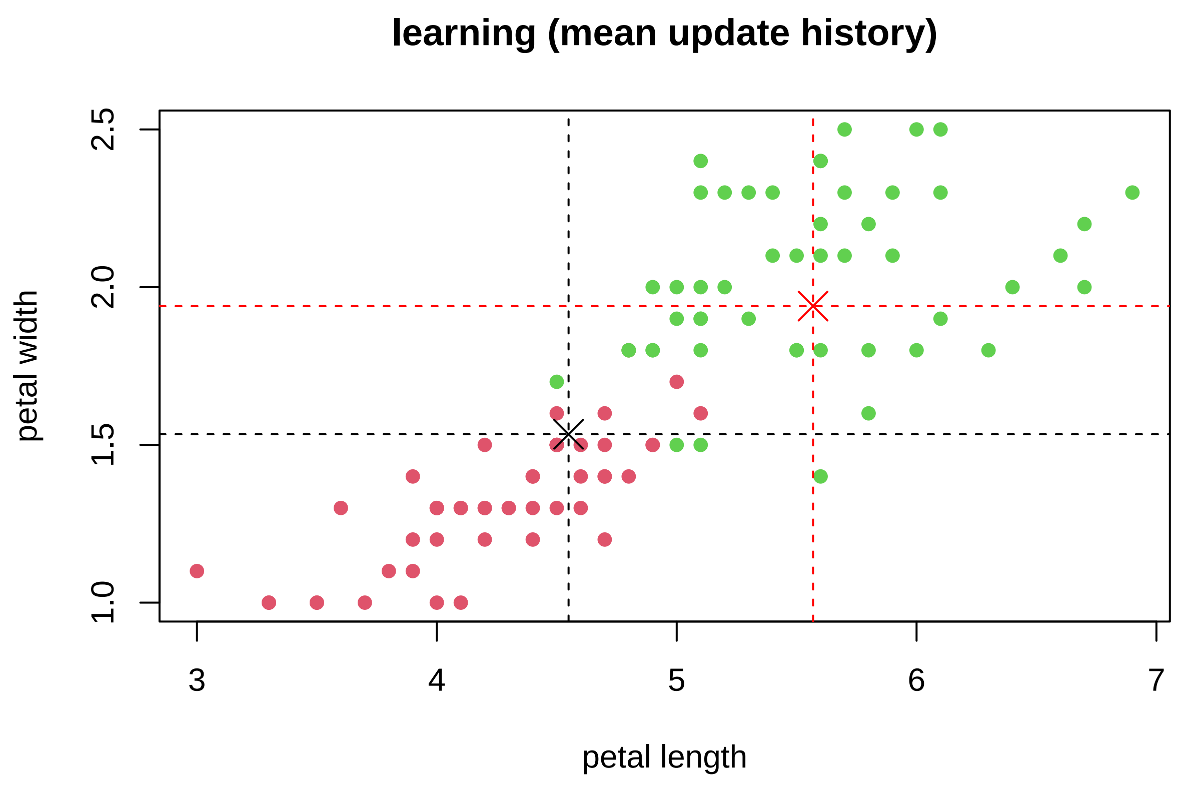



learning (mean update history)
petal width
petal length
1.0
1.5
2.0
2.5
3
4
5
6
7

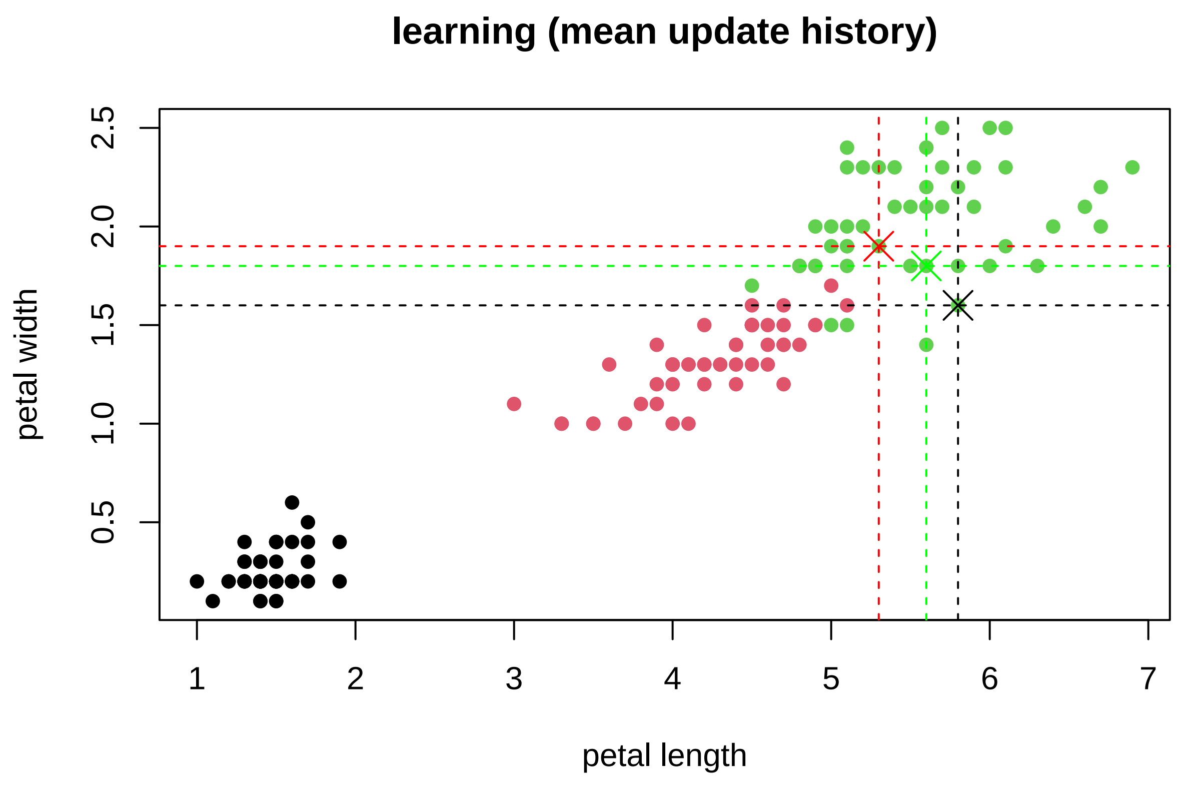

learning (mean update history)
petal width
petal length
1.0
1.5
2.0
2.5
3
4
5
6
7

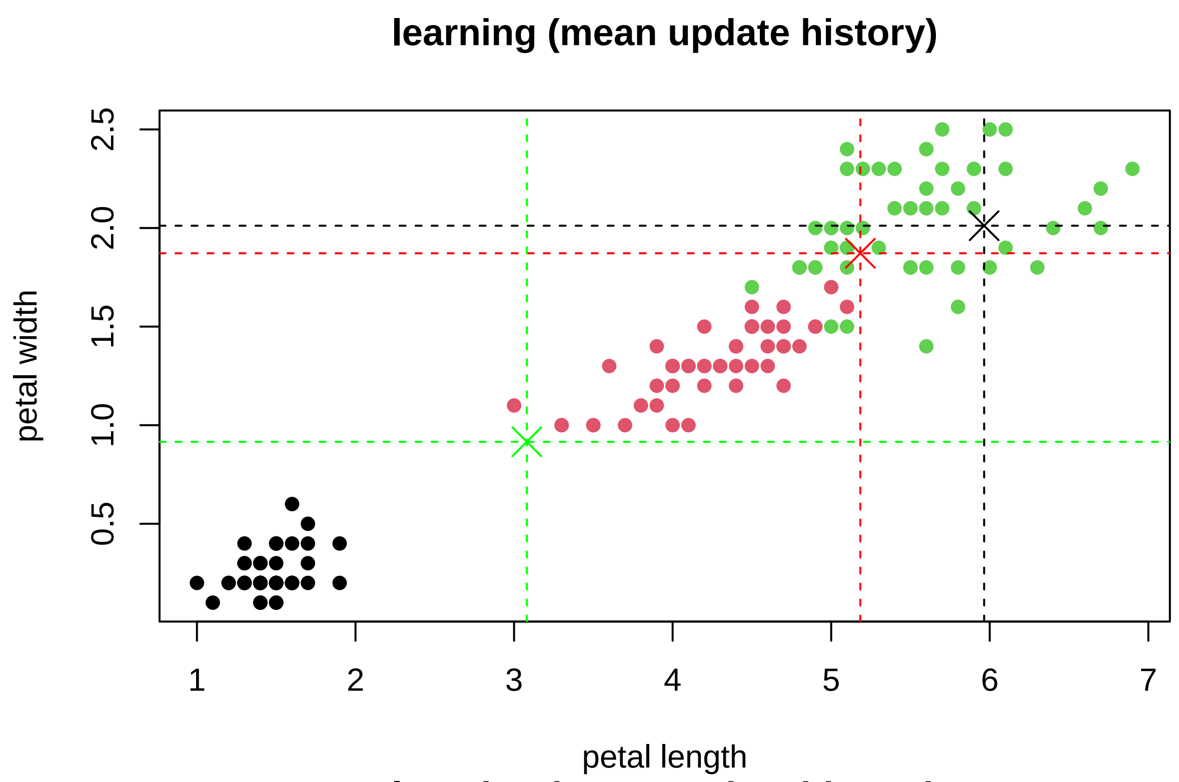

learning (mean update history)
petal width
petal length
1.0
1.5
2.0
2.5
3
4
5
6
7

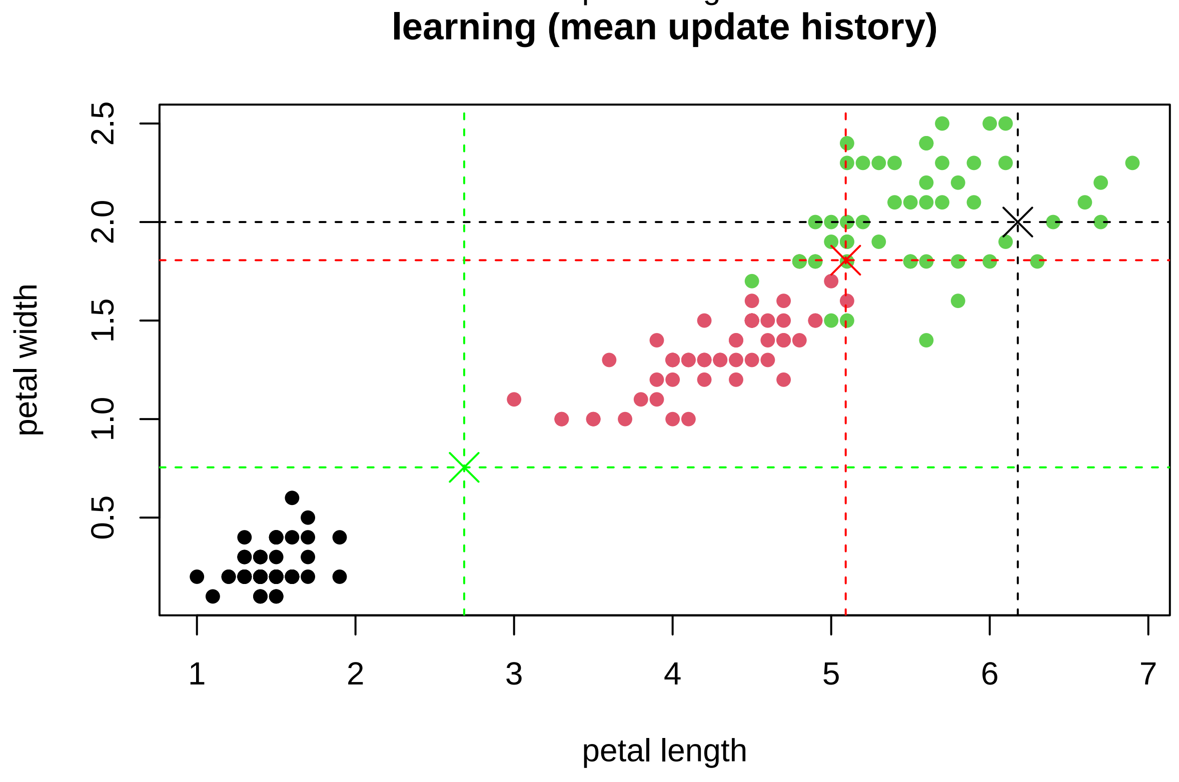

learning (mean update history)
petal width
petal length
1.0
1.5
2.0
2.5
3
4
5
6
7

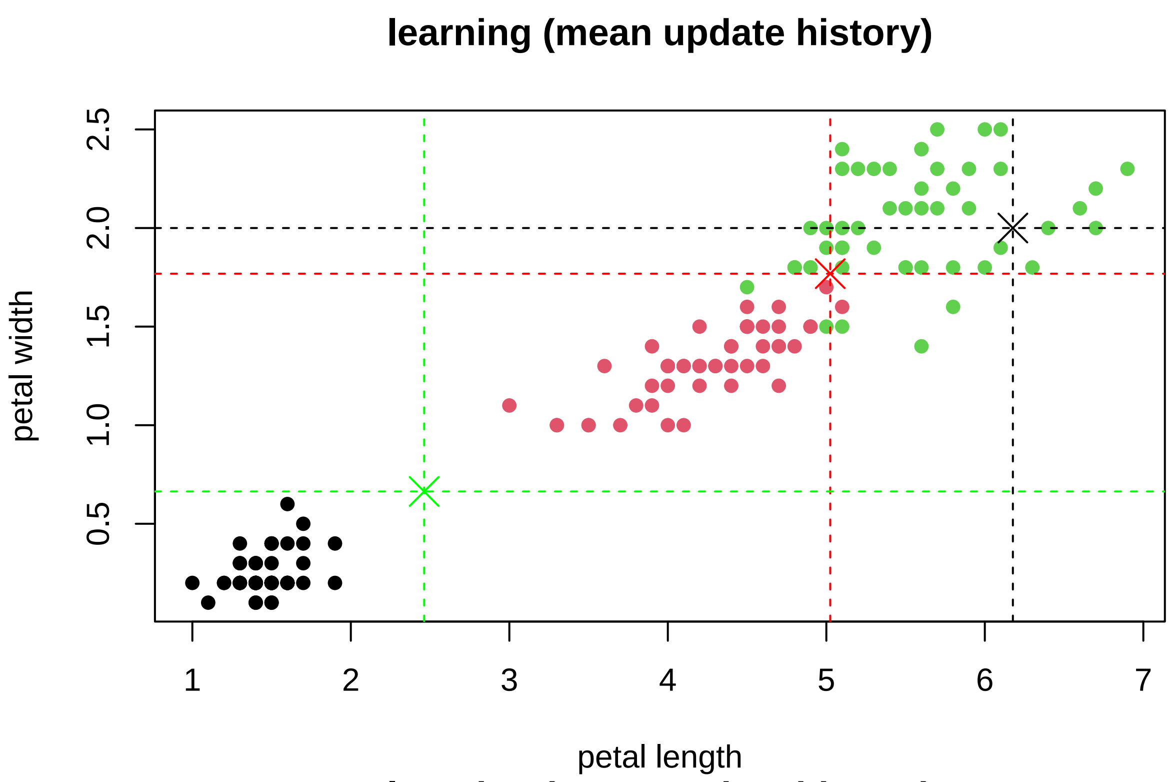


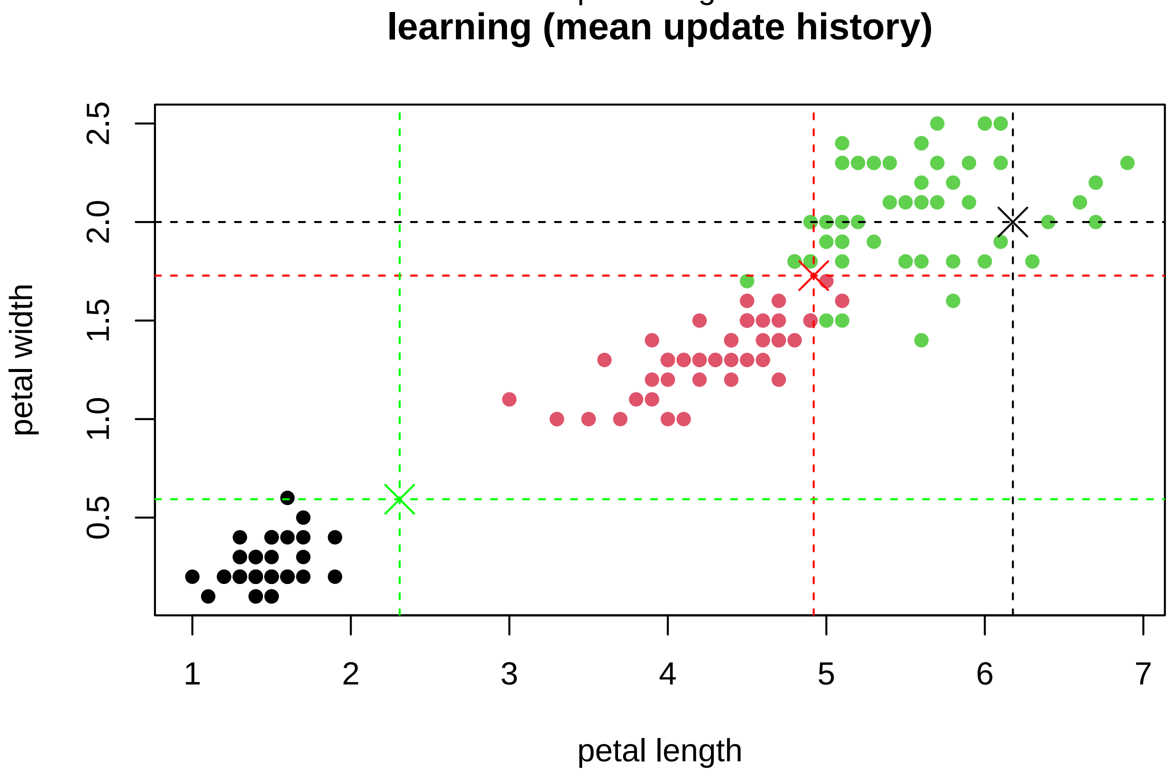
```
# k = 3
plot_learning(k_means_result)
```

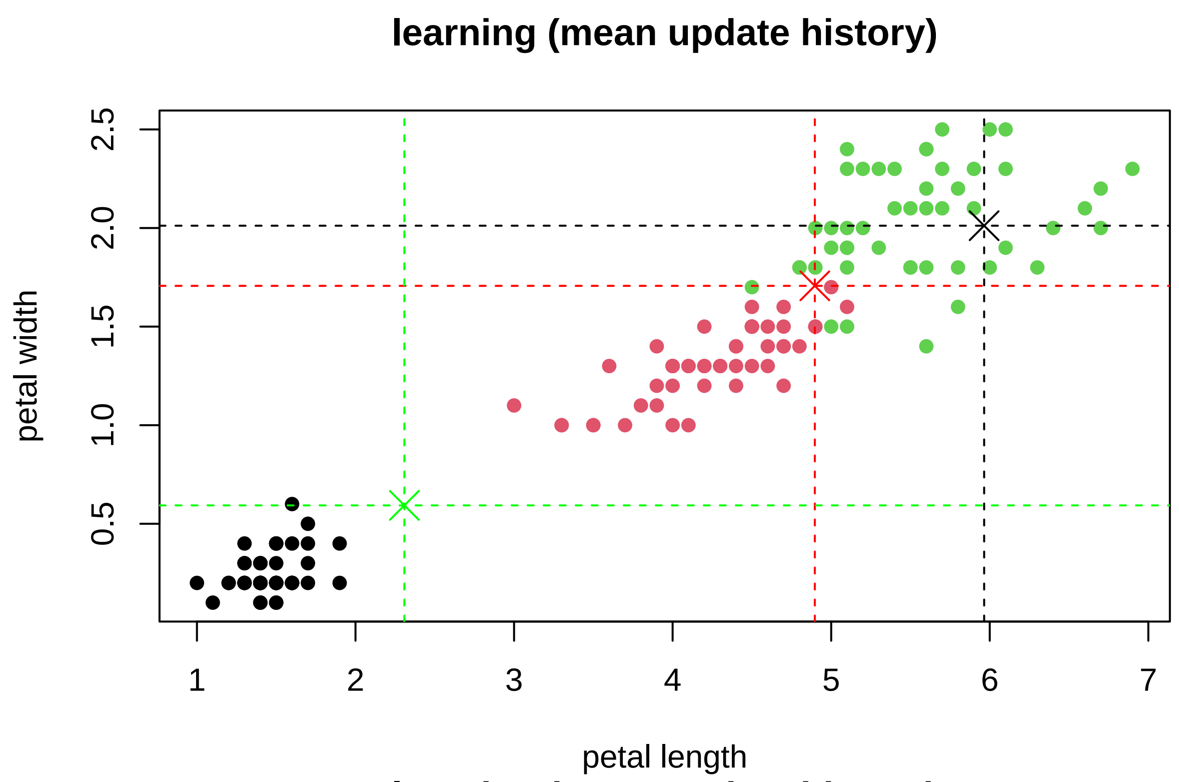


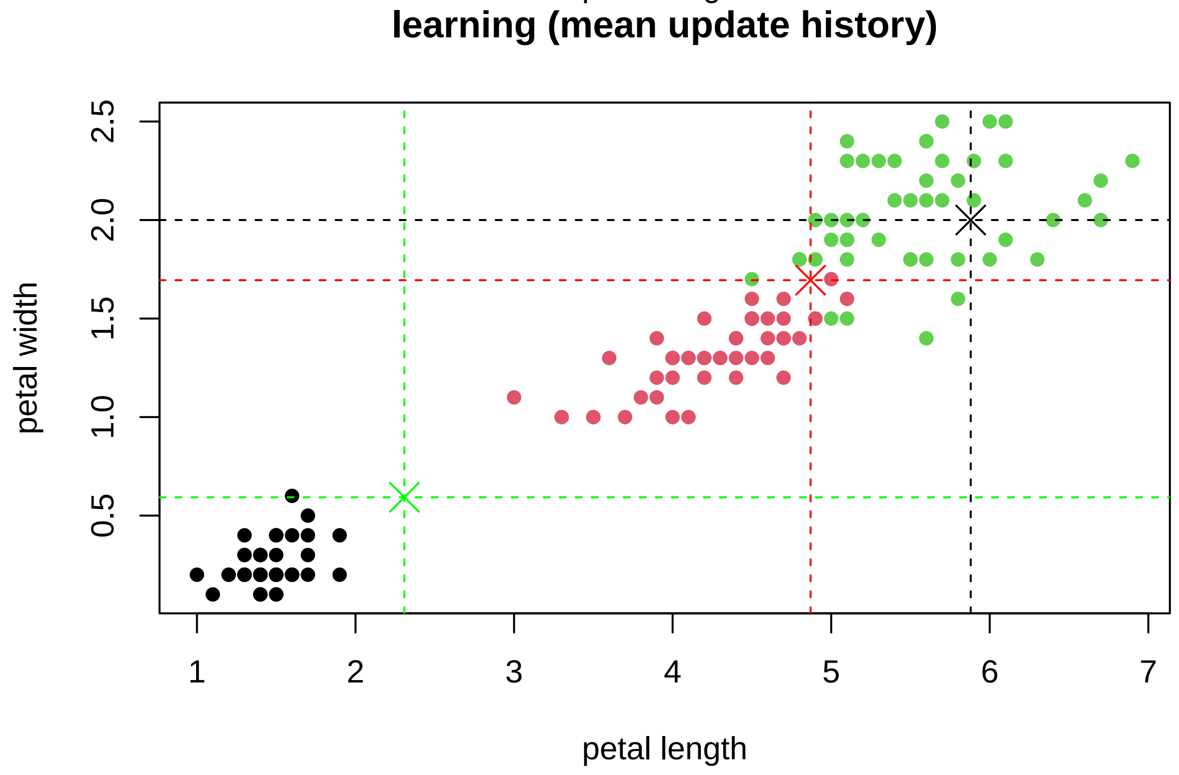

learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7

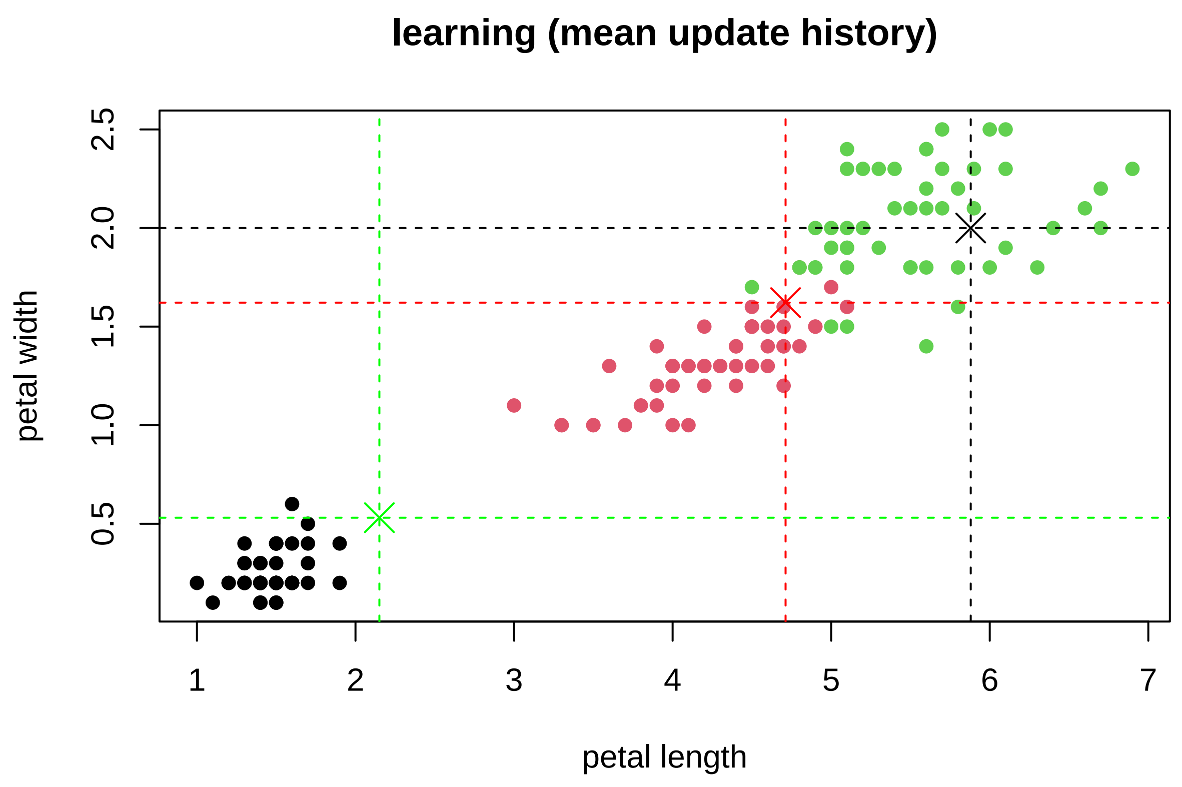

learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


learning (mean update history)
petal width
petal length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7



The classifications change depending on randomization of initial means, as expected.

1.4 d 10P

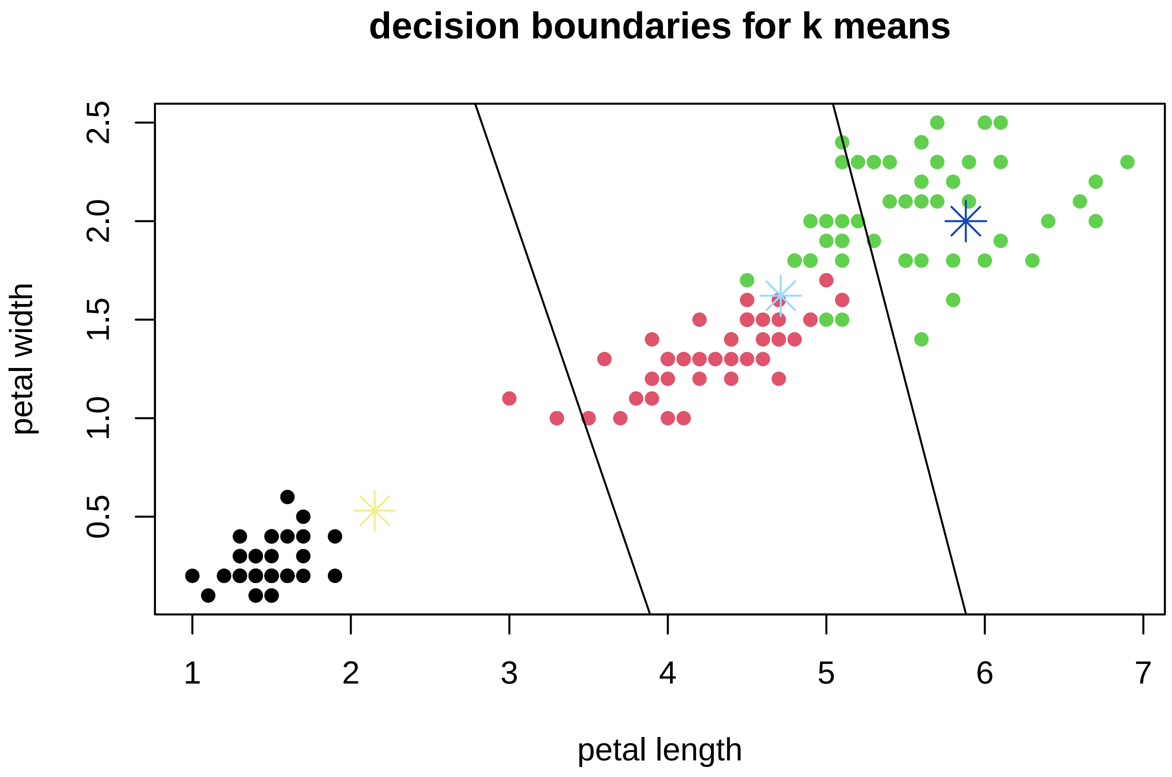
Devise a method to plot the decision boundaries for this dataset using the optimized parameters. Explain your approach and plot your results. (optimized parameters = means)

Approach:

calculate y = mx + b for midpoints between means as decision boundary

```
order_means <- function(means) {
  new_means <- empty_cluster()
  for (k in 1:length(means)) {
    new_means[k, ] <- means[[k]][1, ]
  }
  new_means <- new_means %>% arrange(Petal.Length, Petal.Width)

  return(new_means)
}

plot_decision_boundaries <- function(means) {
  plot(iris$Petal.Length, iris$Petal.Width,
       col = iris$Species, pch = 16,
       xlab = "petal length", ylab = "petal width",
       main = "decision boundaries for k means")
  ordered_means <- order_means(means)
  K <- nrow(ordered_means)
  for (k in 2:K) {
    m1 <- ordered_means[k-1, ]
```

```r
    m2 <- ordered_means[k, ]
    x1 <- m1$Petal.Length
    y1 <- m1$Petal.Width
    if (k == 2) {
      points(m1$Petal.Length, m1$Petal.Width,
             pch = 8, cex = 2, col = randomColor())
    }
    points(m2$Petal.Length, m2$Petal.Width,
           pch = 8, cex = 2, col = randomColor())

    x2 <- m2$Petal.Length
    y2 <- m2$Petal.Width

    reciprocal_slope <- -((x2-x1) / (y2-y1))
    intercept <- ((y1 - (reciprocal_slope * x1)
                  ) + (y2 - (reciprocal_slope * x2)
                       )
                 ) / 2
    abline(intercept, reciprocal_slope)
  }
}

plot_decision_boundaries(k_means_result$centroids)
```



2 Linear Decision Boundaries (30P total)

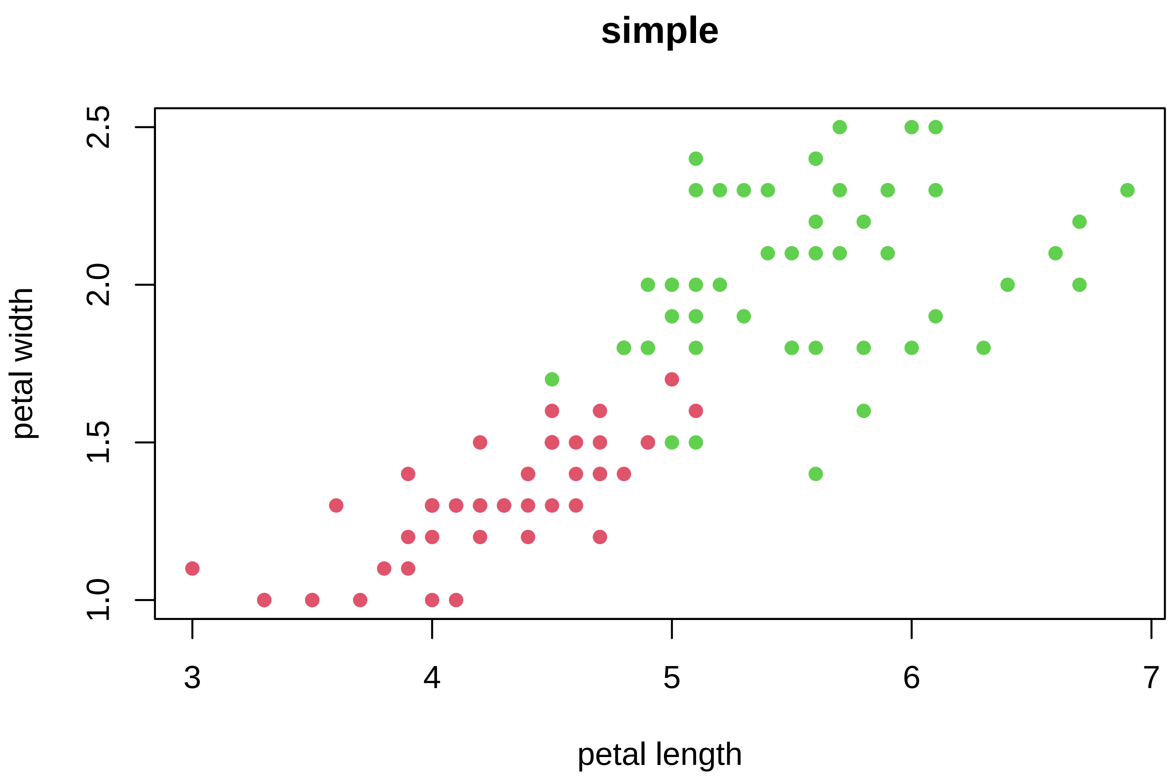
Single layer neural network

2.1 a 10P

Inspect the iris dataset which is provided with the assignment file on Canvas. Write a program (or use a library) that loads the iris data set and plots the 2nd and 3rd iris classes, similar to the plots shown in lecture on neural networks (i.e. using the petal width and length dimensions).

Simple plotting of versicolor (2nd class) and virginica (3rd class) with Petal Width ~ Petal Length for analysis:

```r
plot(ir_c$Petal.Length, ir_c$Petal.Width, col = ir_c$Species,
     pch = 16,
     xlab = "petal length", ylab = "petal width", main = "simple")
```



2.2 b 5P

```r
# setup dataset
# 0, 1 for species as binary classifier
determine_class <- function(species) {
  cols <- c()
  for (s in species) {
    if (s == "versicolor") {
      cols <- c(cols, 0)
    }
    else {
      cols <- c(cols, 1)
    }
  }
  return(cols)
}

df <- mutate(ir_c, y = determine_class(Species))
df <- df [ , -5]


## 3:4 => PetalLength, PetalWidth
# x <- scale(df[ , c(3:4)]) # -> with standardized data
x <- df[ , c(3:4)]          # -> without standardized data

y <- df$y
dim(y) <- c(length(y), 1)

x <- as.matrix(x, byrow = TRUE)
x <- t(x)

y <- as.matrix(y, byrow = TRUE)
y <- t(y)
```

Define a function that computes the output of simple one-layer neural network using a sigmoid non-linearity (linear classification with sigmoid non-linearity).

```r
# add bias to x
x_nn <- function(x) {
  m <- dim(x)[2]
  x_0 <- matrix(rep(1, m), nrow = 1)
  x <- rbind(x_0, x)
  return(x)
}

new_weight <- function(w1, w2, bias = NULL) {
  w0 <- 0
  if (!is.null(bias)) {
    w0 <- bias
  }
  weight <- c(w0, w1, w2)
  return(t(weight))
}

random_weight <- function(n_x, n_y, bias = NULL) {
  w0 <- 0
  if (!is.null(bias)) {
    w0 <- bias
  }
  weight <- matrix(runif(n_y * n_x), nrow = n_y, ncol = n_x, byrow = TRUE)
  weight <- cbind(matrix(c(w0)), weight)
  return(weight)
}

random_weight(2, 1)
```

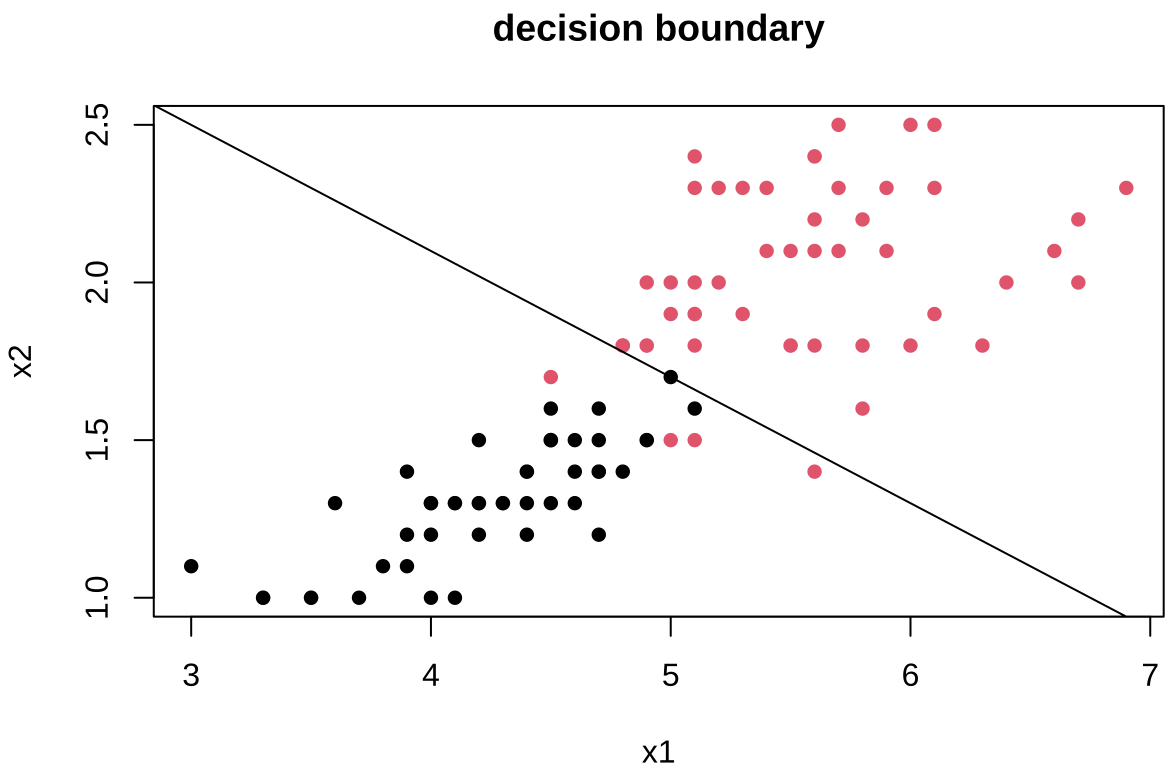
```
##      [,1]      [,2]      [,3]
## [1,]    0 0.1710843 0.7876906
```

```
initial_weight <- function(x, y, weight = NULL, bias = NULL) {
  n_x <- (dim(x)[1] - 1)
  n_y <- dim(y)[1]
  W <- random_weight(n_x, n_y, bias)
  if (!is.null(weight)) {
    W <- weight
  }
  return(W)
}

sigmoid <- function(x) {
  return(1 / (1 + exp(-x)))
}

fwd_prop <- function(x, y, W) {
  n_x <- (dim(x)[1] - 1)
  n_y <- dim(y)[1]

  Z <- W %*% x
  A <- sigmoid(Z)

  fwd_pass <- list("Z" = Z,
                   "A" = A)
  return(fwd_pass)
}

prediction <- function(A) {
  return(round(A))
}

non_linearity <- function(x, y, weight = NULL, bias = NULL) {
  X <- x_nn(x)
  W <- initial_weight(X, y, weight, bias)
  fwd_pass <- fwd_prop(X, y, W)
  predict <- prediction(fwd_pass$A)

  model <- list("x0" = X[1, ],
                "x1" = X[2, ],
                "x2" = X[3, ],
                "y" = y,
                "W" = W,
                "fwd_feed" = fwd_pass,
                "pred" = predict)
  return(model)
}
```

2.3 c 5P

Write a function that plots the decision boundary for the non-linearity above overlaid on the iris data. Choose a boundary (by setting weight parameters by hand) that roughly separates the two classes. Use an output of 0 to indicate the 2nd iris class, and 1 to indicate the 3rd.

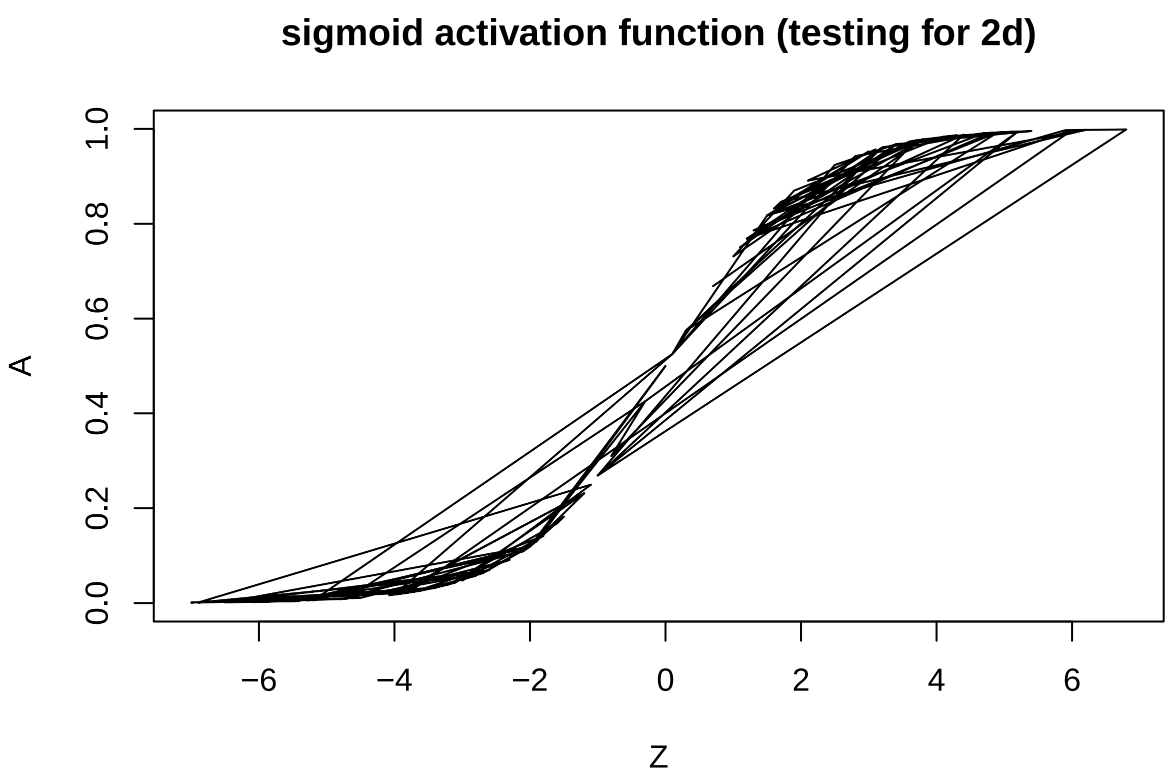
```r
W1 <- 2
W2 <- 5
BIAS <- -18.5

get_slope_intercept <- function(W) {
  w_1 <- W[1, 2]
  w_2 <- W[1, 3]
  w_0 <- W[1, 1]

  m <- -(w_1 / w_2)
  b <- -(w_0 / w_2)

  db_comp <- list("m" = m,
                  "b" = b)

  return(db_comp)
}

plot_nl_db <- function(x, y, w1, w2, w0) {
  W <- new_weight(w1, w2, w0)
  si <- get_slope_intercept(W)
  nl_cw_model <- non_linearity(x, y, W, w0)
  nl_cw_model$y <- as.factor(nl_cw_model$y)
  plot(nl_cw_model$x1, nl_cw_model$x2,
       col = nl_cw_model$y, pch = 16,
       xlab = "x1", ylab = "x2", main = "decision boundary")
  m <- si$m
  b <- si$b
  abline(a = b, b = m)
}

plot_nl_db(x, y, W1, W2, BIAS)
```



2.4 d 5P

Use a surface plot from a 3D plotting libary (e.g. in matlab or using matplotlib in python) to plot the output of your neural network over the input space. This should be similar to learning curve shown in fig 19.17 (18.17 in 3rd ed.) of the textbook.

```r
weight <- new_weight(W1, W2, BIAS)
nl_cw_model <- non_linearity(x, y, weight)
x1 <- nl_cw_model$x1
x2 <- nl_cw_model$x2
summation_output <- nl_cw_model$fwd_feed$Z[1, ]
sigmoid_output <- nl_cw_model$fwd_feed$A[1, ]
plot(summation_output, sigmoid_output, type = "l",
     xlab = "Z", ylab = "A",
     main = "sigmoid activation function (testing for 2d)")
```



```
# dynamic plotting function, have to screenshot
# plot_ly(x = ~x1, y = ~x2, z = ~sigmoid_output, type = 'mesh3d') %>% add_lines()
```

2.5 e 5P

Show the output of your simple classifier using examples from the 2nd and 3rd Iris classes. Choose examples that are unambigious as well as those that are near the decision boundary.

Desired examples:

0:

- closer to db: 3, 23, 28, 37
- farther from db: 49, 32

1:

- closer to db: 70, 89, 84, 77
- farther from db: 51, 73

```
desired_sets <- c(3, 23, 28, 32, 37, 49, 51, 70, 73, 77, 84, 89)
examples_x <- x[ , desired_sets]
examples_y <- t(y[ , desired_sets])
custom_weight <- new_weight(W1, W2, BIAS)
nl_eg_model <- non_linearity(examples_x, examples_y, custom_weight, BIAS)

# display the output of the non-linearity function
nl_eg_model
```

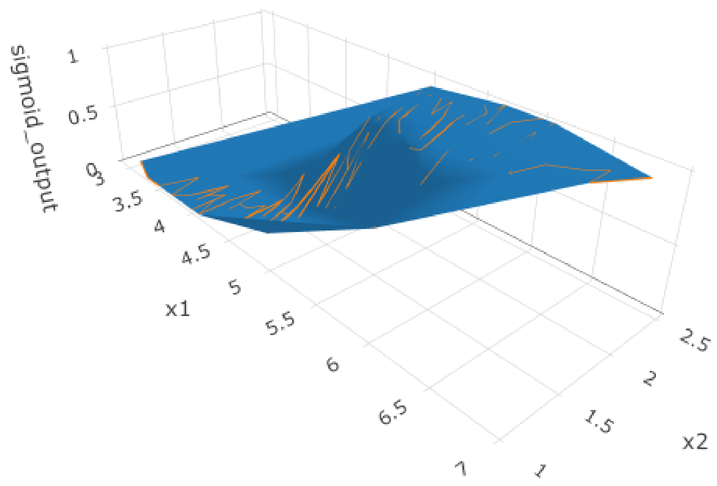
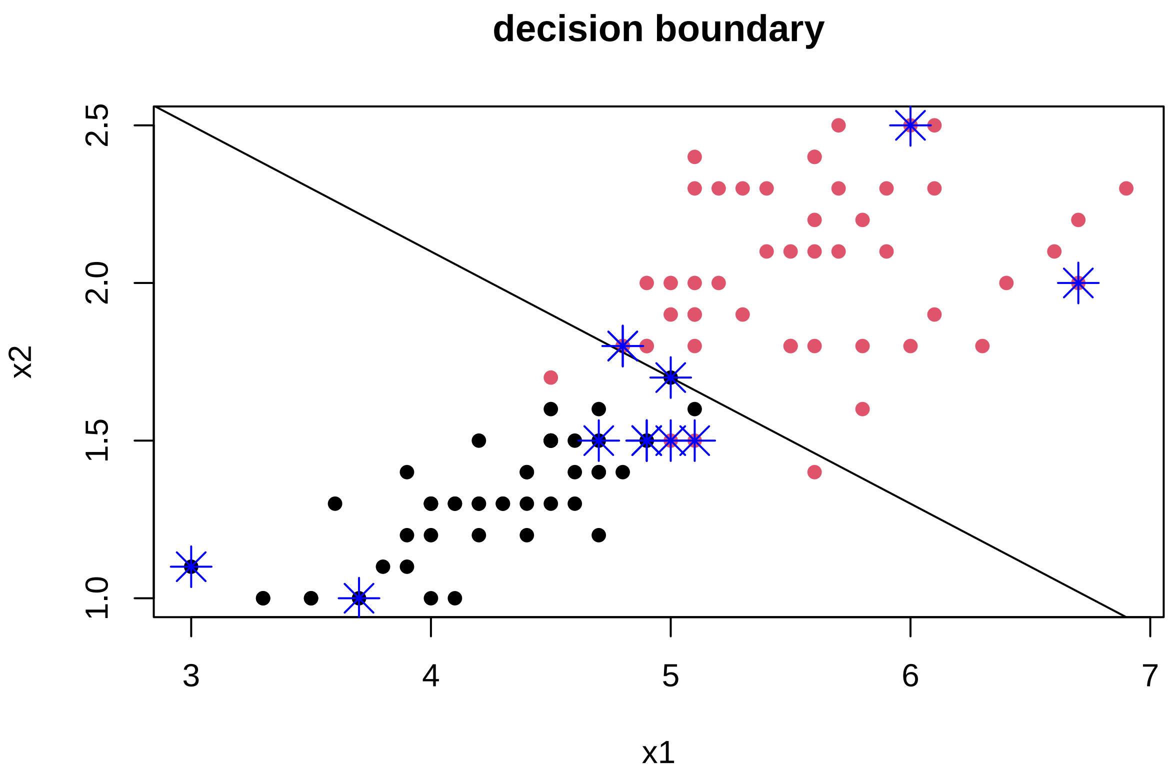


Figure 1: exercise2d

```
## $x0
##   3 23 28 32 37 49 51 70 73 77 84 89
##   1  1  1  1  1  1  1  1  1  1  1  1
##
## $x1
##    3   23   28   32   37   49   51   70   73   77   84   89
## 4.9 4.9 5.0 3.7 4.7 3.0 6.0 5.0 6.7 4.8 5.1 4.8
##
## $x2
##    3   23   28   32   37   49   51   70   73   77   84   89
## 1.5 1.5 1.7 1.0 1.5 1.1 2.5 1.5 2.0 1.8 1.5 1.8
##
## $y
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## [1,]    0    0    0    0    0    0    1    1    1     1     1     1
##
## $W
##       [,1] [,2] [,3]
## [1,] -18.5    2    5
##
## $fwd_feed
## $fwd_feed$Z
##         3   23            28   32   37 49 51 70   73  77   84  89
## [1,] -1.2 -1.2 -2.220446e-16 -6.1 -1.6 -7  6 -1 4.9 0.1 -0.8 0.1
##
## $fwd_feed$A
##                3         23  28           32         37            49         51
## [1,] 0.2314752 0.2314752 0.5 0.002237849 0.1679816 0.0009110512 0.9975274
##               70        73        77        84        89
## [1,] 0.2689414 0.9926085 0.5249792 0.3100255 0.5249792
##
##
## $pred
##      3 23 28 32 37 49 51 70 73 77 84 89
## [1,] 0  0  0  0  0  0  1  0  1  1  0  1
```

```r
plot_nl_db(x, y, W1, W2, BIAS)
points(nl_eg_model$x1, nl_eg_model$x2, pch = 8, col = "blue", cex = 2)
```



3 Neural Networks (30P total)

3.1 a 10P

Write a program that calculates the mean-squared error of the iris data for the neural network defined above. The function should take three arguments: the data vectors, the parameters defining the neural network, and the pattern classes.

```r
# calculate error
# used to determine convergence
err_mse <- function(y, A) {
  mse <- (A - y)^2
  err <- sum(mse) / 2
  return(err)
}
```

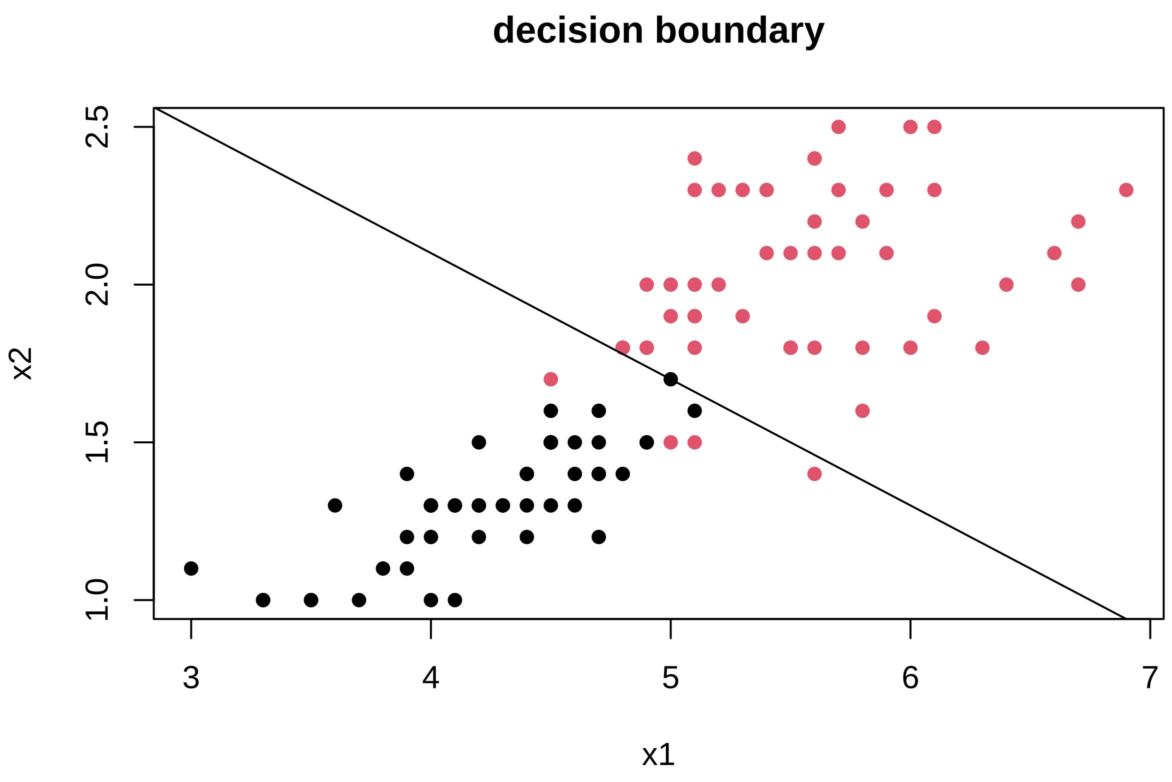
3.2 b 5P

Compute the mean squared error for two different settings of the weights (i.e. two different decision boundaries). Like above, select these by hand and choose settings that give large and small errors respectively. Plot both boundaries on the dataset as above.

```
W1A <- W1
W2A <- W2
W0A <- BIAS

weight_a <- new_weight(W1A, W2A, W0A)
nl_a <- non_linearity(x, y, weight_a)

W1B <- 1
W2B <- 7
W0B <- -23.5

weight_b <- new_weight(W1B, W2B, W0B)
nl_b <- non_linearity(x, y, weight_b)

# compute err for params a (low error)
err_mse(y, nl_a$fwd_feed$A)
```

```
## [1] 2.210654
```

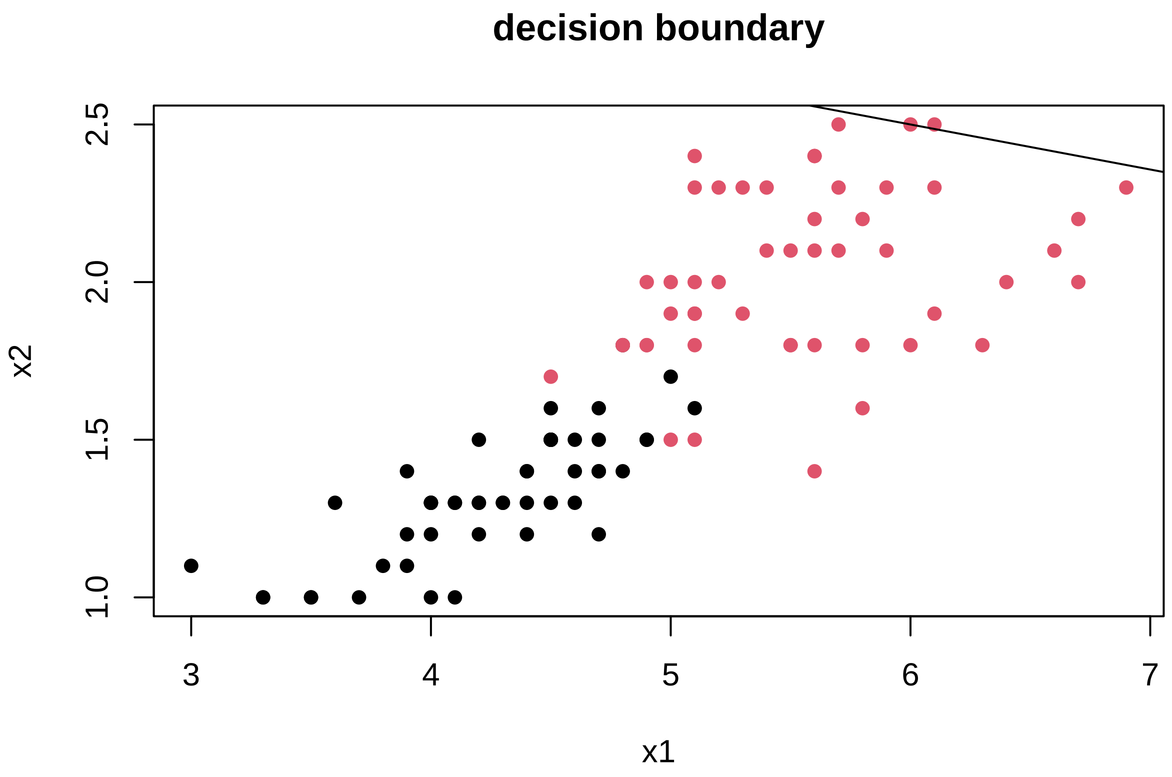
```
plot_nl_db(x, y, W1A, W2A, W0A)
```



```
# compute err for params b (high error)
err_mse(y, nl_b$fwd_feed$A)
```

```
## [1] 21.23984
```

```
plot_nl_db(x, y, W1B, W2B, W0B)
```



```
# error with random weight
nl_r <- non_linearity(x, y)
err_mse(y, nl_r$fwd_feed$A)
```

```
## [1] 14.0386
```

```
# max discovered error bound: 25 (w1 = 8, w2 = 1, w0 = -4)
```

3.3 c 10P

Give a mathematical derivation the gradient of the objective function above with respect to the neural network weights. You will have to use chain rule and the derivative of the sigmoid function as discussed in class. Use w0 to represent the bias term. You should show and explain each step.

The error function is defined as below:

$$E = \frac{1}{2}\sum_{n=1}^{N}(\hat{Y} - Y_{ref})^2 = \frac{1}{2}\sum_{n=1}^{N}(A - Y_{ref})^2$$

where A is the output of the sigmoid activation function (predicted output), Y_{ref} is the actual output

To derive the error function wrt the weight, we will use the chain rule

$$\frac{DE}{DW} = \frac{DE}{DA}\frac{DA}{DZ}\frac{DZ}{DW}$$

where A is the output of the sigmoid activation function, Z is the input for the sigmoid activition function (or the output of the multiplication WX), and W is the weight, all in vector form. After deriving with the chain rule, we get:

$$\frac{DE}{DW} = \sum_{n=1}^{N}(A)(1-A)(A-Y_{ref})X_n$$

in vector form. It should also be known that:

$$DW = \frac{DE}{DW}$$

so that it can be used when determining the update rule for the weight.

3.4 d 5P

Show how the gradient can be written in both scalar and vector form

Part c is derived in vector form

Scalar Form:

$$\frac{DE}{DW} = \sum_{n=1}^{N}\sum_{i=1}^{I}(a_i)(1-a_i)(a_i-y_{i_{ref}})x_{ni}$$

where N is number of dimensions (number of variables); I is number of samples

Vector Form:

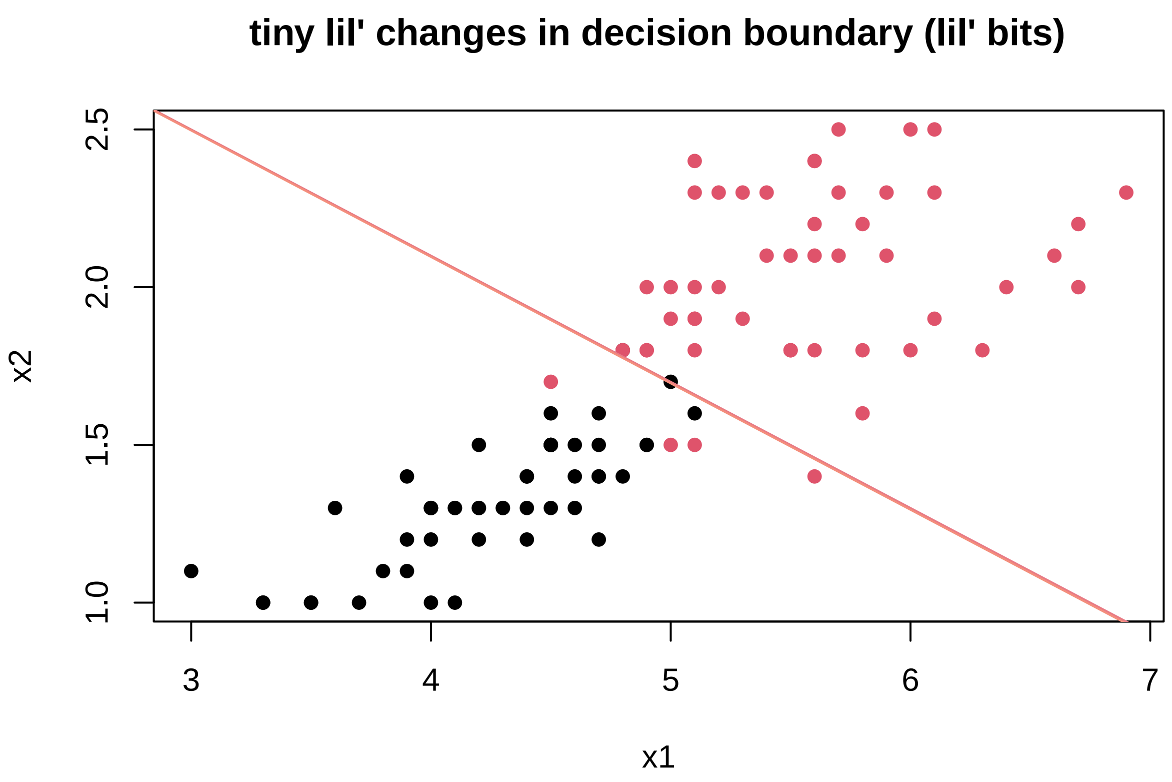
$$\frac{DE}{DW} = \sum_{n=1}^{N}(A)(1-A)(A-Y_{actual})X_n$$

## e 10P

Write code that computes the summed gradient for an ensemble of patterns. Illustrate the gradient by showing (i.e. plotting) how the decision boundary changes for a small step.

```r
# gradient derivation from error (back propagation)
# the code that "computes the summed gradient for an ensemble of patterns"
back_prop <- function(x, y, A) {
  m <- dim(x)[2]

  ones <- t(matrix(rep(1, m)))
  dA <- A * (ones - A)
  dZ <- (A - y)
  deriv_step <- dA * dZ
  dW <- x %*% t(deriv_step)
  return(t(dW))
}

# gradient descent
# (update weight)
descend <- function(dW, W, N) {
  eps <- 0.1 / N
  update <- eps %*% dW
  new_W <- W - update
  return(new_W)
}
```

```r
three_e_m <- c()
three_e_b <- c()

moon <- x_nn(x)
earth <- dim(moon)[2]

small_step_for_man <- new_weight(W1, W2, BIAS)
large_leap_for_mankind <- get_slope_intercept(small_step_for_man)
to_the_moon <- fwd_prop(moon, y, small_step_for_man)
three_e_m <- c(three_e_m, large_leap_for_mankind$m)
three_e_b <- c(three_e_b, large_leap_for_mankind$b)

back_to_earth <- back_prop(moon, y, to_the_moon$A)
small_step_for_man <- descend(back_to_earth, small_step_for_man, earth)
large_leap_for_mankind <- get_slope_intercept(small_step_for_man)
three_e_m <- c(three_e_m, large_leap_for_mankind$m)
three_e_b <- c(three_e_b, large_leap_for_mankind$b)

plot(x[1, ], x[2, ],
     col = as.factor(y),
     pch = 16,
     xlab = "x1", ylab = "x2",
     main = "tiny lil' changes in decision boundary (lil' bits)")

abline(three_e_b[1], three_e_m[1], col = randomColor())
abline(three_e_b[2], three_e_m[2], col = randomColor())
```



4 Learning Decision Boundary through optimization (30P total)

4.1 a 5P

Using your code above, write a program that implements gradient descent to optimize the decision boundary for the iris dataset.

```
# putting it all together
# keep history of decision boundary data
# keep history of iteration and "objective function" data
# 10 max iterations by default
nn_model <- function(x, y, weight = NULL, bias = NULL, min_diff = 1e-6) {
  # initialization
  X <- x_nn(x)
  N <- dim(X)[2]

  # reused vars
  W <- initial_weight(X, y, weight, bias)
  fwd_pass <- fwd_prop(X, y, W)
  Z <- fwd_pass$Z
  A <- fwd_pass$A
  err <- err_mse(y, A)
  dec_bound <- get_slope_intercept(W)
  # preservation of history
  m_hist <- c(dec_bound$m)
  b_hist <- c(dec_bound$b)
  err_hist <- c(err)
```

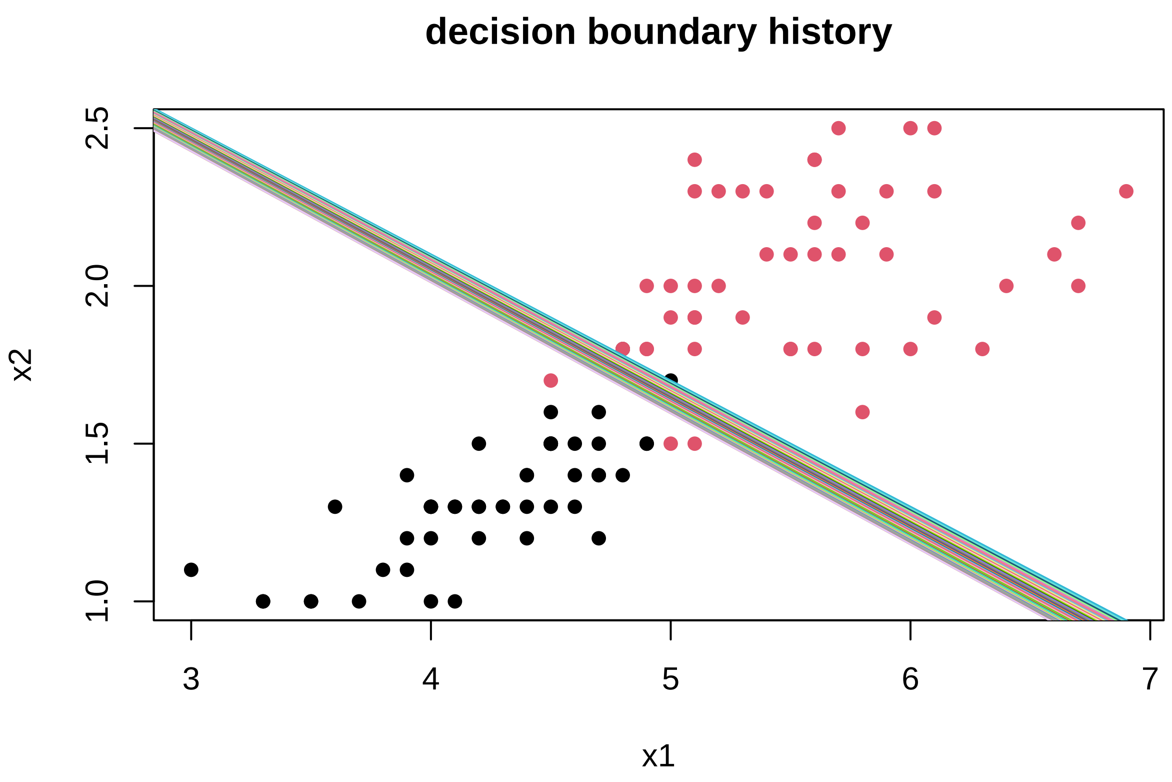
```r
  # increment vars
  diff <- abs(0 - err)
  iter <- 1
  # iteration history
  diff_hist <- c(diff)
  iter_hist <- c(iter)
  while (diff > min_diff) { # diff > 0 |
    # loop preprocess
    back_pass <- back_prop(X, y, A)
    # reused vars
    W <- descend(back_pass, W, N)
    fwd_pass <- fwd_prop(X, y, W)
    Z <- fwd_pass$Z
    A <- fwd_pass$A
    err <- err_mse(y, A)
    dec_bound <- get_slope_intercept(W)
    # preservation of history
    m_hist <- c(m_hist, dec_bound$m)
    b_hist <- c(b_hist, dec_bound$b)
    err_hist <- c(err_hist, err)
    # increments
    diff <- abs(diff - err)
    iter <- iter + 1
    # increment history
    diff_hist <- c(diff_hist, diff)
    iter_hist <- c(iter_hist, iter)
  }

  model <- list("X"=X,
                "y"=y,
                "W"=W,
                "Z"=fwd_pass$Z,
                "A"=fwd_pass$A,
                "err"=err,
                "dec_bound"=dec_bound,
                "err_hist"=err_hist,
                "b_hist"=b_hist,
                "m_hist"=m_hist,
                "diff_hist"=diff_hist,
                "iter_hist"=iter_hist,
                "iteration"=iter)

  return(model)
}

my_weight <- new_weight(W1, W2, BIAS)
nn_full <- nn_model(x, y, my_weight, BIAS)
```

4.2 b 5P

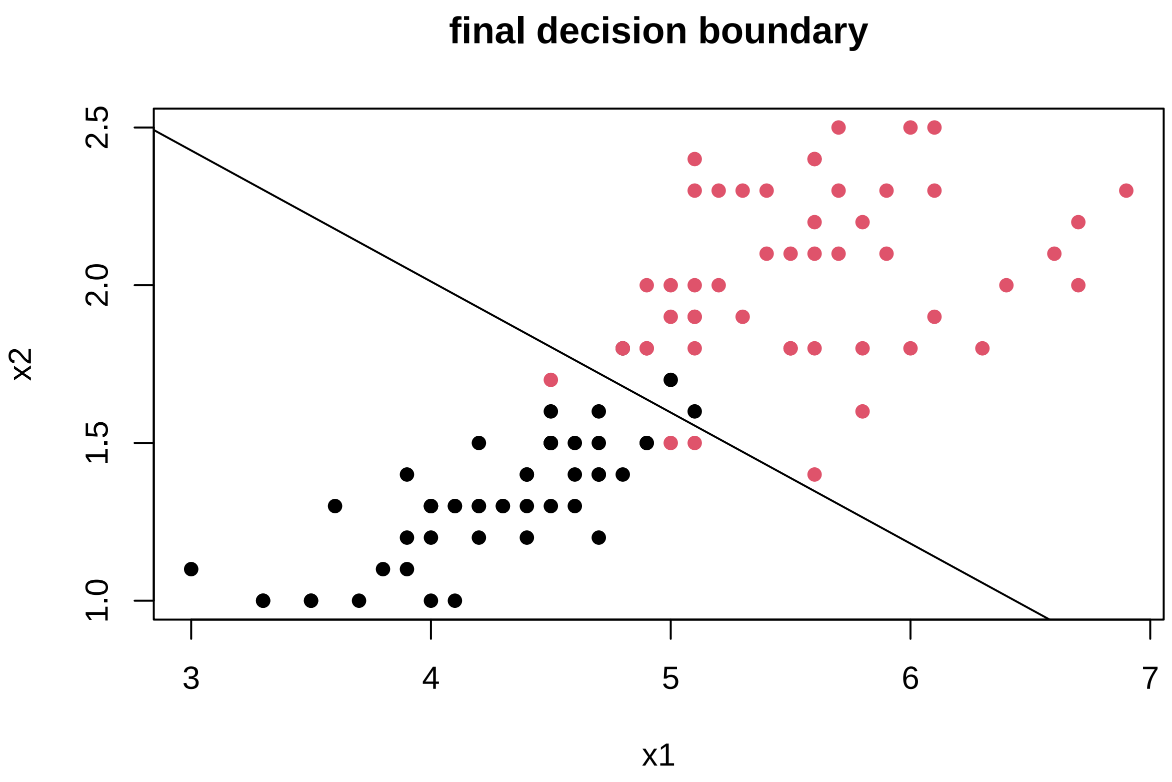
In your program, include code that shows the progress in two plots: the first should show the current decision boundary location overlaid on the data; the second should show the learning curve, i.e. a plot of the objective function as a function of the iteration.

```r
# plot functions
# plot decision boundary history over time
# "x0" = X[1, ],
# "x1" = X[2, ],
# "x2" = X[3, ],
plot_dec_bound_hist <- function(nn_model) {
  x1 <- nn_model$X[2, ]
  x2 <- nn_model$X[3, ]
  y <- as.factor(nn_model$y)
  tital <- paste("decision boundary history")
  plot(x1, x2,
       col = y, pch = 16,
       xlab = "x1", ylab = "x2",
       main = tital)
  for (i in nn_model$iter_hist) {
    a <- nn_model$b_hist[i]
    b <- nn_model$m_hist[i]
    abline(a, b, col = randomColor())
  }
}

plot_db_nn <- function(nn_model) {
    x1 <- nn_model$X[2, ]
    x2 <- nn_model$X[3, ]
    y <- as.factor(nn_model$y)
    tital <- paste("final decision boundary")
    plot(x1, x2,
         col = y, pch = 16,
         xlab = "x1", ylab = "x2",
         main = tital)
    a <- nn_model$dec_bound$b
    b <- nn_model$dec_bound$m
    abline(a, b, col = "black")
}

# plot objective function by iteration over time
plot_nn_learning <- function(nn_model) {
  plot(nn_model$iter_hist, nn_model$err_hist,
       type = "l",
       pch = 16, col = "black",
       xlab = "iteration", ylab = "objective function",
       main = "objective function v iteration for nn")
}

plot_step_sizes <- function(nn_model) {
  plot(nn_model$iter_hist, nn_model$diff_hist,
       type = "l",
       pch = 16, col = "black",
       xlab = "iteration", ylab = "step size",
       main = "step sizes over iteration")
}
```

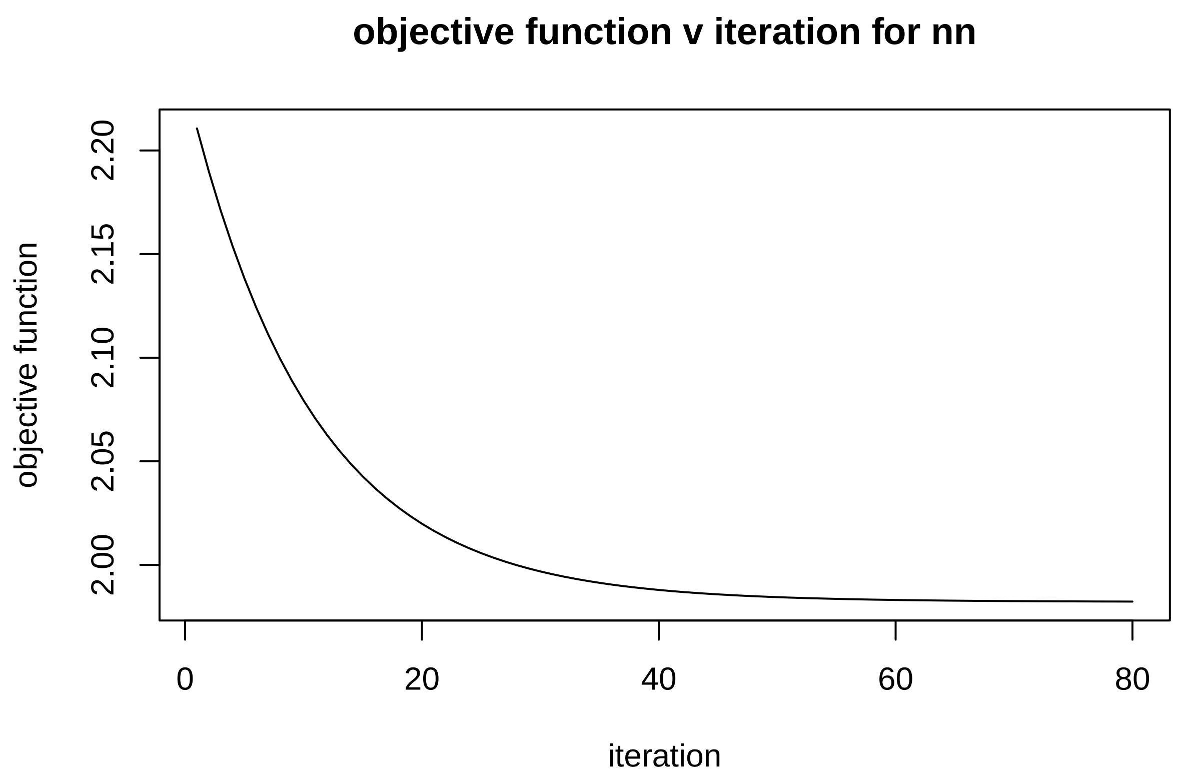
```
plot_dec_bound_hist(nn_full)
```



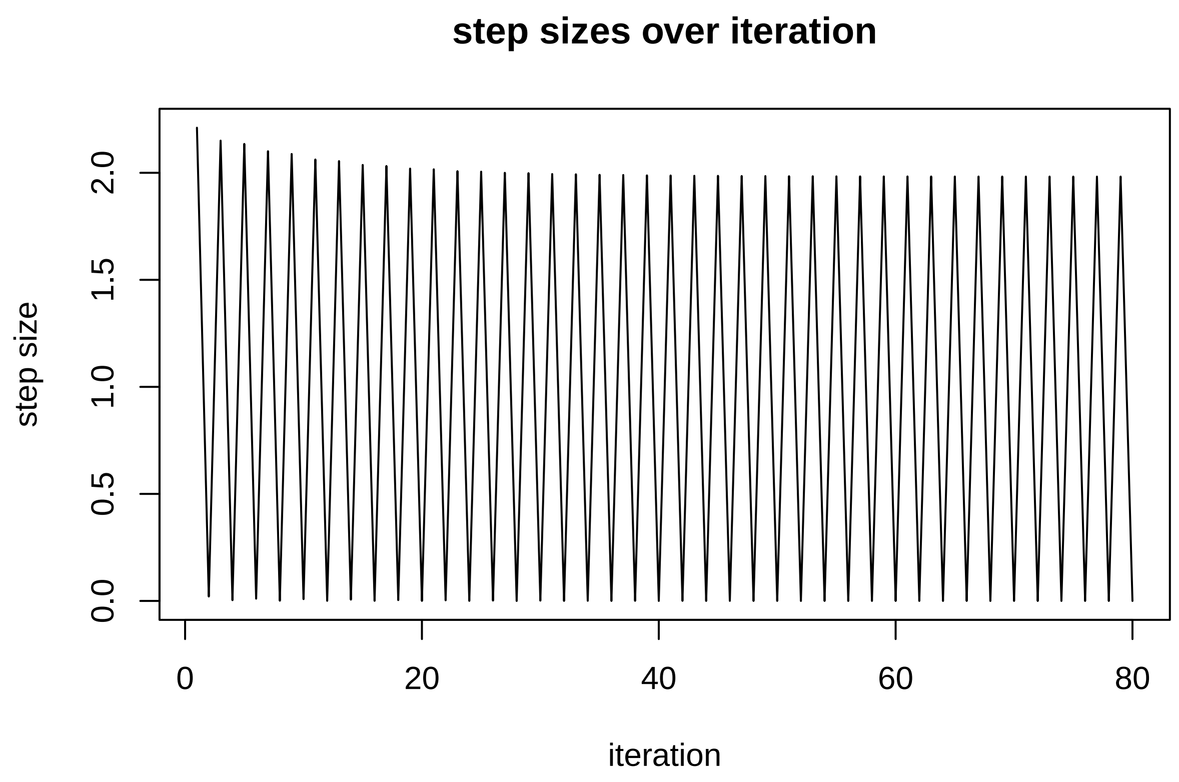
```
plot_db_nn(nn_full)
```



```
plot_nn_learning(nn_full)
```



```
plot_step_sizes(nn_full)
```



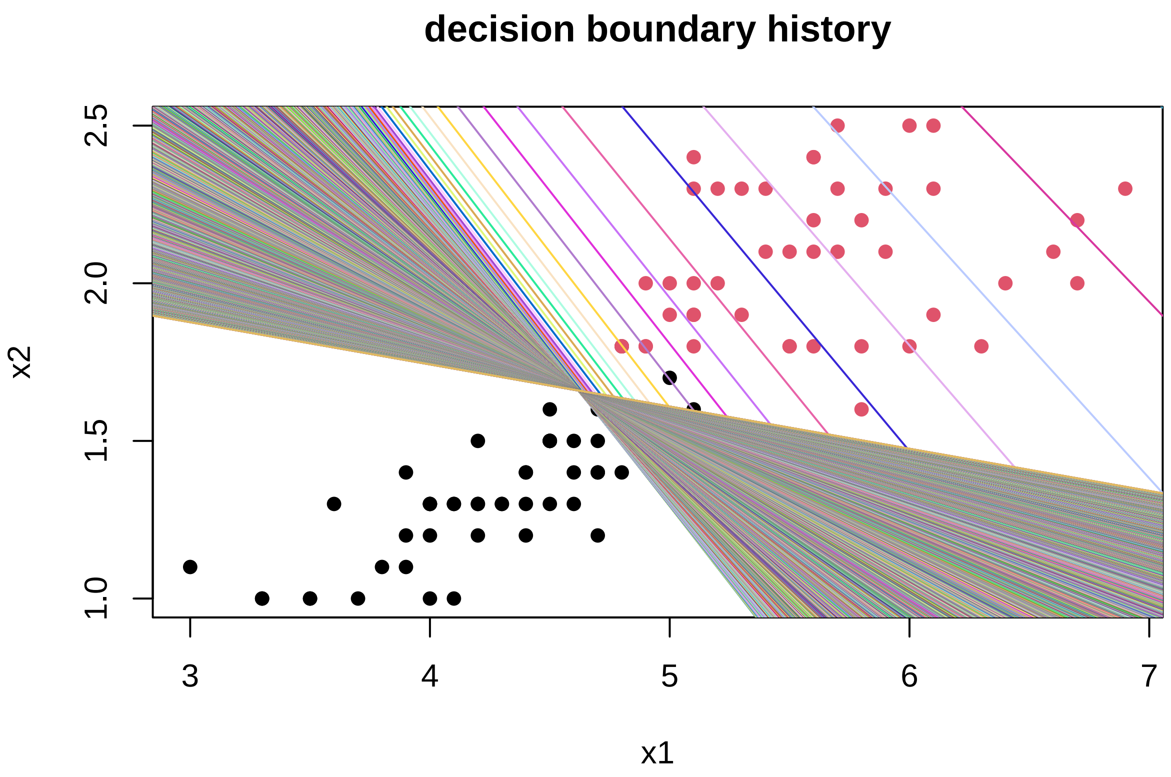
4.3 c 10P

Run your code on the iris data set starting from a random setting of the weights.

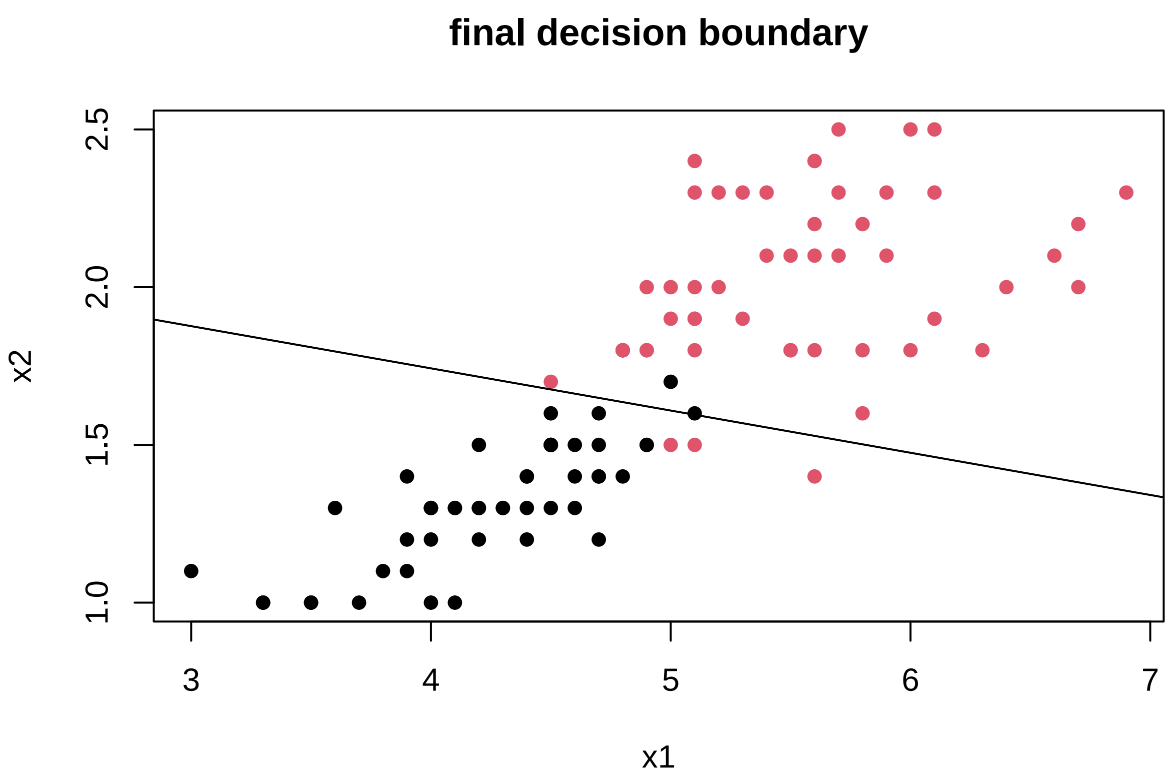
Note: you might need to restrict the degree of randomness so that the initial decision boundary is visible somewhere in the plot.

In your writeup, show the two output plots at the initial, middle, and final locations of the decision boundary

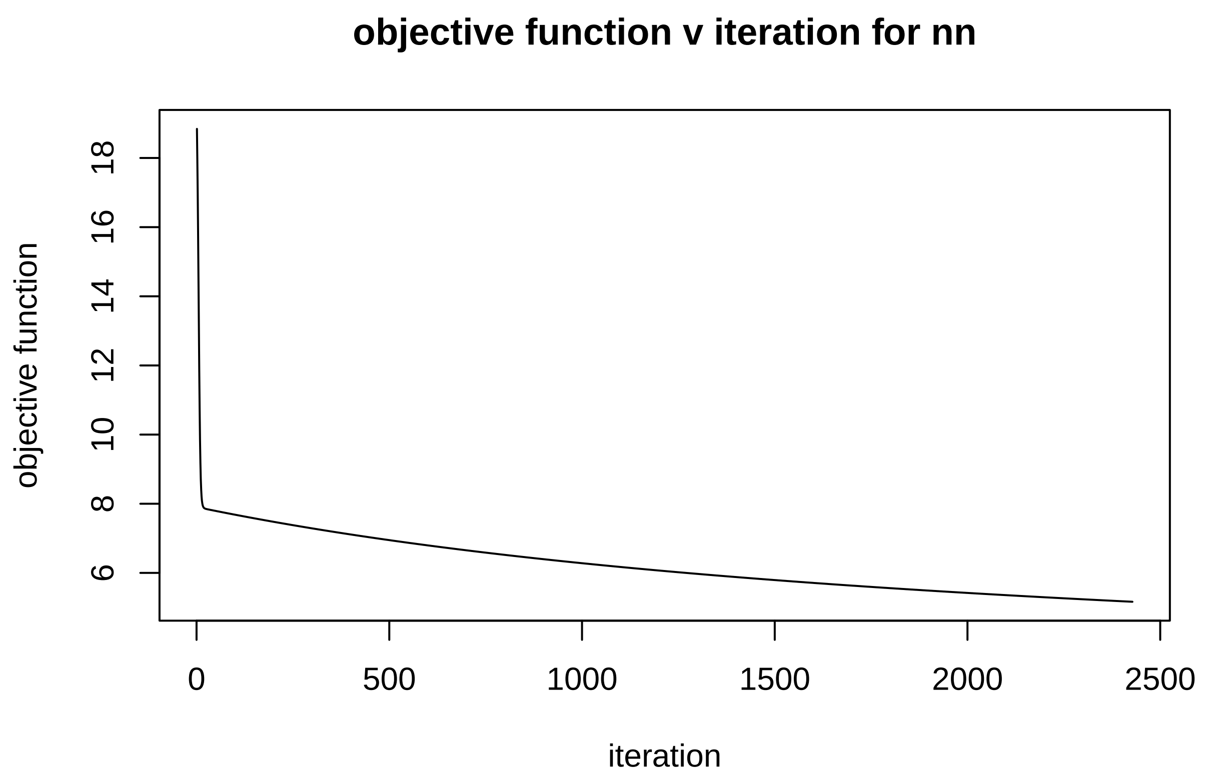
```
nn_random <- nn_model(x, y, bias = -3, min_diff = 1e-9)
plot_dec_bound_hist(nn_random)
```



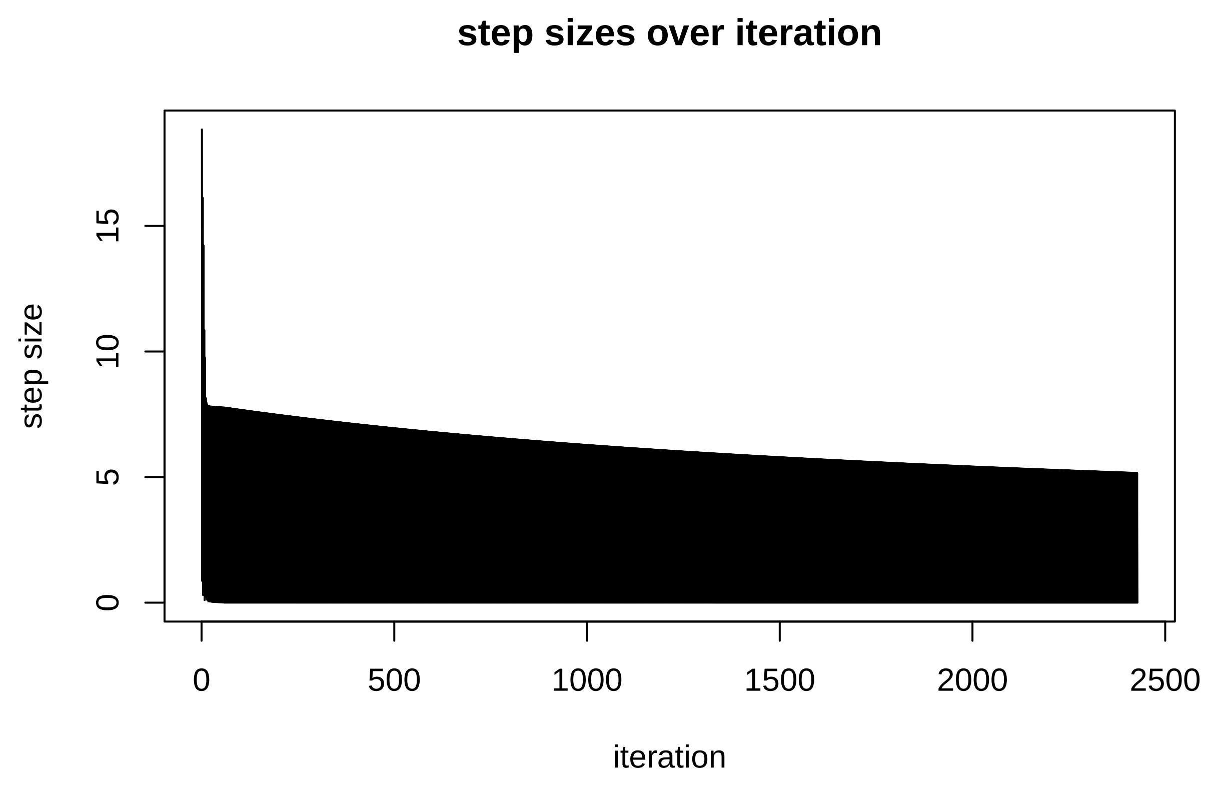
```
plot_db_nn(nn_random)
```



```
plot_nn_learning(nn_random)
```



```
plot_step_sizes(nn_random)
```



4.4 d 5P

Explain how you chose the gradient step size

Minimum step size is generally defined to be a value that's 0.001 or smaller. To choose an appropriate step size, we could look at the differences between errors throughout history:

Once the history in the differences between errors has been examined, the minimum gradient step size was chosen to be $exp(-6) = e^{-6} = 10^{-6} = 0.000001$ and is the point where we stop the learning process. When testing, this step size proved to be efficient.

But if this is referring to the ϵ value (gradient step size ~= learning rate), then during class discussion, it was explained to be

$$\epsilon = \frac{0.1}{N}$$

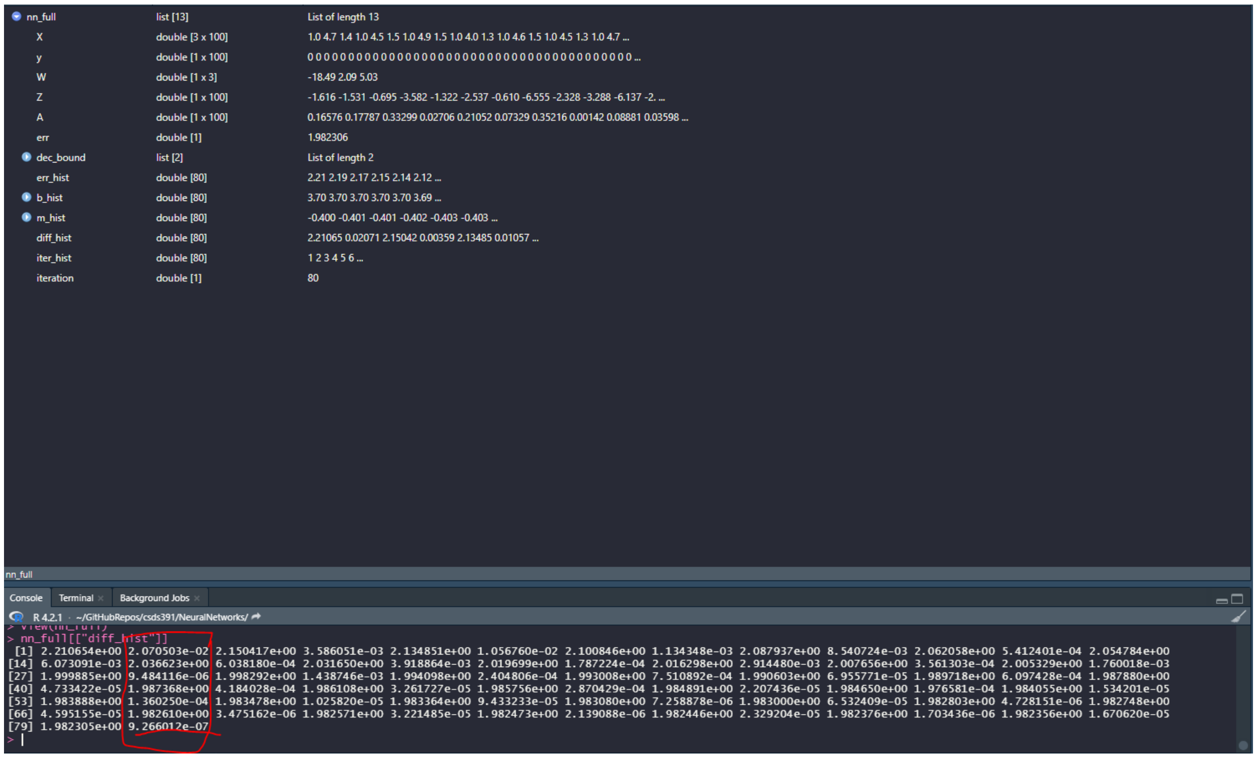
where N is # samples (observations, ...). I ultimately let the code handle the step size. This gives us slow convergence.

4.5 e 5P

Explain how you chose a stopping criterion

Initially decided on max iteration, but max iteration requires manual input. A more dynamic approach to discovering iteration would be better.

So instead of having a max iteration, we could have a minimum difference threshold (min gradient step size) that we think is appropriate which indicates a point where we'd assume the objective function (differences in error) to have been converged.



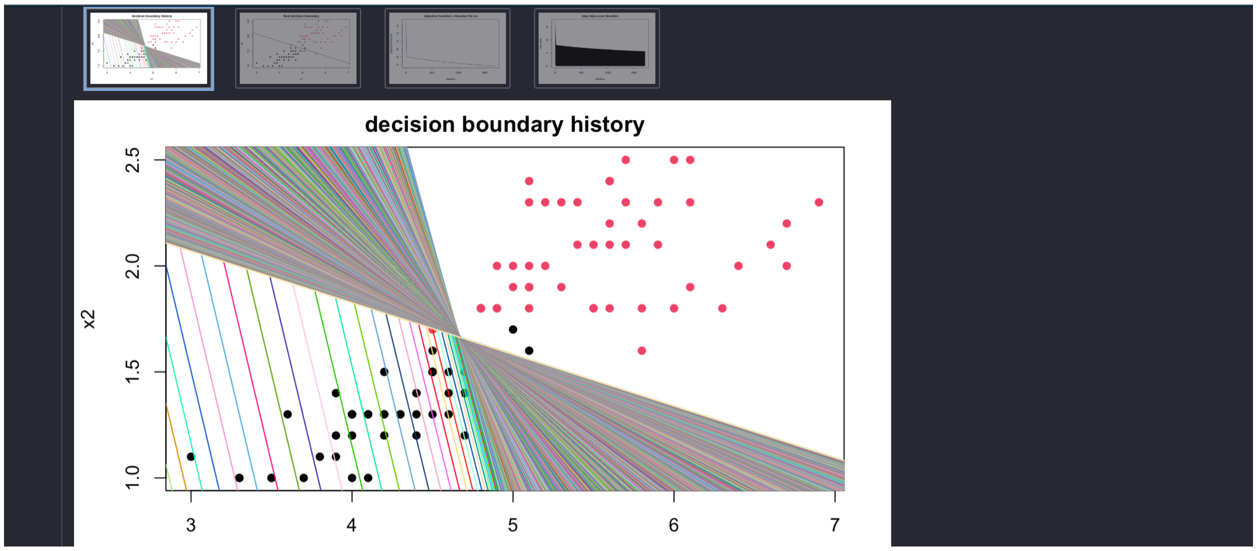
```
nn_full       list [13]         List of length 13
  X           double [3 x 100]  1.0 4.7 1.4 1.0 4.5 1.5 1.0 4.9 1.5 1.0 4.0 1.3 1.0 4.6 1.5 1.0 4.5 1.3 1.0 4.7 ...
  y           double [1 x 100]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
  W           double [1 x 3]    -18.49 2.09 5.03
  Z           double [1 x 100]  -1.616 -1.531 -0.695 -3.582 -1.322 -2.537 -0.610 -6.555 -2.328 -3.288 -6.137 -2. ...
  A           double [1 x 100]  0.16576 0.17787 0.33299 0.02706 0.21052 0.07329 0.35216 0.00142 0.08881 0.03598 ...
  err         double [1]        1.982306
  dec_bound   list [2]          List of length 2
  err_hist    double [80]       2.21 2.19 2.17 2.15 2.14 2.12 ...
  b_hist      double [80]       3.70 3.70 3.70 3.70 3.70 3.69 ...
  m_hist      double [80]       -0.400 -0.401 -0.401 -0.402 -0.403 -0.403 ...
  diff_hist   double [80]       2.21065 0.02071 2.15042 0.00359 2.13485 0.01057 ...
  iter_hist   double [80]       1 2 3 4 5 6 ...
  iteration   double [1]        80
```

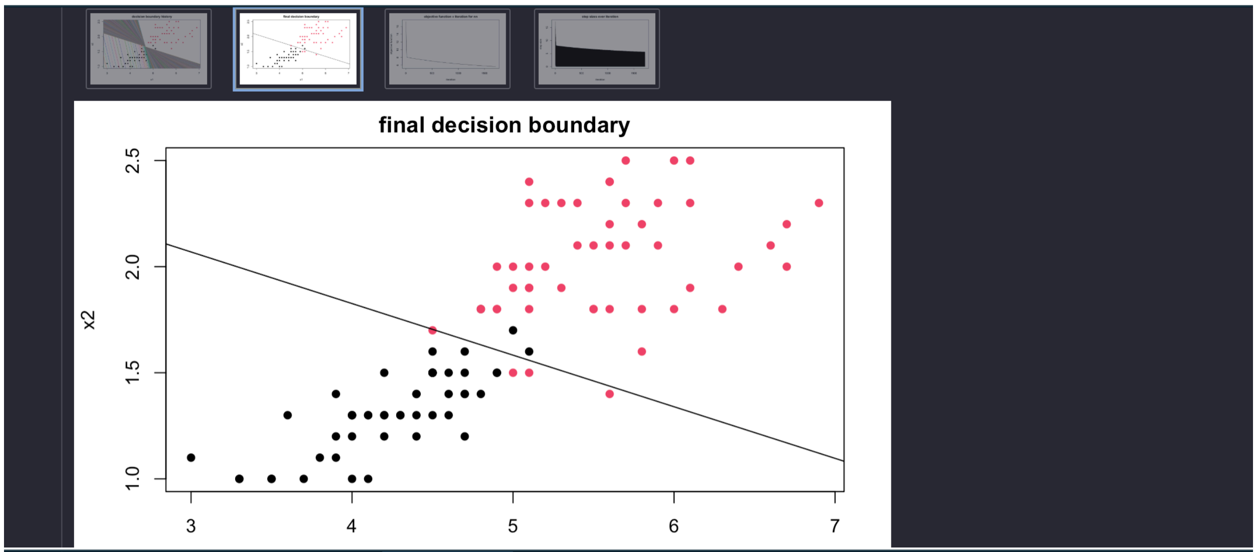
```
R 4.2.1 · ~/GitHubRepos/csds391/NeuralNetworks/
> View(nn_full)
> nn_full[["diff_hist"]]
 [1] 2.210654e+00 2.070503e-02 2.150417e+00 3.586051e-03 2.134851e+00 1.056760e-02 2.100846e+00 1.134348e-03 2.087937e+00 8.540724e-03 2.062058e+00 5.412401e-04 2.054784e+00
[14] 6.073091e-03 2.036623e+00 6.038180e-04 2.031650e+00 3.918864e-03 2.019699e+00 1.787224e-04 2.016298e+00 2.914480e-03 2.007656e+00 3.561303e-04 2.005329e+00 1.760018e-03
[27] 1.999885e+00 9.484116e-06 1.998292e+00 1.438746e-03 1.994098e+00 2.404806e-04 1.993008e+00 7.510892e-04 1.990603e+00 6.955771e-05 1.989718e+00 6.097428e-04 1.987880e+00
[40] 4.733422e-05 1.987368e+00 4.184028e-04 1.986108e+00 3.261727e-05 1.985756e+00 2.870429e-04 1.984891e+00 2.207436e-05 1.984650e+00 1.976581e-04 1.984055e+00 1.534201e-05
[53] 1.983888e+00 1.360250e-04 1.983478e+00 1.025820e-05 1.983364e+00 9.433233e-05 1.983080e+00 7.258878e-06 1.983000e+00 6.532409e-05 1.982803e+00 4.728151e-06 1.982748e+00
[66] 4.595155e-05 1.982610e+00 3.475162e-06 1.982571e+00 3.221485e-05 1.982473e+00 2.139088e-06 1.982446e+00 2.329204e-05 1.982376e+00 1.703436e-06 1.982356e+00 1.670620e-05
[79] 1.982305e+00 9.266012e-07
> 
```

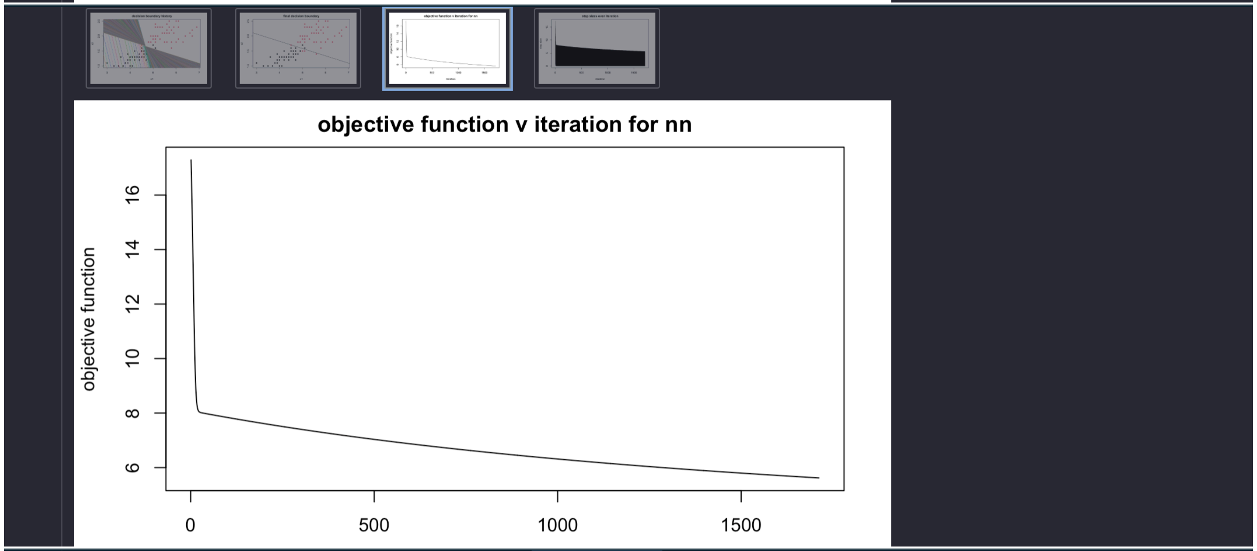
Figure 2: ex4d

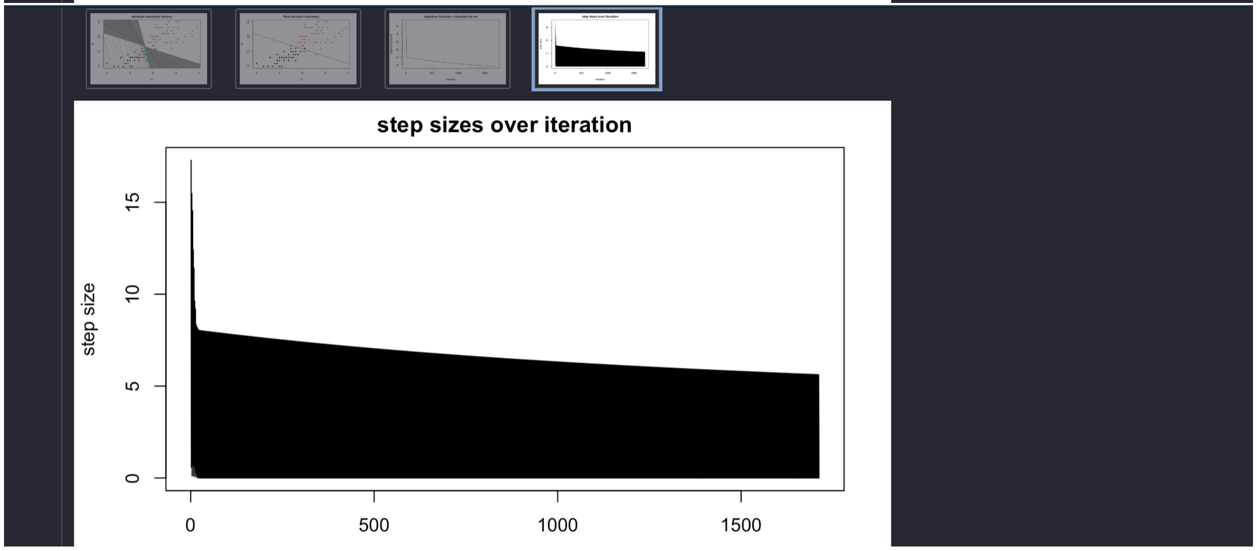
Basically the difference in error doesn't have to be exactly 0, but a value close to 0. The greater the precision, the more accurate the data becomes, but the less payoff there is. A good default value I came up with is $exp(-6) = e^{-6} = 10^{-6} = 0.000001$. It's precise enough for assuming convergence, but can be changed depending on how much of a convergence is desired. This allows the error and decision boundary to converge to a certain point given the weights, meaning the error depends on how accurate the weights are and never really hits 0 (at least based on my error implementation).

There may be some issues when plotting this randomly, but we'll have to "restrict the degree of randomness" to ensure a valid input space:




final decision boundary
x2
1.0
1.5
2.0
2.5
3
4
5
6
7


objective function v iteration for nn
objective function
6
8
10
12
14
16
0
500
1000
1500


step sizes over iteration
step size
0
5
10
15
0
500
1000
1500

5 Using an ML Toolbox (27P total)

Use a machine learning toolbox to generalize the network to perform classification on the full iris dataset.

Your network should have one or more hidden layers and classify all three iris classes using all four data dimensions. Show your results and explain your architecture. In addition to implementing the neural network classifier, also think of an issue to explore.

Write your own mini-tutorial to report your results. Here are some examples of topics you can consider: - Illustrate how a multi-layer network makes a decision. - Explore different non-linearities. - Explore different network architectures and their classication and generalization performance. - Explore algorithms for adjusting the learning rate.

There are many tutorials using different toolboxes available on the internet. You can follow any you find useful to get started but do not just copy them for this extra credit portion of the assignment. You can make use of material you find useful, but cite your sources in your writeup. The TAs will check for straight copying, so please don't do this. If you do use material from a source, simplify it to focus on a single topic and understand it well enough that you can explain it and modify it for our own purposes. Your write up should explain the ideas, code, and results in your own words and make use of tables or figures. The extra credit is worth 20% of the points, so it should be about 20% of the effort for the main part of the assignment. **Trivial examples will receive little or no credit.** Your goal should be to teach yourself about something about a neural network-related topic, and the write up should written as if you are explaining it to one of your classmates.

Step 1 -> Figure out an appropriate ml toolbox to use (some options: neuralnet, nnet, tensorflow, keras, ...). This can change depending on how the data will be analyzed.

Multi Layered Neural Network How-To (for training/learning only)

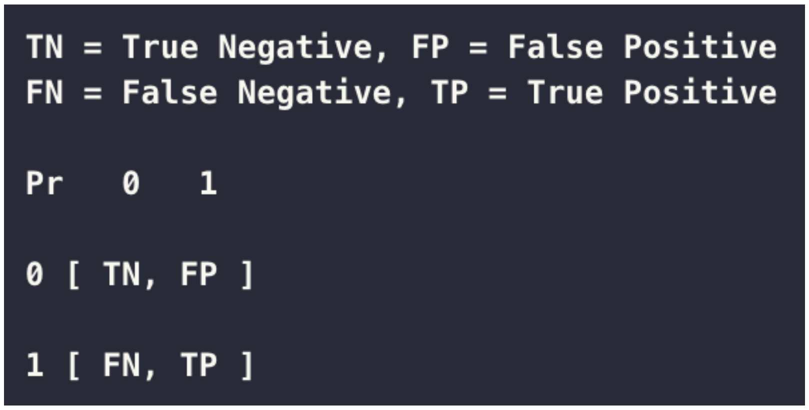
Step 1 -> Perform analysis on the data to determine how many layers are needed, what activation functions will be used for each layer, and how big each layer will be (neural network architecture for data), and the error for the data (maybe, because it's highly likely that the error is dynamically derived based on the activation function used). It may be important to understand the mathematical derivation of the architecture before building it out.

Step 2 -> Consider the analyses and plotting that will be performed to prove learning and learning efficiency

Step 3 -> Start building designed architecture

Step 4 -> Plot learning and learning performance, analyze general performance of neural network. Useful performance metrics:

confusion matrix



```
TN = True Negative, FP = False Positive
FN = False Negative, TP = True Positive

Pr   0   1

0 [ TN, FP ]

1 [ FN, TP ]
```

Figure 3: confusion matrix

precision = # TP / (# TP + # FP) recall = # TP / (# TP + # FN) f1 score = harmonic mean of precision and recall (2 * (p * r) / (p + r)) accuracy = percentage of the all correct predictions out total predictions made

```r
# confusion matrix
# TN = True Negative, FP = False Positive
# FN = False Negative, TP = True Positive
#      0   1
# 0 [ TN, FP ]
# 1 [ FN, TP ]
confusion_matrix <- function(y, y_prediction) {
  tb_cm <- table(y, y_prediction)
  return(tb_cm)
}

# metrics of performance
# precision (p) =  # true positives / # true positives + # false positives
# recall (r)    =  # true positives / # true positives + # false negatives
# f1 score (f1) =  harmonic mean of precision and recall (2 * (p * r) / (p + r))
# accuracy (a)  =  percent of correct predictions out of total predictions
#               =  (#tp + #tn)/(#tp + #fp + #tn + #fn)
metrics <- function(tb) {
  tn <- tb[1]
  tp <- tb[4]
  fn <- tb[3]
  fp <- tb[2]
  a <- (tp + tn) / (tp + fp + tn + fn)
  r <- tp / (tp + fn)
  p <- tp / (tp + fp)
  f1 <- 2 * ((p * r) / (p + r))

  metric <- list("acc" = a,
                 "rec" = r,
                 "prec" = p,
                 "f" = f1)
  return(metric)
}
```

We're going to be using the full iris dataset to perform learning/training, and no testing. We will be using rules of convergence to see how

```r
factor_species <- function(species) {
  cols <- c()
  n_species <- length(species)
  for (i in 1:n_species) {
    s <- species[i]
    if (s == "setosa") {
      cols <- c(cols, 0)
      next
    }
    if (s == "versicolor") {
      cols <- c(cols, 1)
      next
    }
    if (s == "virginica") {
      cols <- c(cols, 2)
    }
  }
  return(cols)
}

iris_data <- mutate(iris, species = factor_species(Species))[ , -5]
```

```r
iris_net <- neuralnet(species ~ Petal.Length+Petal.Width+Sepal.Length+Sepal.Width,
                      data = iris_data, hidden = c(4, 4), linear.output = FALSE,
                      act.fct = 'logistic', err.fct = 'sse', threshold = 1e-9)

plot(iris_net, col.hidden = 'darkgreen', col.hidden.synapse = 'red')
iris_net$result.matrix
```

```
##                                   [,1]
## error                     2.500000e+01
## reached.threshold         8.527401e-10
## steps                     9.100000e+01
## Intercept.to.1layhid1     6.123830e+00
## Petal.Length.to.1layhid1 -7.941937e+00
## Petal.Width.to.1layhid1  -8.733563e+00
## Sepal.Length.to.1layhid1  1.265729e+00
## Sepal.Width.to.1layhid1   7.083766e+00
## Intercept.to.1layhid2     7.002578e+00
## Petal.Length.to.1layhid2 -3.985633e+00
## Petal.Width.to.1layhid2  -6.846319e+00
## Sepal.Length.to.1layhid2 -5.166903e-01
## Sepal.Width.to.1layhid2   4.970781e+00
## Intercept.to.1layhid3    -6.461122e+00
## Petal.Length.to.1layhid3  7.927633e+00
## Petal.Width.to.1layhid3   9.884172e+00
## Sepal.Length.to.1layhid3 -1.695508e+00
## Sepal.Width.to.1layhid3  -6.527172e+00
## Intercept.to.1layhid4     7.230446e+00
## Petal.Length.to.1layhid4 -8.241996e+00
## Petal.Width.to.1layhid4  -9.466023e+00
## Sepal.Length.to.1layhid4  6.811062e-01
## Sepal.Width.to.1layhid4   8.458762e+00
## Intercept.to.2layhid1    -7.242578e+00
## 1layhid1.to.2layhid1      1.011210e+01
## 1layhid2.to.2layhid1      7.807804e+00
## 1layhid3.to.2layhid1     -9.437337e+00
## 1layhid4.to.2layhid1      8.825820e+00
## Intercept.to.2layhid2     6.416430e+00
## 1layhid1.to.2layhid2     -8.407548e+00
## 1layhid2.to.2layhid2     -6.890137e+00
## 1layhid3.to.2layhid2      9.593733e+00
## 1layhid4.to.2layhid2     -8.718054e+00
## Intercept.to.2layhid3    -6.971373e+00
## 1layhid1.to.2layhid3      8.809455e+00
## 1layhid2.to.2layhid3      7.823042e+00
## 1layhid3.to.2layhid3     -9.138874e+00
## 1layhid4.to.2layhid3      8.624464e+00
## Intercept.to.2layhid4     8.092038e+00
## 1layhid1.to.2layhid4     -8.229436e+00
## 1layhid2.to.2layhid4     -8.654256e+00
## 1layhid3.to.2layhid4      8.795649e+00
## 1layhid4.to.2layhid4     -1.097683e+01
## Intercept.to.species      5.245634e+00
## 2layhid1.to.species      -8.881751e+00
## 2layhid2.to.species       9.920378e+00
## 2layhid3.to.species      -8.896341e+00
## 2layhid4.to.species       9.628580e+00
```

```
iris_net$net.result
```

```
## [[1]]
##                 [,1]
##   [1,] 3.607623e-06
##   [2,] 3.607623e-06
##   [3,] 3.607623e-06
##   [4,] 3.607623e-06
##   [5,] 3.607623e-06
##   [6,] 3.607623e-06
##   [7,] 3.607623e-06
##   [8,] 3.607623e-06
##   [9,] 3.607623e-06
##  [10,] 3.607623e-06
##  [11,] 3.607623e-06
##  [12,] 3.607623e-06
##  [13,] 3.607623e-06
##  [14,] 3.607623e-06
##  [15,] 3.607623e-06
##  [16,] 3.607623e-06
##  [17,] 3.607623e-06
##  [18,] 3.607623e-06
##  [19,] 3.607623e-06
##  [20,] 3.607623e-06
##  [21,] 3.607623e-06
##  [22,] 3.607623e-06
##  [23,] 3.607623e-06
##  [24,] 3.607623e-06
##  [25,] 3.607623e-06
##  [26,] 3.607623e-06
##  [27,] 3.607623e-06
##  [28,] 3.607623e-06
##  [29,] 3.607623e-06
##  [30,] 3.607623e-06
##  [31,] 3.607623e-06
##  [32,] 3.607623e-06
##  [33,] 3.607623e-06
##  [34,] 3.607623e-06
##  [35,] 3.607623e-06
##  [36,] 3.607623e-06
##  [37,] 3.607623e-06
##  [38,] 3.607623e-06
##  [39,] 3.607623e-06
##  [40,] 3.607623e-06
##  [41,] 3.607623e-06
##  [42,] 3.607623e-06
##  [43,] 3.607623e-06
##  [44,] 3.607623e-06
##  [45,] 3.607623e-06
##  [46,] 3.607623e-06
##  [47,] 3.607623e-06
##  [48,] 3.607623e-06
##  [49,] 3.607623e-06
##  [50,] 3.607623e-06
##  [51,] 1.000000e+00
##  [52,] 1.000000e+00
##  [53,] 1.000000e+00
##  [54,] 1.000000e+00
##  [55,] 1.000000e+00
##  [56,] 1.000000e+00
##  [57,] 1.000000e+00
##  [58,] 1.000000e+00
##  [59,] 1.000000e+00
##  [60,] 1.000000e+00
##  [61,] 1.000000e+00
##  [62,] 1.000000e+00
##  [63,] 1.000000e+00
##  [64,] 1.000000e+00
##  [65,] 1.000000e+00
##  [66,] 1.000000e+00
##  [67,] 1.000000e+00
##  [68,] 1.000000e+00
##  [69,] 1.000000e+00
##  [70,] 1.000000e+00
##  [71,] 1.000000e+00
##  [72,] 1.000000e+00
##  [73,] 1.000000e+00
##  [74,] 1.000000e+00
##  [75,] 1.000000e+00
##  [76,] 1.000000e+00
##  [77,] 1.000000e+00
##  [78,] 1.000000e+00
##  [79,] 1.000000e+00
##  [80,] 1.000000e+00
##  [81,] 1.000000e+00
##  [82,] 1.000000e+00
##  [83,] 1.000000e+00
##  [84,] 1.000000e+00
##  [85,] 1.000000e+00
##  [86,] 1.000000e+00
##  [87,] 1.000000e+00
##  [88,] 1.000000e+00
##  [89,] 1.000000e+00
##  [90,] 1.000000e+00
##  [91,] 1.000000e+00
##  [92,] 1.000000e+00
##  [93,] 1.000000e+00
##  [94,] 1.000000e+00
##  [95,] 1.000000e+00
##  [96,] 1.000000e+00
##  [97,] 1.000000e+00
##  [98,] 1.000000e+00
##  [99,] 1.000000e+00
## [100,] 1.000000e+00
## [101,] 1.000000e+00
## [102,] 1.000000e+00
## [103,] 1.000000e+00
## [104,] 1.000000e+00
## [105,] 1.000000e+00
## [106,] 1.000000e+00
## [107,] 1.000000e+00
## [108,] 1.000000e+00
## [109,] 1.000000e+00
## [110,] 1.000000e+00
## [111,] 1.000000e+00
## [112,] 1.000000e+00
## [113,] 1.000000e+00
## [114,] 1.000000e+00
## [115,] 1.000000e+00
## [116,] 1.000000e+00
## [117,] 1.000000e+00
## [118,] 1.000000e+00
## [119,] 1.000000e+00
## [120,] 1.000000e+00
## [121,] 1.000000e+00
## [122,] 1.000000e+00
## [123,] 1.000000e+00
## [124,] 1.000000e+00
## [125,] 1.000000e+00
## [126,] 1.000000e+00
## [127,] 1.000000e+00
## [128,] 1.000000e+00
## [129,] 1.000000e+00
## [130,] 1.000000e+00
## [131,] 1.000000e+00
## [132,] 1.000000e+00
## [133,] 1.000000e+00
## [134,] 1.000000e+00
## [135,] 1.000000e+00
## [136,] 1.000000e+00
## [137,] 1.000000e+00
## [138,] 1.000000e+00
## [139,] 1.000000e+00
## [140,] 1.000000e+00
## [141,] 1.000000e+00
## [142,] 1.000000e+00
## [143,] 1.000000e+00
## [144,] 1.000000e+00
## [145,] 1.000000e+00
## [146,] 1.000000e+00
## [147,] 1.000000e+00
## [148,] 1.000000e+00
## [149,] 1.000000e+00
## [150,] 1.000000e+00
```