HW4_dbagchi2

Diptendra Nath Bagchi (dbagchi2@illinois.edu)

19 March, 2020

Question 1 [40 Points] K-Means Clustering

Let's consider coding our own K-means algorithm. Perform the following:

a. [15 Points] Write your own code of k-means that iterates between two steps, and stop when the cluster membership does not change. Use the l2 norm as the distance metric. Write your algorithm as efficiently as possible. Avoiding extensive use of for-loop could help.

- updating the cluster means given the cluster membership
- updating the cluster membership based on the cluster means

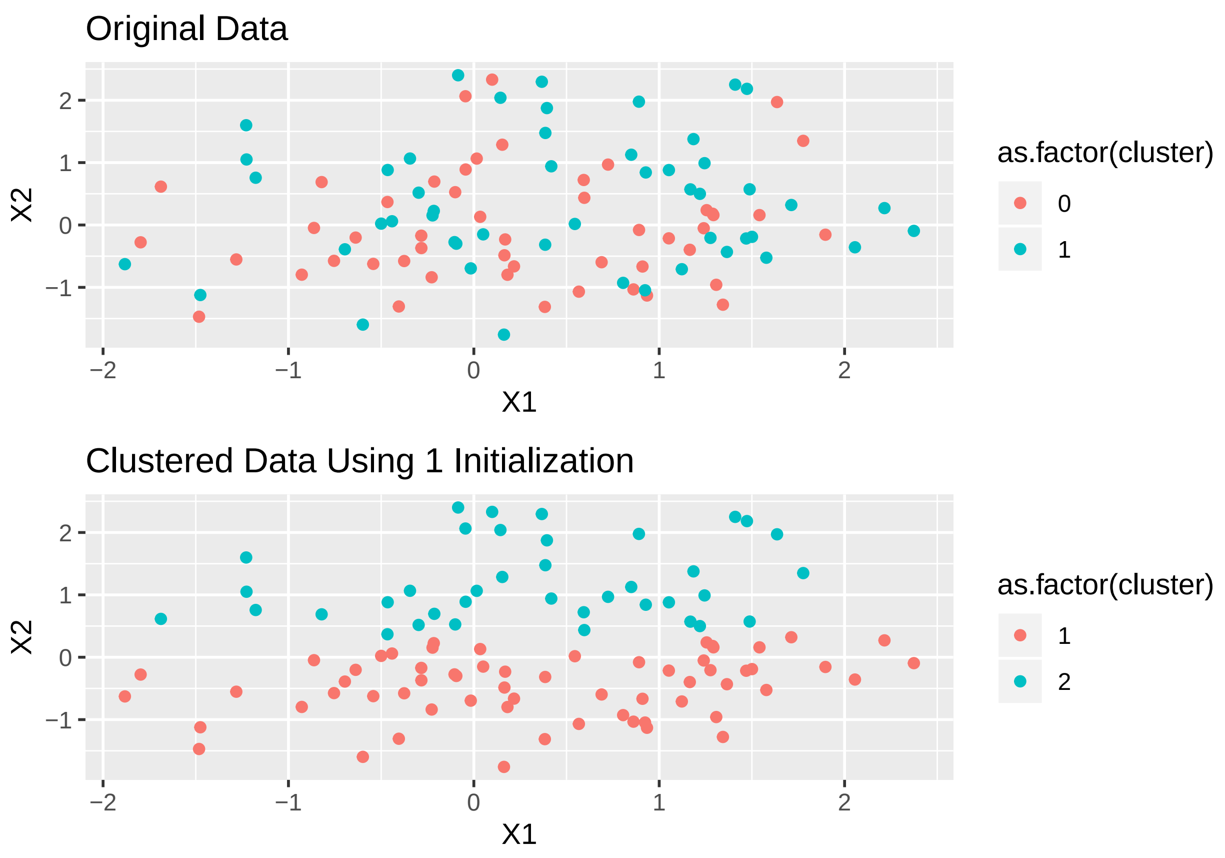
```r
distance_calc = function(x, centroids) {
  apply(centroids, MARGIN = 1, function(y) sqrt(sum((y - x) ^ 2)))
}

distance_withinss = function(x, centroids) {
  apply(centroids, MARGIN = 1, function(y) (sum((y - x) ^ 2)))
}

centroid_avg_calc = function(x, x_with_clusters)
  return(apply(subset(x_with_clusters, cluster == x)[, -ncol(x_with_clusters)], 2, mean))

my_kmeans = function(x, k = 3, max_iter = 5000) {
  # Initialization of the centroids
  centroids = as.data.frame(x[sample.int(nrow(x), k), ])
  old_cluster = rep(1, nrow(x))
  new_cluster = rep(1, nrow(x))

  for (i in 1 : max_iter) {
    distance_from_centroid = data.frame(t
                                        (apply(
                                          x,
                                          MARGIN = 1,
                                          FUN = distance_calc,
                                          centroids = centroids
                                        )))
    # Step 1: Assign the cluster
    # Assigning the cluster to the each data point
    cluster = apply(distance_from_centroid, 1, FUN = function(x) which.min(x))
    # Append the cluster into the x data frame
    x_with_clusters = cbind.data.frame(x, cluster)
    new_cluster = x_with_clusters$cluster
    # Step 2: Update the Centroids
    # Updating the centroids
    centroids = data.frame(t
                           (sapply(
                             seq(1, k, 1),
                             FUN = centroid_avg_calc,
                             x_with_clusters = x_with_clusters
                             )
                             )
                           )

    if(identical(old_cluster, new_cluster) == TRUE){
      break;
    } else { old_cluster = new_cluster }
  }
  # To calculate the within sum of squares
  distance_from_centroid_withinss = data.frame(t
                                          (apply(
                                            x,
                                            MARGIN = 1,
                                            FUN = distance_withinss,
                                            centroids = centroids
                                          )))
  all_withinss = data.frame(t(apply(distance_from_centroid_withinss,1,
                   FUN = function(x) c(x[which.min(x)], which.min(x)
                                         ))))
  colnames(all_withinss) = c("withinss", "cluster")
  withinss = data.frame(t(sapply(seq(1, k, 1),
                    FUN = function(x, all_withinss) {
                      temp = subset(all_withinss, cluster == x)
                      return(c(x, sum(temp$withinss)))
                      },
                    all_withinss = all_withinss
                    )))
  colnames(withinss) = c("cluster", "withinss")
  return(list("data_with_clusters" = x_with_clusters,
              "centroids" = centroids,
              "withinss" = withinss,
              "tot_withinss" = sum(withinss$withinss)
              )
         )}
```

b. [5 points] Test your code with this small-sized example by performing the following. Make sure that you save the random seed.

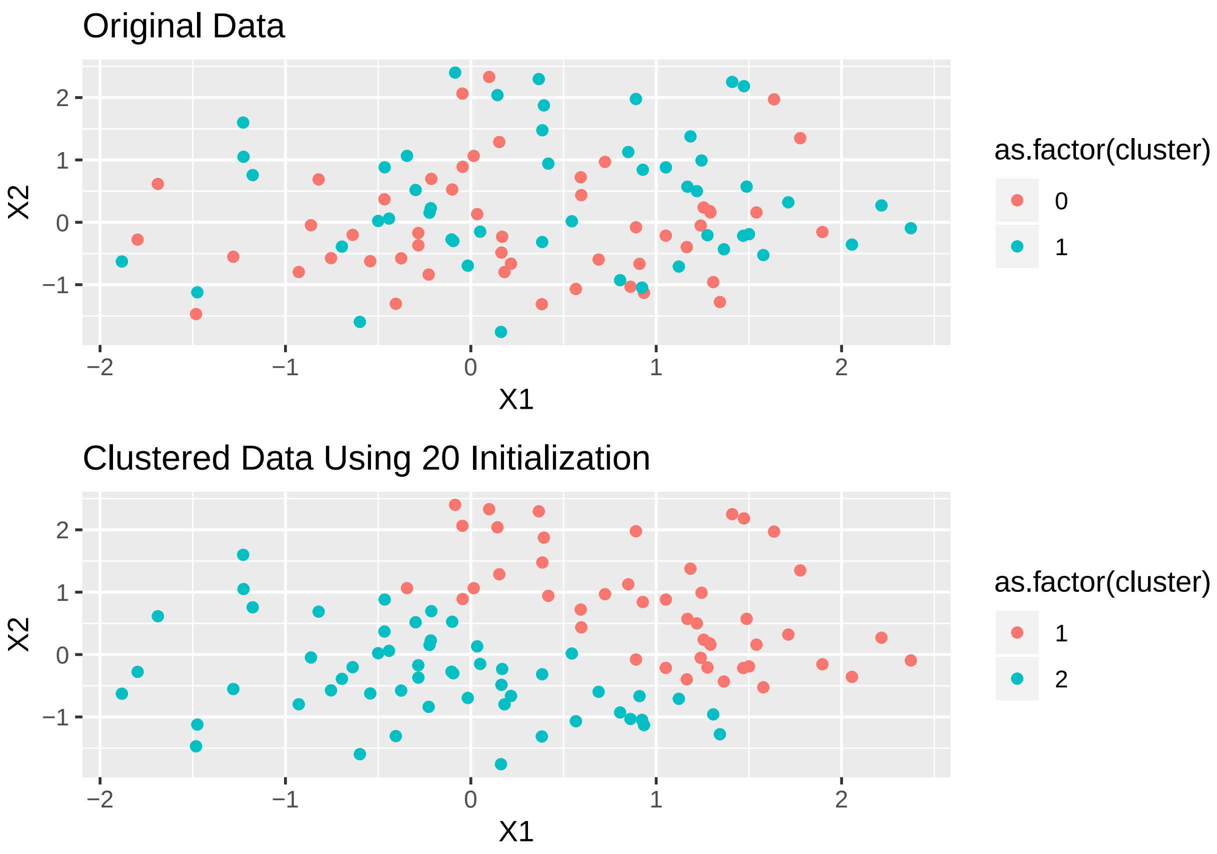
- Run your algorithm on the data using ONE random initialization, output the within-cluster distance, and compare the resulting clusters with the underlying truth.
- Run your algorithm on the data using 20 random initialization, output the within-cluster distance, and compare the resulting clusters with the underlying truth.
- Do you observe any difference? Note that the result may vary depending on the random seed.

```r
set.seed(4)
x = rbind(matrix(rnorm(100), 50, 2), matrix(rnorm(100, mean = 0.5), 50, 2))
y = c(rep(0, 50), rep(1, 50))
x_y = cbind.data.frame(x, y)
colnames(x_y) = c("x1", "x2", "cluster")
```

```r
random_init_1 = my_kmeans(x = x, k = 2, max_iter = 5000)
tot_withinss_01 = random_init_1$tot_withinss
p1 = ggplot(x_y, aes(x = x1, y = x2, color = as.factor(cluster))) +
  geom_point() +
  labs(x = "X1", y = "X2") +
  ggtitle("Original Data")
colnames(random_init_1$data_with_clusters) = c("x1", "x2", "cluster")
colnames(x_y) = c("x1", "x2", "cluster")
p2 = ggplot(random_init_1$data_with_clusters,
            aes(x = x1, y = x2, color = as.factor(cluster))) +
  geom_point() +
  labs(x = "X1", y = "X2") +
  ggtitle("Clustered Data Using 1 Initialization")
grid.arrange(p1, p2)
```



```r
initialization = 1:20
random_init_20 = lapply(initialization, my_kmeans, x = x, k = 2)
least_withinss = sapply(initialization,
                        function(x,
                                 random_init_20) random_init_20[[x]]$tot_withinss,
                        random_init_20 = random_init_20)
tot_withinss_20 = least_withinss[which.min(least_withinss)]
best_random_k = random_init_20[[which.min(least_withinss)]]
colnames(best_random_k$data_with_clusters) = c("x1", "x2", "cluster")
p3 = ggplot(best_random_k$data_with_clusters,
            aes(x = x1, y = x2, color = as.factor(cluster))) +
  geom_point() +
  labs(x = "X1", y = "X2") +
  ggtitle("Clustered Data Using 20 Initialization")
grid.arrange(p1, p3)
```



```
# Using the already existing kmeans algorithm to verify
clusters_using_kmeans = kmeans(x, 2)
```

```
df = tibble("Within-SS (1)" = tot_withinss_01,
            "Within-SS (20)" = tot_withinss_20,
            "Within-SS (Kmeans)" = clusters_using_kmeans$tot.withinss)
kable(df, digits = 4, row.names = FALSE, align = "c") %>%
  kable_styling("striped", full_width = FALSE) %>%
  add_header_above(c("Total Within-SS for Different Initialization" = 3))
```

Total Within-SS for Different Initialization		
Within-SS (1)	Within-SS (20)	Within-SS (Kmeans)
121.5713	116.2215	116.2215

We see there is a difference in the Total Within Sum of Squares with **one** initialization and **20** initalizations. The Total Within SS are 121.5713023 and 116.2214995. We clearly see that the **20** initilizations gives a less Within-SS compared to the **one** initilization. I also checked this using the **K-Means** algorithm already implemented in R which gives us the same result as **20** initilizations.

This is one of the problems in K-Means, where if the initiliazation is not done correctly, the values cluster generation might not be optimal. Hence, it is important for us to do multiple random initilizations to make sure that we get the best clusters for the data.

c. [10 Points] Load the zip.train (handwritten digit recognition) data from the ElemStatLearn package and the goal is to identify clusters of digits based on only the pixels. Apply your algorithm on the handwritten digit data with k=15, with ten random initialization. Then, compare your cluster membership to the true digits.

- For each of your identified clusters, which is the representative (most dominating) digit?
- How well does your clustering result recover the truth? Which digits seem to get mixed up (if any) the most? Again, this may vary depending on the random seed. Hence, make sure that you save the random seed.

```r
trn = as.data.frame(zip.train)
initialization = 1:10
image_k_means = lapply(initialization, my_kmeans, x = trn[, c(-1)], k = 15)
least_withinss = sapply(initialization,
                        function(x,
                                 image_k_means) image_k_means[[x]]$tot_withinss,
                        image_k_means = image_k_means)
best_init = which.min(least_withinss)
final_cluster = image_k_means[[best_init]]
```

```r
y = trn[, 1]
df = cbind.data.frame(y,
                       "cluster"= final_cluster$data_with_clusters[,ncol(final_cluster$data_with_clusters
frequent_number = function(x, df){
  temp = subset(df, cluster == x)
  all_nos = table(temp$y)
  most_freq = as.integer(names(all_nos[which.max(all_nos)]))
  return(c(x, most_freq))
}
dominating_digit = data.frame(t(
  sapply(1:15, frequent_number, df = df)
))
colnames(dominating_digit) = c("Cluster Number", "Dominating Digit")
kable(dominating_digit, digits = 2, row.names = FALSE) %>%
  kable_styling("striped", full_width = FALSE) %>%
  add_header_above(c("Dominating Digit in Each Cluster" = 2)) %>%
  column_spec(column = 1, bold = TRUE)
```

Dominating Digit in Each Cluster	
Cluster Number	Dominating Digit
1	2
2	1
3	5
4	7
5	0
6	9
7	9
8	6
9	4
10	0
11	2
12	3
13	8
14	3
15	6

```r
mixed_up_numbers = as.data.frame.matrix(table(df))
mixed_up_numbers = cbind("Numbers" = row.names(mixed_up_numbers), mixed_up_numbers)
kable(mixed_up_numbers, digits = 2, row.names = FALSE) %>%
  kable_styling("striped", full_width = FALSE, latex_options = "scale_down") %>%
  add_header_above(c("Number and Cluster Frequency" = 16)) %>%
  column_spec(column = 1, bold = TRUE)
```

Number and Cluster Frequency															
Numbers	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	483	0	3	91	0	427	18	13	5	3	150
1	0	1000	1	0	0	0	1	0	0	0	0	0	3	0	0
2	336	3	0	12	0	1	18	2	7	1	275	5	30	29	12
3	6	0	13	2	0	0	11	0	2	2	3	159	27	433	0
4	0	34	0	1	0	86	181	4	290	1	49	0	4	0	2
5	1	0	308	1	3	7	8	40	10	2	20	129	9	11	7
6	0	6	0	0	3	0	5	439	1	3	18	1	1	0	187
7	1	2	0	464	0	36	114	0	23	0	3	0	2	0	0
8	4	6	14	1	1	4	52	3	8	4	10	11	400	24	0
9	0	3	1	19	0	305	205	0	105	0	0	0	4	2	0

From the table above, the digits that seem to get mixed up the most are (4, 7 and 9) and (4, 9). ## d. [10 Points] If you are asked to use this clustering result as a predictive model and suggest (predict) a label for each of the testing data zip.test, what would you do? Properly describe your prediction procedure and carry out the analysis. What is the prediction accuracy of your model? Note that you can use the training data labels for prediction.

The procedure to use the clustering technique for prediction is the following:

1. Pick the values of the independent column values for each testing data point.
2. Calculate the distance of the above testing data point to all the cluster centroids.
3. Save the distances into a data frame.
4. Pick the cluster centroid that has the minimum distance from the testing point.
5. Using the cluster point, predict the most frequent value of the digit for the testing point.
6. Repeat Steps 1 through 5 for all the data points in the testing data set.

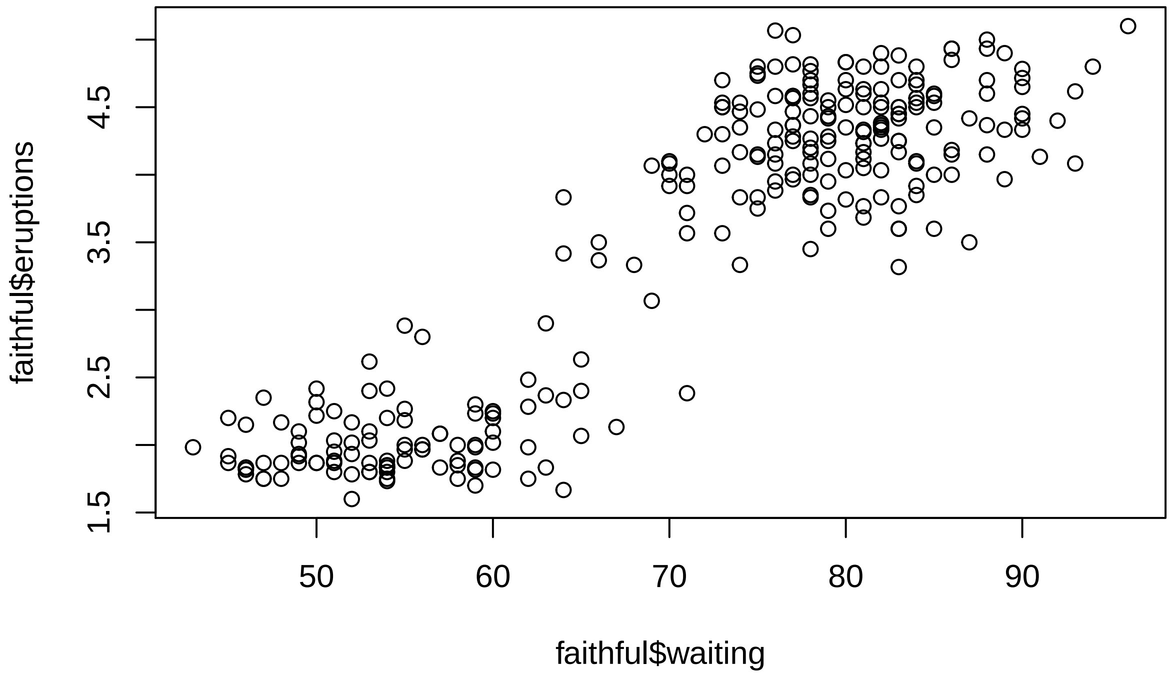
```r
tst = as.data.frame(zip.test)
y_tst = tst[, 1]
x_tst = tst[, c(-1)]
prediction_using_kmeans = function(x, centroids, lookup_table) {
  temp = apply(centroids, 1, function(y, x) sqrt(sum((y - x) ^ 2)), x = x)
  prediction = lookup_table[which.min(temp),"Dominating Digit"]
}
predicted_value = apply(X = x_tst, MARGIN = 1,
                        prediction_using_kmeans,
                        centroids = final_cluster$centroids,
                        lookup_table = dominating_digit)
accuracy = mean(predicted_value == y_tst)
```

The prediction accuracy for the the `zip.tst` is 0.7468859

Question 3 [40 Points] Two-dimensional Kernel Regression

For this question, you need to write your own functions. The goal is to model and predict the eruption duration using the waiting time in between eruptions of the Old Faithful Geyser. You can download the data from [kaggle: Old Faithful] or our course website.

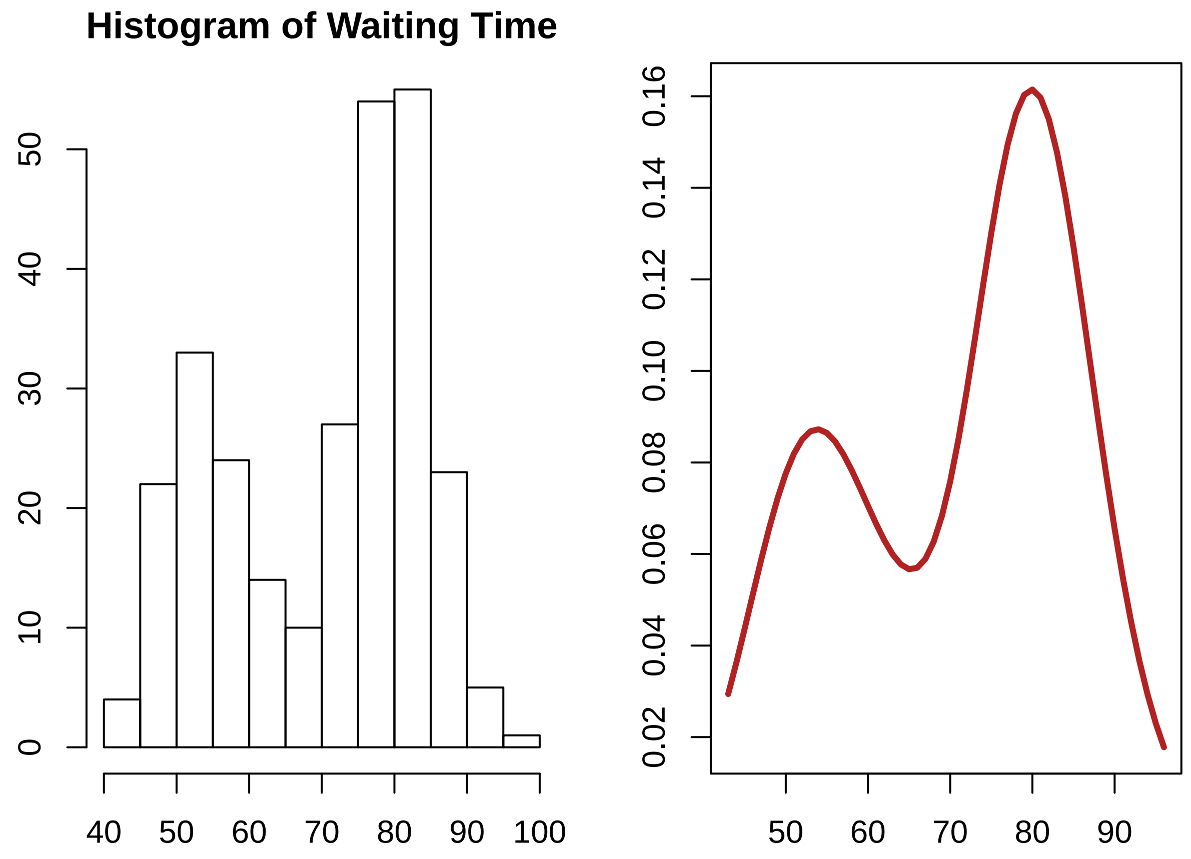
```r
faithful = read.csv("data/faithful.csv", stringsAsFactors = FALSE)
plot(faithful$waiting, faithful$eruptions)
```



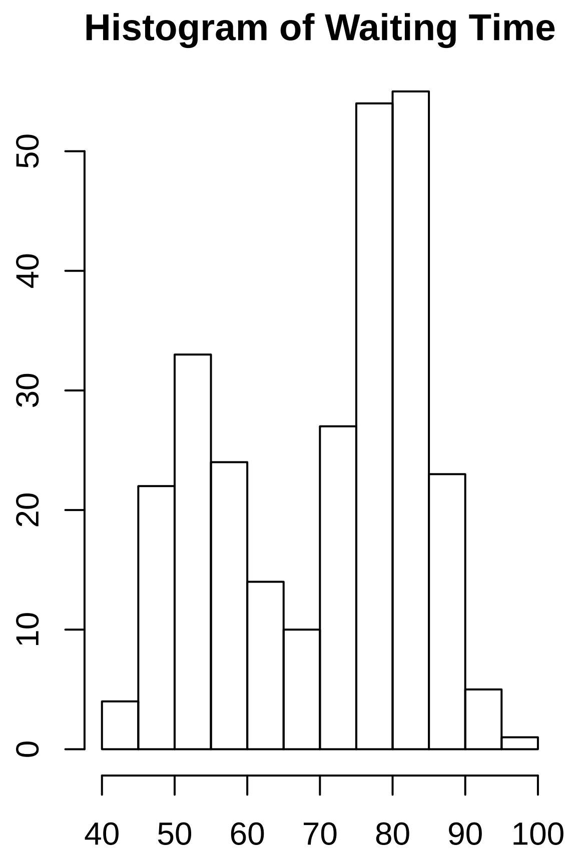
a) Use a kernel density estimator to estimate the density of the eruption waiting time. Use a bandwidth with the theoretical optimal rate (the Silverman rule-of-thumb formula). Plot the estimated density function. You should consider two kernel functions:

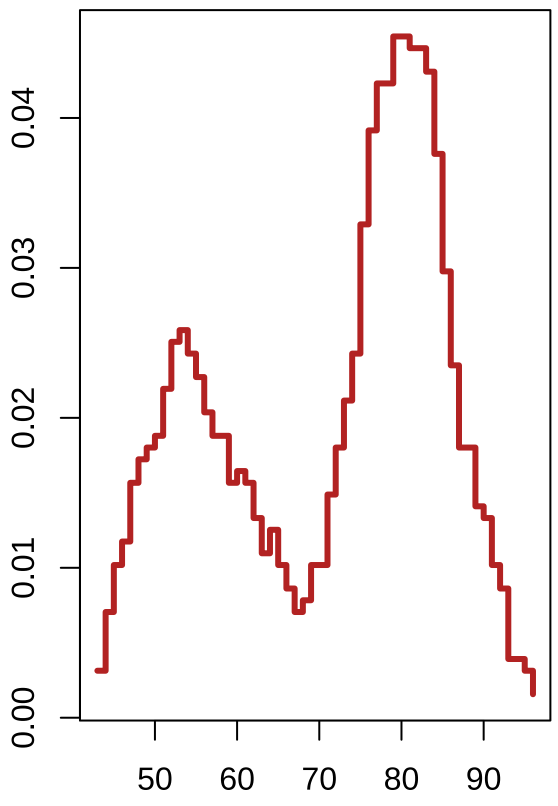
- Gaussian Kernel
- Uniform kernel

```r
# Silverman Rule-of-Thumb
lambda_hat = sd(faithful$waiting) * (4 / 3 / length(faithful$waiting)) ^ (1 / 5)
# Creating a grid to plot the density
x_grid = seq(min(faithful$waiting), max(faithful$waiting), 1)
# Gaussian kernel for density estimation
gaussian_kernel = function(x, obs, h) sum(exp(-0.5 * ((x - obs) / h) ^ 2) / sqrt(2 * pi))
# Plotting the density estimation for the Waiting time
par(mfrow = c(1, 2), mar=rep(2, 4))
hist(faithful$waiting, main = "Histogram of Waiting Time")
plot(x_grid, sapply(x_grid,
                   FUN = gaussian_kernel, obs = faithful$waiting,
                   h = lambda_hat) / length(faithful$waiting),
     type = "l",
     xlab = "Waiting Time",
     ylab = "Estimated Density",
     col = "firebrick",
     lwd = 3)
```



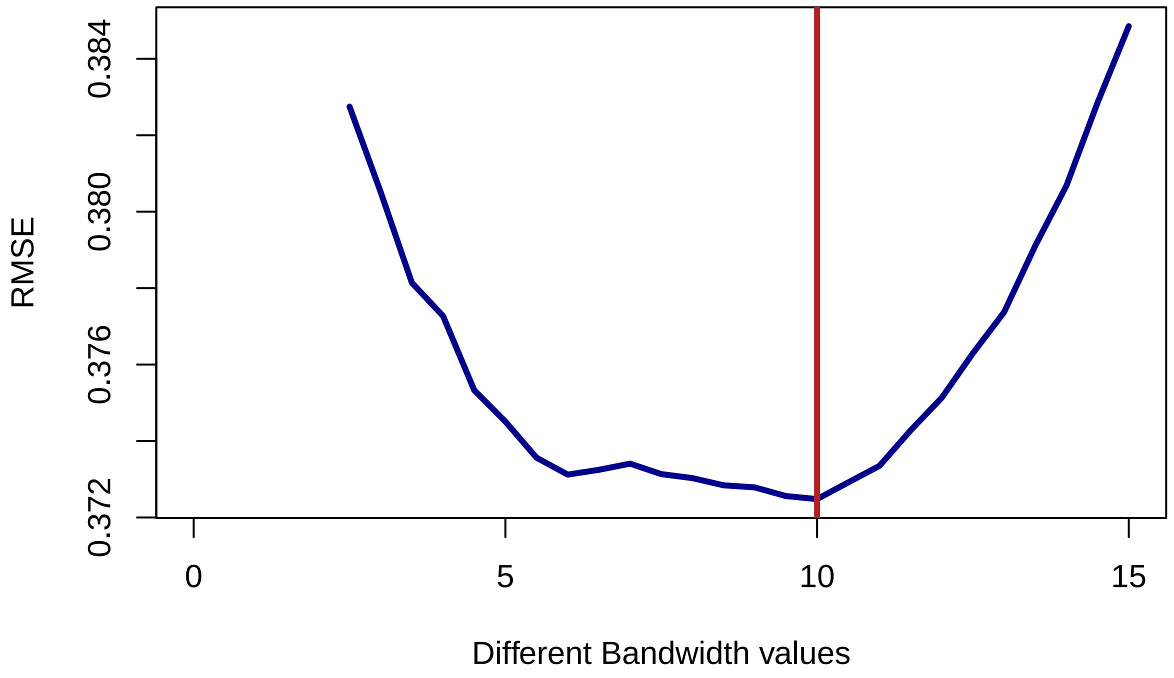
```r
# Uniform Kernel Estimation
uniform_kernel = FUN = function(x, obs, h)
  sum( ((x-h/2) <= obs) * ((x+h/2) > obs))/h/length(obs)
par(mfrow = c(1, 2), mar=rep(2, 4))
hist(faithful$waiting, main = "Histogram of Waiting Time")
plot(x_grid, sapply(x_grid, FUN = uniform_kernel ,
                   obs = faithful$waiting, h = lambda_hat),
     type = "s",
     col = "firebrick",
    lwd = 3)
```





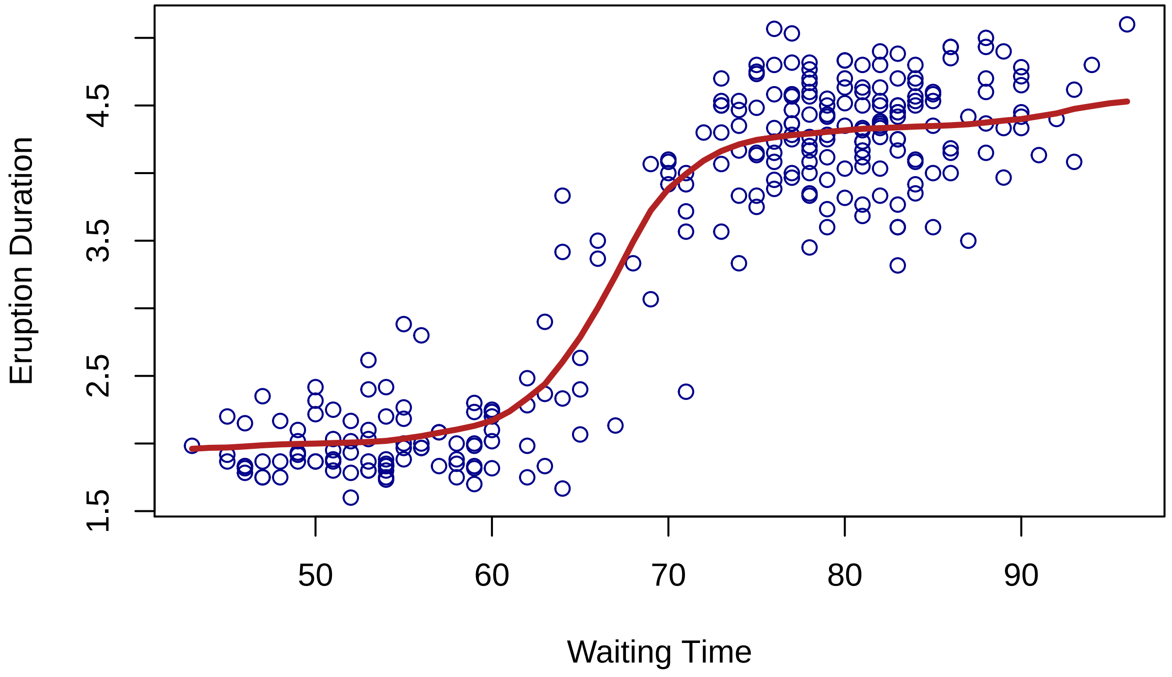
b. [15 points] Use a Nadaraya-Watson kernel regression estimator to model the conditional mean eruption duration using the waiting time. Use a 10-fold cross-validation to select the optimal bandwidth. You should use the Epanechnikov kernel for this task. Report the optimal bandwidth and plot the fitted regression line using the optimal bandwidth.

```
rmse_calc = function(actual, predicted) {
  sqrt(mean((actual - predicted) ^ 2))
}
kernel_regression = function(x, obs, h, y) {
  k = ifelse(abs((x - obs) / h) <= 1, (3/4) * (1 - (((x - obs) / h) ^ 2)), 0)
  k_sum = sum(k)
  weight = k / k_sum
  y_k = sum(weight * y)
  y_k
}
nfold = 10
index_fold = sample(rep(1:nfold, length.out = nrow(faithful)))
all_k = c(seq(from = 0, to = 15, by = 0.5))
accu_matrix = rep(0, length(all_k))
for (k in 1:length(all_k)) {
  temp_rmse = rep(0, nfold)
  for (l in 1:nfold) {
    train = faithful[index_fold != l, ]
    test = faithful[index_fold == l, ]
    estimate_x = test$waiting
    estimated_df = (sapply(X = test$waiting,
                           FUN = kernel_regression,
                           obs = train$waiting,
                           h = all_k[k],
                           y = train$eruptions)
                    )
    temp_rmse[l] = rmse_calc(actual = test$eruptions, predicted = estimated_df)
  }
  accu_matrix[k] = mean(temp_rmse)
}
# Optimal bandwidth for the kernal regression
optimal_bandwidth = all_k[which.min(accu_matrix)]
# Plotting the fir for all the possible bandwidth
plot(all_k, accu_matrix, type = "l",
     xlab = "Different Bandwidth values",
     ylab = "RMSE",
     col = "darkblue",
     lwd = 3)
abline(v = optimal_bandwidth, col = "firebrick", lwd = 3)
```



The optimal value of the Bandwidth is 10

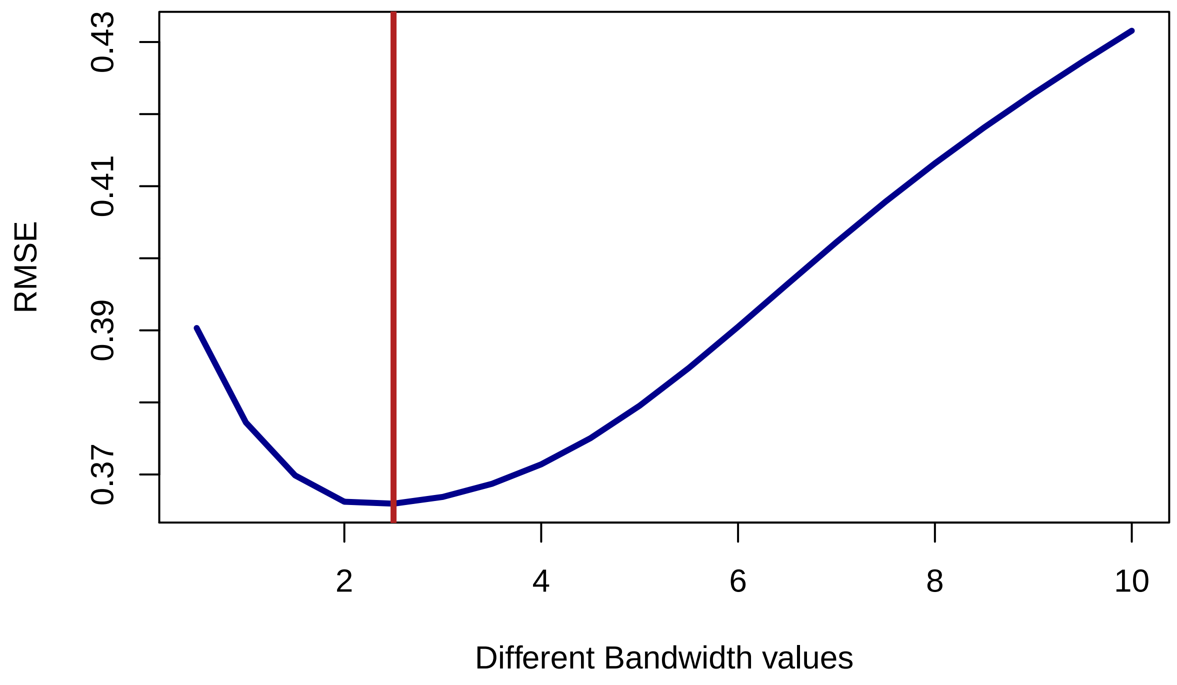
```
# For plotting the data for the optimal bandwidth
estimated_df = sapply(X = seq(min(faithful$waiting), max(faithful$waiting), 1),
                           FUN = kernel_regression,
                           obs = faithful$waiting,
                           h = optimal_bandwidth,
                           y = faithful$eruptions
                )
plot(faithful$waiting,faithful$eruptions,
     xlab = "Waiting Time", ylab = "Eruption Duration",
     col = 'darkblue')
lines(seq(min(faithful$waiting), max(faithful$waiting), 1),
      estimated_df, col = 'firebrick', lwd = 3)
```



c. [15 points] Fit a local linear regression to model the conditional mean eruption duration using the waiting time. Use a 10-fold cross-validation to select the optimal bandwidth. You should use the Gaussian kernel for this task. Report the optimal bandwidth and plot the fitted regression line using the optimal bandwidth.

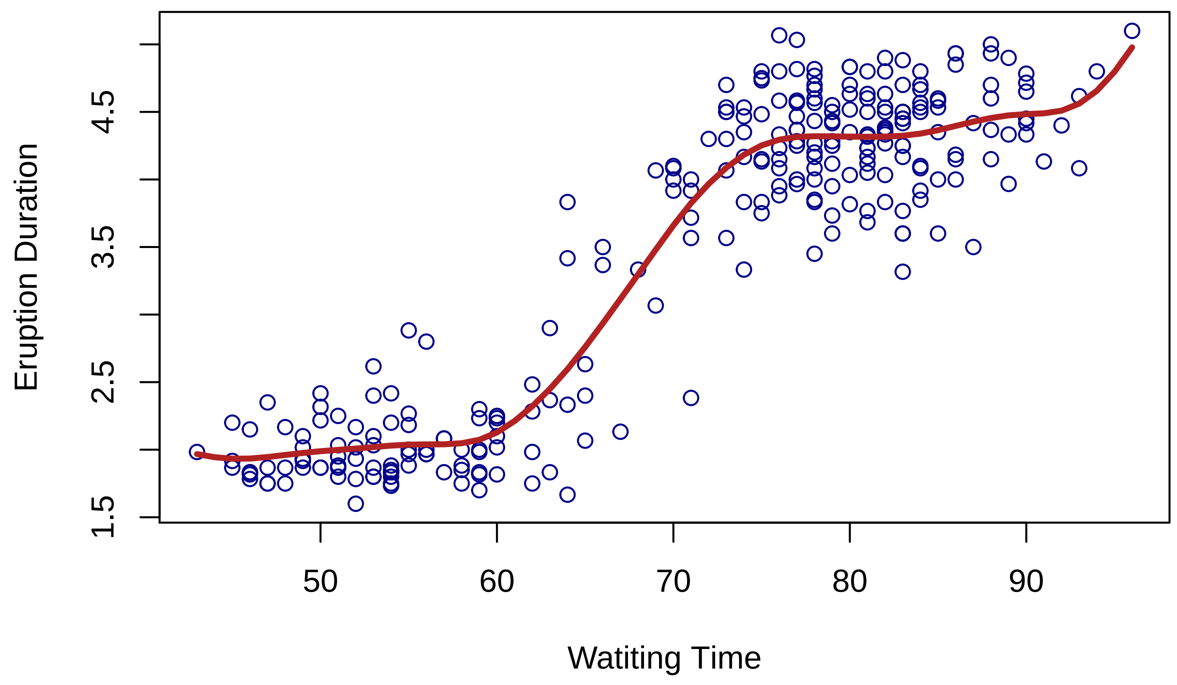
```
kernel_linear_regression = function(x, obs, h, y) {
  k = exp(-0.5 * ((x - obs) / h) ^ 2) / sqrt(2 * pi)
  w = diag(k)
  X = cbind(rep(1, length(obs)), obs)
  beta = solve(t(X) %*% w %*% X) %*% t(X) %*% w %*% y
  beta_0 = beta[1, 1]
  beta_1 = beta[2, 1]
  y_k = beta_0 + (beta_1 * x)
}
nfold = 10
index_fold = sample(rep(1:nfold, length.out = nrow(faithful)))
all_k = c(seq(from = 0.5, to = 10, by = 0.5))
accu_matrix = rep(0, length(all_k))
for (k in 1:length(all_k)) {
  temp_rmse = rep(0, nfold)
  for (l in 1:nfold) {
    train = faithful[index_fold != l, ]
    test = faithful[index_fold == l, ]
    estimate_x = test$waiting
    estimated_df = sapply(X = test$waiting,
```

```r
                                    FUN = kernel_linear_regression,
                                    obs = train$waiting,
                                    h = all_k[k],
                                    y = train$eruptions)
    temp_rmse[l] = rmse_calc(actual = test$eruptions, predicted = estimated_df)
  }
  accu_matrix[k] = mean(temp_rmse)
}
# Optimal bandwidth for the kernal regression
optimal_bandwidth = all_k[which.min(accu_matrix)]
# Plotting the fir for all the possible bandwidth
plot(all_k, accu_matrix, type = "l",
     xlab = "Different Bandwidth values",
     ylab = "RMSE",
     col = "darkblue",
     lwd = 3)
abline(v = optimal_bandwidth, col = "firebrick", lwd = 3)
```

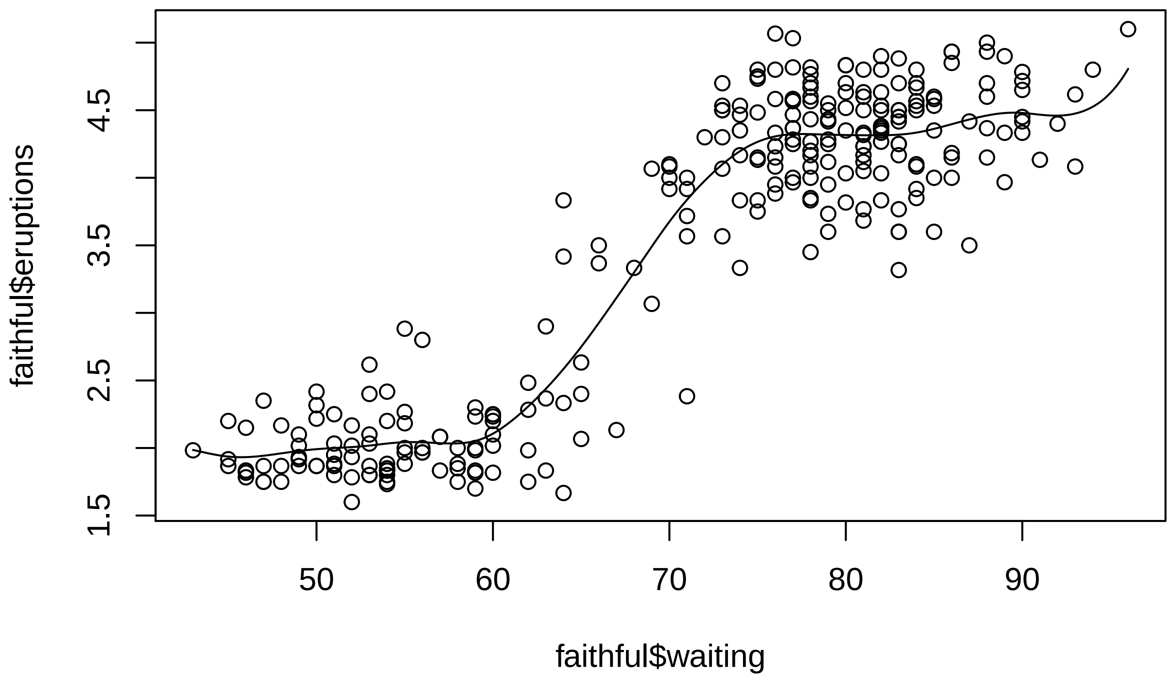


The optimal value of the Bandwidth is 2.5

```r
# For plotting the data for the optimal bandwidth
estimated_df = sapply(X = seq(min(faithful$waiting), max(faithful$waiting), 1),
                                    FUN = kernel_linear_regression,
                                    obs = faithful$waiting,
                                    h = optimal_bandwidth,
                                    y = faithful$eruptions
                      )
plot(faithful$waiting,faithful$eruptions,
     xlab = "Watiting Time",
     ylab = "Eruption Duration",
     col = 'darkblue')
lines(seq(min(faithful$waiting), max(faithful$waiting), 1),
      estimated_df, col = 'firebrick', lwd = 3)
```



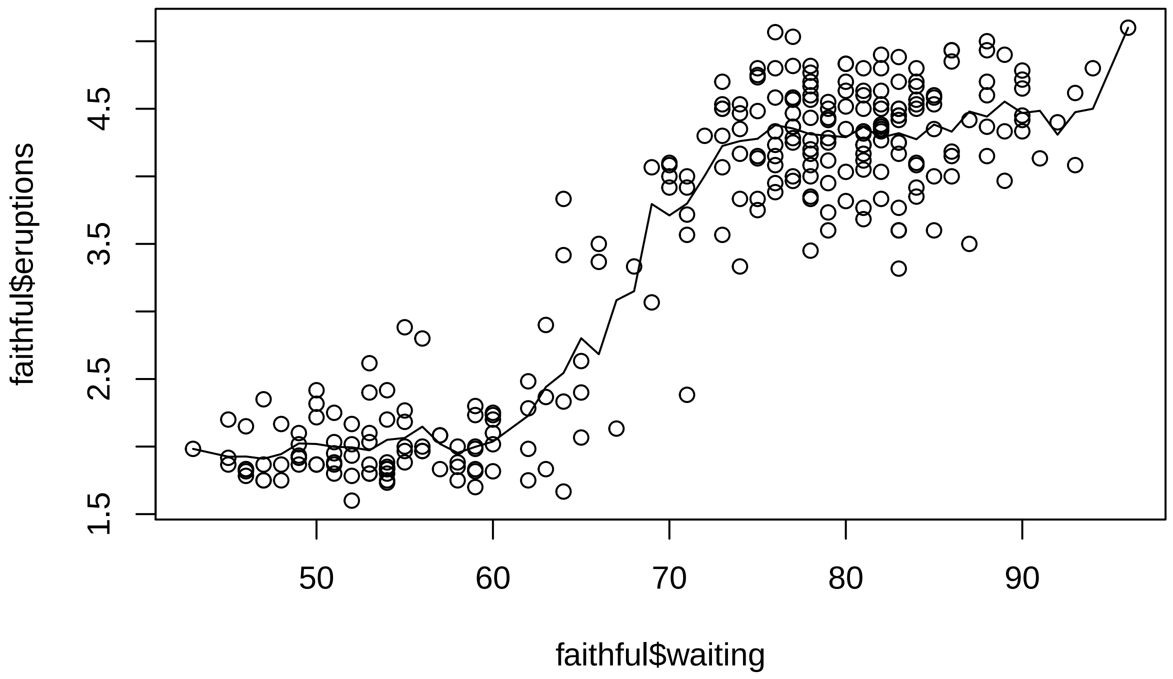
```
h = dpill(faithful$waiting, faithful$eruptions)
locpoly_fit = locpoly(faithful$waiting, faithful$eruptions, bandwidth = h)
plot(faithful$waiting, faithful$eruptions)
lines(locpoly_fit)
```



```
h
```

```
## [1] 2.169462
```

```r
h = 2.5
ksmooth_fit = ksmooth(faithful$waiting, faithful$eruptions,
                      bandwidth = h, kernel = "box", x.points = faithful$waiting)
plot(faithful$waiting, faithful$eruptions)
lines(ksmooth_fit)
```



```r
h
```

```
## [1] 2.5
```