# Homework 2

*Richard Albright*
*ISYE6501*
*Spring 2018*

## Question 3.1

Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 2.2, use the ksvm or kknn function to find a good classifier:

### (a) using cross-validation (do this for the k-nearest-neighbors model; SVM is optional)

Read in the CSV

```
data <-
  read.table(
    "/Users/ralbright/Documents/ISYE6501/week2/credit_card_data-headers.txt",
    header=TRUE,
    sep="\t"
  )
```

Lets check if the data loaded by checking the head and tail.

Head:

```
table <- xtable(head(data))
print(table, type='latex', comment=FALSE)
```

| | A1 | A2 | A3 | A8 | A9 | A10 | A11 | A12 | A14 | A15 | R1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 30.83 | 0.00 | 1.25 | 1 | 0 | 1 | 1 | 202 | 0 | 1 |
| 2 | 0 | 58.67 | 4.46 | 3.04 | 1 | 0 | 6 | 1 | 43 | 560 | 1 |
| 3 | 0 | 24.50 | 0.50 | 1.50 | 1 | 1 | 0 | 1 | 280 | 824 | 1 |
| 4 | 1 | 27.83 | 1.54 | 3.75 | 1 | 0 | 5 | 0 | 100 | 3 | 1 |
| 5 | 1 | 20.17 | 5.62 | 1.71 | 1 | 1 | 0 | 1 | 120 | 0 | 1 |
| 6 | 1 | 32.08 | 4.00 | 2.50 | 1 | 1 | 0 | 0 | 360 | 0 | 1 |

Tail:

```
table <- xtable(tail(data))
print(table, type='latex', comment=FALSE)
```

| | A1 | A2 | A3 | A8 | A9 | A10 | A11 | A12 | A14 | A15 | R1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 649 | 1 | 40.58 | 3.29 | 3.50 | 0 | 1 | 0 | 0 | 400 | 0 | 0 |
| 650 | 1 | 21.08 | 10.09 | 1.25 | 0 | 1 | 0 | 1 | 260 | 0 | 0 |
| 651 | 0 | 22.67 | 0.75 | 2.00 | 0 | 0 | 2 | 0 | 200 | 394 | 0 |
| 652 | 0 | 25.25 | 13.50 | 2.00 | 0 | 0 | 1 | 0 | 200 | 1 | 0 |
| 653 | 1 | 17.92 | 0.20 | 0.04 | 0 | 1 | 0 | 1 | 280 | 750 | 0 |
| 654 | 1 | 35.00 | 3.38 | 8.29 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

## KNN Cross Validation

Since we will be using K-Fold cross validation, we do not need separate training and validation sets, as the folding in the cv.kknn function will do both by splitting our training set automatically into its relevant training and validation sets based on the k value provided. Therefore, our training/validation set is 80% of the data, and our testing set is 20% of the data. We will separate out our training and testing sets using the R sample() function.

```r
row_count = nrow(data)
sample_rows <- sample(seq(1:row_count))
train_splice <- as.integer(row_count*0.8)
train_rows = sample_rows[1:train_splice-1]
test_rows = sample_rows[train_splice:row_count]

train_set = data[train_rows,]
test_set = data[test_rows,]
```

Our resulting sets contain the following amount of observations - Training: 522, Testing: 132.

The cv.kknn function performs K-Fold Validation for K Nearest Neighbor. We want to run this function on our training set for a given k for KNN. A kcv value of 10 splits the data into 10 partitions, with 9 (90%) used for training, and 1 (10%) used for validation. The function train_kknn_fold takes a specified k value to test against our training set and returns the resulting accuracy.

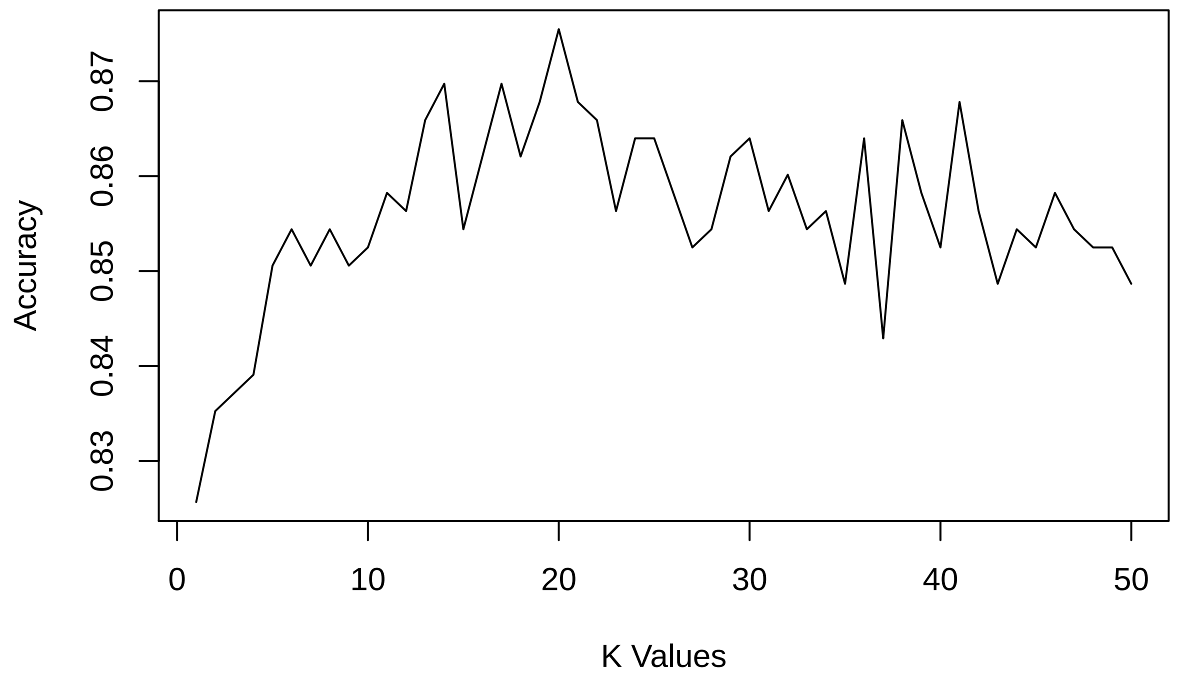
```r
train_kknn_fold <- function(k_val) {
  model.cv <- cv.kknn(R1~., train_set, kcv=10, k=k_val, scale=TRUE)
  fitted_model <- as.matrix(model.cv[[1]])
  predictions  <- as.matrix(lapply(fitted_model[, 2], round))
  accuracy <- sum(predictions == train_set[, c('R1')]) / nrow(train_set)
  return(accuracy)
}
```

Now that we have our K-Fold cross validation training function. Let's loop through k from 1 to 50 to find the best k value for our training set. We will store the accuracy of each training set in an accuracies array. We will then find the maximum accuracy and look up the index value of that maximum accuracy in the accuracies array, this is our best k value for our KNN model.

```r
ksteps = seq(1,50)
kknn_accuracies = matrix(1:length(ksteps), nrow = length(ksteps), ncol = 1)
for(k in ksteps) {
  kknn_accuracies[k] = train_kknn_fold(k)
}
```

Below is the resulting plot of our K Value accuracies.

```r
plot(kknn_accuracies, type='l', xlab='K Values', ylab='Accuracy')
```



```
max_accuracy = max(kknn_accuracies)
max_accuracy
```

```
## [1] 0.8754789
```

```
max_ks = which.max(kknn_accuracies)
max_ks
```

```
## [1] 20
```

The best K value from our training set is 20 with an accuracy of 0.8754789.

## Final KNN Model

Now that we have our best k value. Let's retrain our KNN model over the entire set using the training and testing sets.

```
model <- kknn(R1 ~., train_set, test_set, k = max_ks, scale = TRUE)
predictions <- as.matrix(lapply(model$fit, round))
kknn_accuracy = sum(predictions == test_set[, c('R1')]) / nrow(test_set)
```

Our KNN model's accuracy is 0.8106061.

## SVM Cross Validation

First we must convert our training set to a matrix and create trainX and trainY variables for the predictor columns and the response column.

```
train_matrix = as.matrix(train_set)
trainX <- train_matrix[,1:10]
trainY <- train_matrix[,11]
test_matrix = as.matrix(test_set)
```

```
testX = test_matrix[,1:10]
testY = test_matrix[,11]
```

The ksvm function also has the ability to do K-Fold cross validation through the use of the cross input argument. Using cross=5 partitions our data into 5 parts, 4 for training, 1 for validation. We will be testing our model using the linear kernel. First train the model using the training set, then test the predictions vs the testing set.

```
train_ksvm <- function(c_val, cross_val) {
  model <- ksvm(trainX, trainY, type="C-svc", kernel="vanilladot",
        C=c_val, cross=cross_val, scaled=TRUE)
  predictions <- predict(model,testX)
  accuracy = sum(predictions == testY) / nrow(test_matrix)
  return(accuracy)
}
```

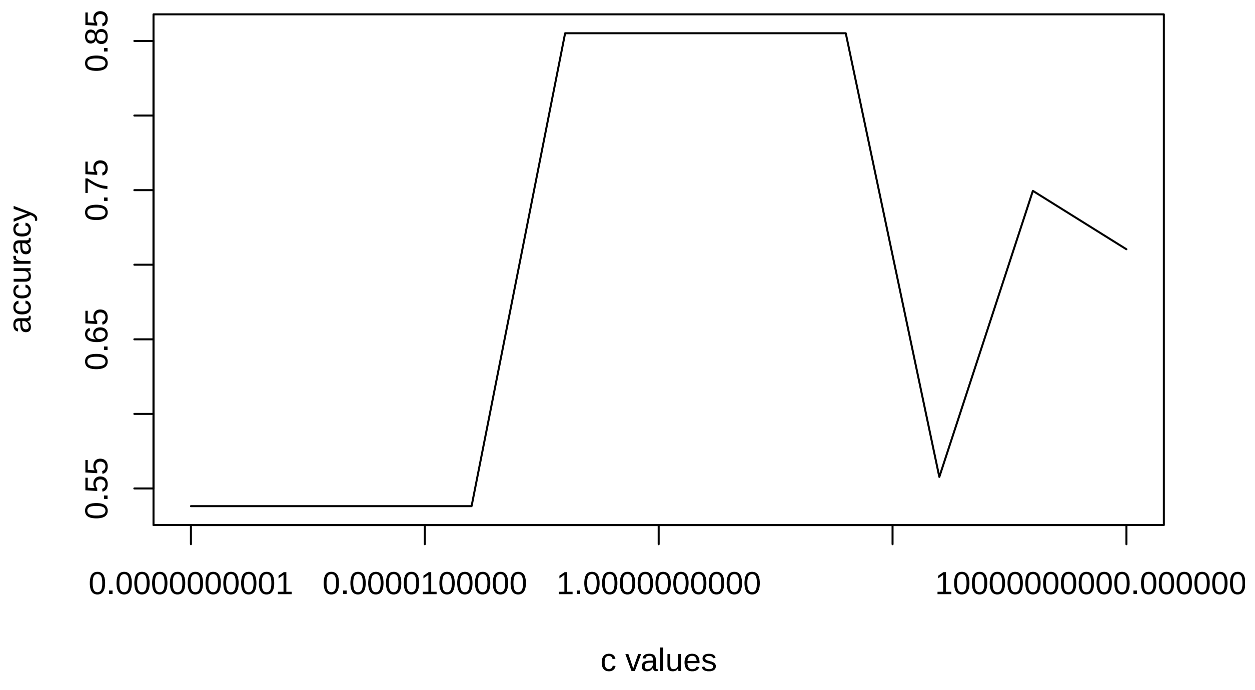
We want step through a wide range of C values to see what values generate the best accuracy in the model, then plot out the resulting accuracies.

lets skip every odd exponent just to hurry things along

```
cexpvalues <- seq(-10, 10, 2)
accuracies = matrix(1:length(cexpvalues), nrow = length(cexpvalues), ncol = 2)
i = 1
for (cexp in cexpvalues) {
  c <- 10^cexp
  accuracies[i,1] = c
  accuracies[i,2] = train_ksvm(c, 5)
  i = i + 1
}
```

The following is our plot of the resulting accuracies for a given c value.

```
plot(accuracies, type='l', log="x", xlab='c values', ylab='accuracy', )
```



We'll then get the maximum accuracy from the accuracies array and get the c values for that max accuracy. We'll then pretty it up by converting the c values from scientific notation.

```
ksvm_accuracy = max(accuracies[,2])
idxs = which(accuracies[,2] == ksvm_accuracy)
c_vals = accuracies[idxs,1]
```

Our training set has a maximum accuracy of 0.8754789. With c values of:

```
c_vals
```

```
## [1]     0.01     1.00   100.00 10000.00
```

Since there are multiple c values with the same accuracy, and the ksvm's function c value default of C=1 is one of them, we will use that on our testing set.

```
model <- ksvm(testX, testY, type="C-svc", kernel="vanilladot",
        C=1, scaled=TRUE)
```

```
##  Setting default kernel parameters
```

```
predictions <- predict(model,testX)
accuracy = sum(predictions == testY) / nrow(test_matrix)
```

Our test set model accuracy is 0.8863636.

## Question 3.1

### (b) splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

I will test using KNN.

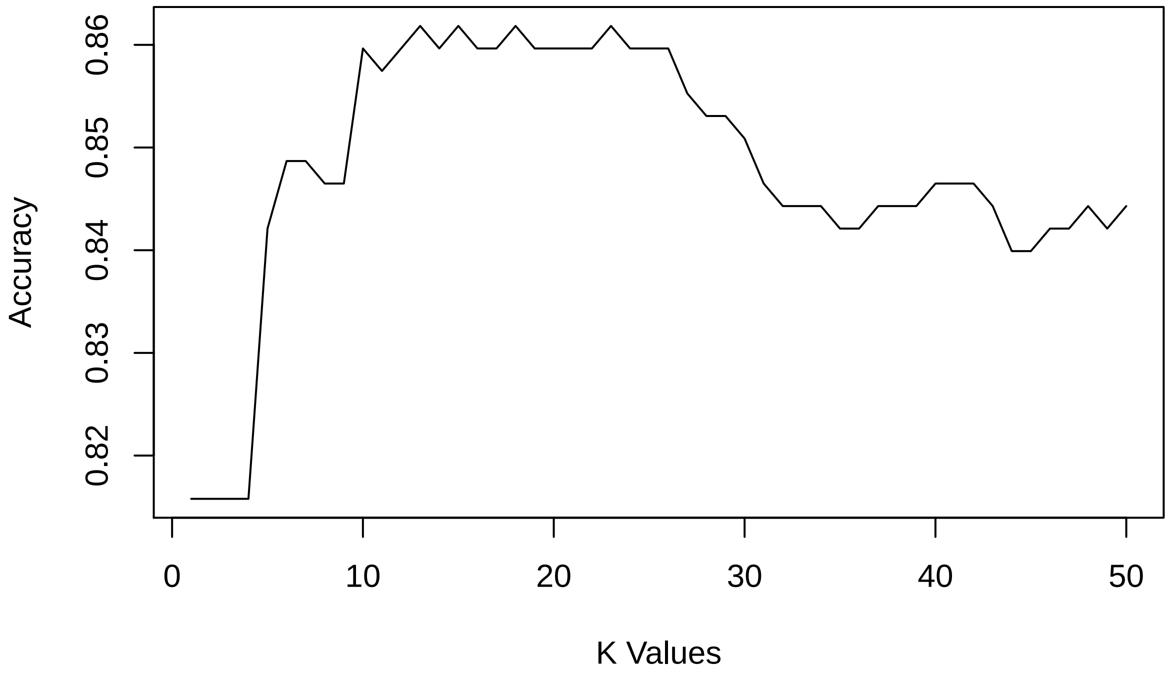
Split the data into training, validation, and testing sets (70, 15, 15).

```
row_count = nrow(data)
sample_rows <- sample(seq(1:row_count))
train_splice <- as.integer(row_count*0.7)
valid_splice <- as.integer(row_count*0.85)
train_rows = sample_rows[1:train_splice-1]
valid_rows = sample_rows[train_splice:valid_splice-1]
test_rows = sample_rows[valid_splice:row_count]

train_set = data[train_rows,]
valid_set = data[valid_rows,]
test_set = data[test_rows,]
```

```
  train_kknn <- function(dataset, k_max) {
  predictions <- rep(0,(nrow(dataset)))
  ksteps = seq(1:k_max)
  accuracies <- rep(0,(length(ksteps)))
  model <- train.kknn(R1~., dataset, kmax=tail(ksteps, 1), scale=TRUE)

  for(k in ksteps) {
    fitted_model = fitted(model)[[k]][1:nrow(dataset)]
    predictions = as.integer(round(fitted_model, 0))
    accuracies[k] = sum(predictions == dataset[, c('R1')]) / nrow(dataset)
  }
  return (accuracies)
}
train_accuracies = train_kknn(train_set, 50)
```

Below is the resulting plot of our K Value accuracies.

```
plot(train_accuracies, type='l', xlab='K Values', ylab='Accuracy')
```



```
train_accuracy <- max(train_accuracies)
train_ks = which.max(train_accuracies)
train_accuracy
```

```
## [1] 0.8618421
```

```
train_ks
```

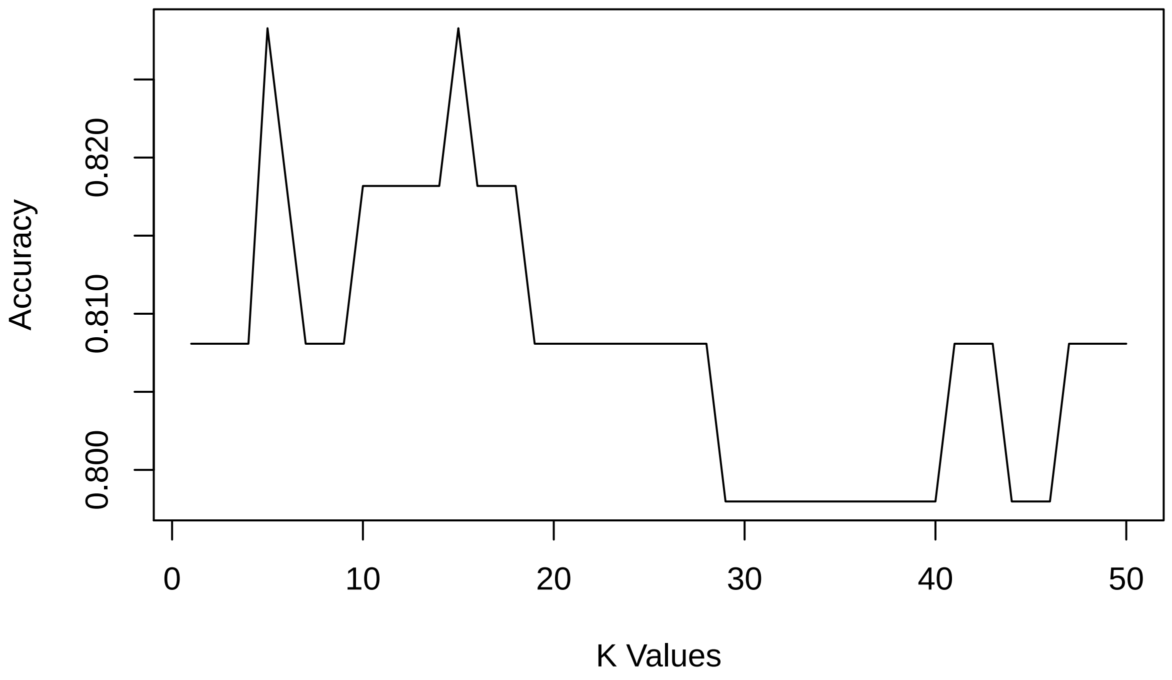
```
## [1] 13
```

The best accuracy using KNN by looping though our training set is 0.8618421 with a k values of 13.

Lets now test KNN on our validation set, looping through k values 1 to 50.

```
validate_kknn <- function(train, test, k) {
  ksteps <- seq(1:k)
  predictions = rep(0,nrow(valid_set))
  valid_accuracies <-rep(0,length(ksteps))
  for (ks in ksteps) {
    model <- kknn(R1~.,train_set,valid_set,k=ks,scale=TRUE)
    predictions <- as.integer(fitted(model)+0.5)
    valid_accuracies[ks] = sum(predictions == valid_set$R1) / nrow(valid_set)
  }
  return (valid_accuracies)
}
valid_accuracies <- validate_kknn(train_set, valid_set, 50)
```

```r
plot(valid_accuracies, type='l', xlab='K Values', ylab='Accuracy')
```



```r
valid_accuracies
```

```
##  [1] 0.8080808 0.8080808 0.8080808 0.8080808 0.8282828 0.8181818 0.8080808
##  [8] 0.8080808 0.8080808 0.8181818 0.8181818 0.8181818 0.8181818 0.8181818
## [15] 0.8282828 0.8181818 0.8181818 0.8181818 0.8080808 0.8080808 0.8080808
## [22] 0.8080808 0.8080808 0.8080808 0.8080808 0.8080808 0.8080808 0.8080808
## [29] 0.7979798 0.7979798 0.7979798 0.7979798 0.7979798 0.7979798 0.7979798
## [36] 0.7979798 0.7979798 0.7979798 0.7979798 0.7979798 0.8080808 0.8080808
## [43] 0.8080808 0.7979798 0.7979798 0.7979798 0.8080808 0.8080808 0.8080808
## [50] 0.8080808
```

```r
valid_accuracy = max(valid_accuracies)
valid_ks = which.max(valid_accuracies)
valid_accuracy
```

```
## [1] 0.8282828
```

```r
valid_ks
```

```
## [1] 5
```

While 5 is not so hot in the validation set, k=13 still performs well in the validation phase, we will use that for our final model.

```r
predictions = rep(0,nrow(test_set))
model <- kknn(R1~.,train_set,test_set,k=13,scale=TRUE)
predictions <- as.integer(fitted(model)+0.5)
test_accuracy = sum(predictions == test_set$R1) / nrow(test_set)
test_accuracy
```

```
## [1] 0.87
```

The best accuracy using KNN by looping though our training set is 0.8618421 with a k value of 13. Our validation model is a bit worse at its best k=5, but that also looks like an outlier in the plot. On our validation plot, k=13 still looks to perform rather well with an accuracy of 0.8181818. While aour testing set performed similarly to our training set with an accuraxy of 0.87. This is still close to our k value of 20 that was picked using cross validation, which used 80% of the data for testing/validation vs the 70% used here to train and then 15% to validate our data. The cross validation used more of the data then what I used here in the looping method. Reviewing the plot of accuracies, the looping method looks a little too optimistic when compared to the cross validation approach.

## Question 4.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a clustering model would be appropriate. List some (up to 5) predictors that you might use.

I work for a FinTech Company that parses Quarterly and Annual Reports from the SEC Edgar Database. We have a precanned list of risk factors that are associated with each document based on a regexes of keywords. When a new risk factor appears in a company's report, we would like to be able to tell which risk factor is significant based on past return performance of that risk factor initially appearing in a Quarterly or Annual report. Some examples of the risk factors are 'currency', 'trade war', 'bankruptcy'. Our predictors would be the risk factor vs a count of word matches of each risk factor in the company's report. We would then use the results to determine which risk factor's are significant in regards to future returns.

## Question 4.2

The iris data set iris.txt contains 150 data points, each with four predictor variables and one categorical response. The predictors are the width and length of the sepal and petal of flowers and the response is the type of flower. The data is available from the R library datasets and can be accessed with iris once the library is loaded. It is also available at the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Iris ). The response values are only given to see how well a specific method performed and should not be used to build the model.

Read in the data from the R Iris Dataset

```
irisdata <- iris
```

Lets check if the data loaded by checking the head and tail.

```
table <- xtable(head(irisdata))
print(table, type='latex', comment=FALSE)
```

| | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|---|---|---|---|---|---|
| 1 | 5.10 | 3.50 | 1.40 | 0.20 | setosa |
| 2 | 4.90 | 3.00 | 1.40 | 0.20 | setosa |
| 3 | 4.70 | 3.20 | 1.30 | 0.20 | setosa |
| 4 | 4.60 | 3.10 | 1.50 | 0.20 | setosa |
| 5 | 5.00 | 3.60 | 1.40 | 0.20 | setosa |
| 6 | 5.40 | 3.90 | 1.70 | 0.40 | setosa |

```
table <- xtable(tail(irisdata))
print(table, type='latex', comment=FALSE)
```

Lets split our data into training and test sets.

```
samples <- sample(1:2, size=nrow(irisdata), prob=c(0.70,0.30), replace = TRUE)
iris_train = irisdata[samples==1,]
iris_test = irisdata[samples==2,]
```

|  | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
|---|---|---|---|---|---|
| 145 | 6.70 | 3.30 | 5.70 | 2.50 | virginica |
| 146 | 6.70 | 3.00 | 5.20 | 2.30 | virginica |
| 147 | 6.30 | 2.50 | 5.00 | 1.90 | virginica |
| 148 | 6.50 | 3.00 | 5.20 | 2.00 | virginica |
| 149 | 6.20 | 3.40 | 5.40 | 2.30 | virginica |
| 150 | 5.90 | 3.00 | 5.10 | 1.80 | virginica |

We need to determine the optimal clusters to use with our dataset. We will use the elbow method to loop through values of k # of clusters from 1 to 15 on our training, validation, and test sets. Then plot the results of the total within sum of squares to determine the optimal k value.

```
x <- iris_train[,1:4]
species <- iris_train[, 5]

kmeans_elbow <- function(kmax, dataset) {
  scaled_x <- as.matrix(scale(x, center=FALSE))
  wss <- sapply(1:kmax,
    function(k){
      kmeans(scaled_x, k, nstart=50,iter.max = 15 )$tot.withinss
    }
  )
return(wss)
}
k.max = 15
wss = kmeans_elbow(k.max, x)
```

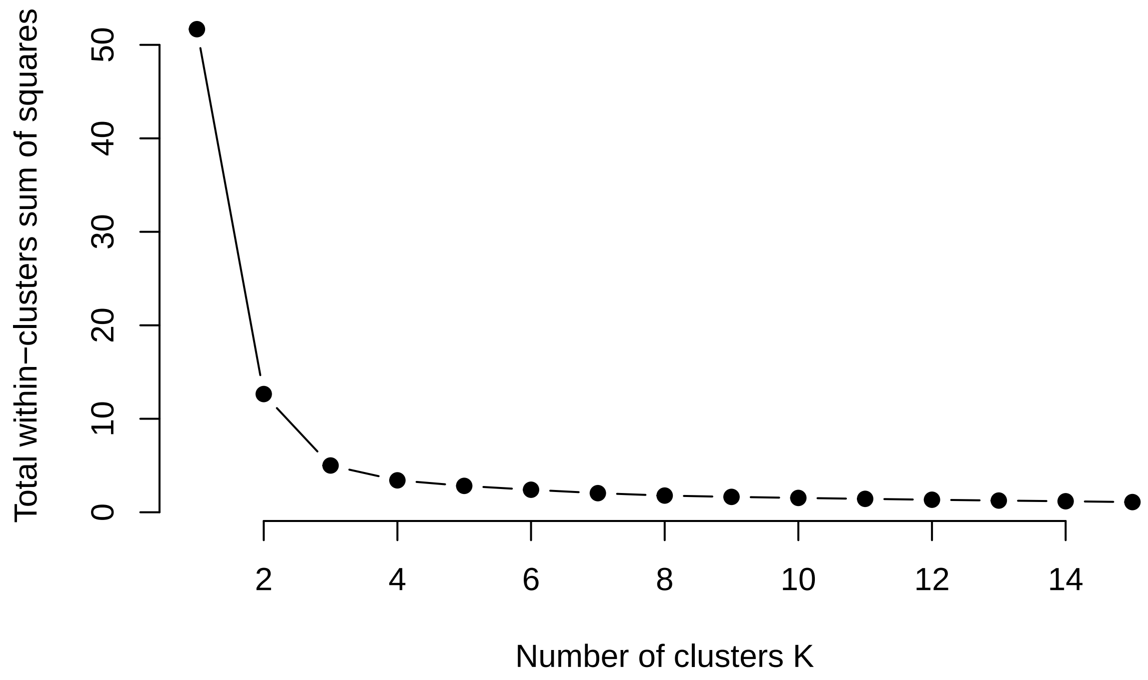
our sum of squares array is:

```
table = xtable(t(as.matrix(wss)))
print(table, type='latex', comment=FALSE)
```

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 51.68 | 12.65 | 5.00 | 3.42 | 2.82 | 2.41 | 2.05 | 1.79 | 1.64 | 1.54 | 1.44 | 1.34 | 1.25 | 1.18 | 1.09 |

Our plot looks like:

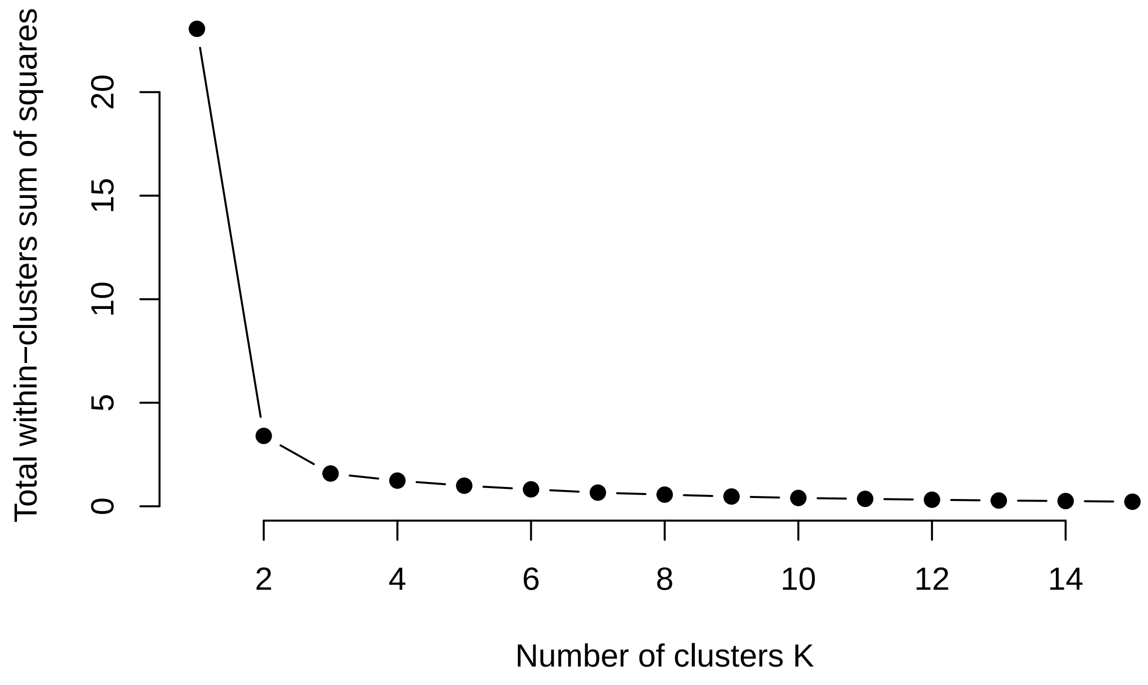
```
plot(1:k.max, wss,
     type="b", pch = 19, frame = FALSE,
     xlab="Number of clusters K",
     ylab="Total within-clusters sum of squares")
```



The optimal number of clusters to use for our kmeans model is 3 from our training set.

Lets run the same analysis on our test set.

```
x <- iris_test[,1:4]
species <- iris_test[, 5]
wss = kmeans_elbow(k.max, x)
plot(1:k.max, wss,
     type="b", pch = 19, frame = FALSE,
     xlab="Number of clusters K",
     ylab="Total within-clusters sum of squares")
```



Our test set below confirms that k=3 is an optimal number of clusters for our model.

## Optimal Kmeans Models

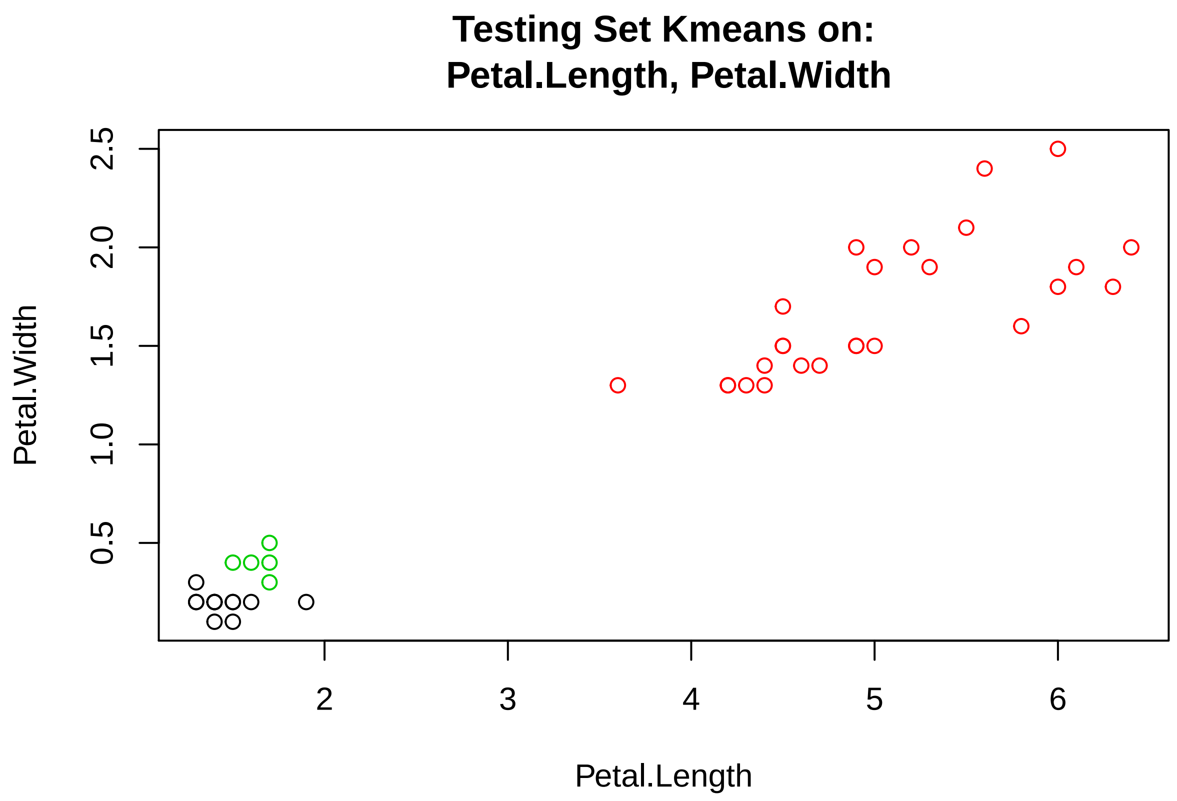
This is the function I used for determining the best Kmeans model to use. It plots cluster scatter plots of the passed in data set and an optional summary table displaying accuracy of the clusters. The model parameters selected were based on reviewing the cluster plots of all possible models.

```
kmeans_runplot <- function(dataset, species, columns, set_type, displaytable=TRUE, paginate=FALSE) {
  # convert the passed in column numbers to column names
  column_names <- colnames(dataset)
  cname = column_names[c(columns)]
  # title for our summary table and plot
  title =  paste(set_type, paste('Kmeans on:\n', paste(cname, collapse=', '), collapse=' '), collapse='
  x <- dataset[,c(cname)]
  # scale our data set
  scaled_x <- as.matrix(scale(x, center=FALSE))
  # run kmeans on our scaled data set
  kx <- kmeans(scaled_x, 3)
  # create our summary table
  datatable = xtable(table(species, kx$cluster), caption=title)
  # add a page break of selected
  if (paginate) {
    cat("\n\n\\pagebreak\n")
  }
  # print our summary table if selected
  if (displaytable) {
    print(datatable, comment=FALSE, caption.placement='top')
  }
  # plot our cluster chart
  plot(x[c(cname)], col=kx$cluster, main=title)
  # plot the centroids
  points(x$centers[,c(cname)], col=1:3, pch=23, cex=3)
}
```

By analyzing the plots at the end of this document for the various Kmeans models below and in particular the "Sepal.Length, Sepal.Width, Petal.Length, Petal.Width" on page 12 and "Petal.Length, Petal.Width" on page 22. I determined that the best clustering occurs on the model using "Petal.Length, Petal.Width" in the training set.

We will use the Kmeans model only using "Petal.Length, Petal.Width" with our testing set.

```
x <- iris_test[,3:4]
species <- iris_test[, 5]
kmeans_runplot(x, species, c(1,2), 'Testing Set', displaytable=FALSE)
```



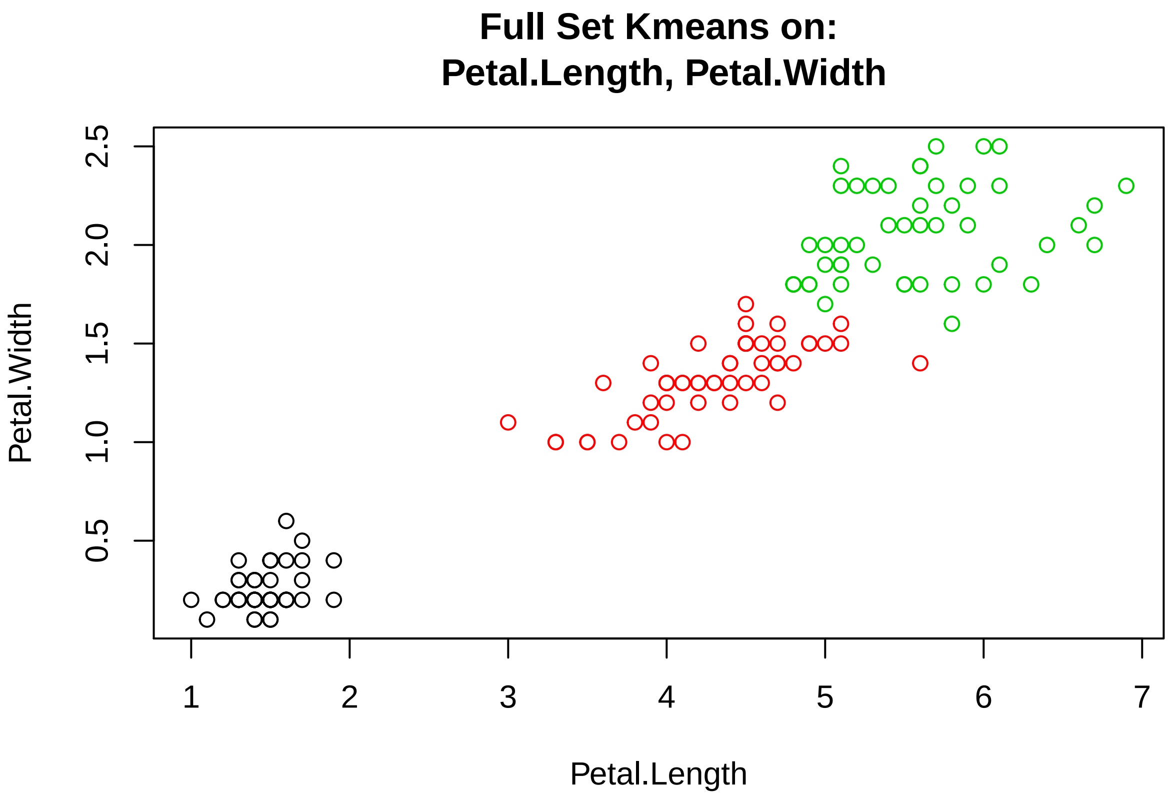
The performance of the Kmeans model using "Petal.Length, Petal.Width" looks acceptable on the testing set.

Lets run the Kmeans model using "Petal.Length, Petal.Width" now over the full data set.

```
x <- irisdata[, 3:4]
species <- irisdata[, 5]
kmeans_runplot(x, species, c(1,2), 'Full Set', paginate=TRUE)
```

Table 1: Full Set Kmeans on: Petal.Length, Petal.Width

| | 1 | 2 | 3 |
|---|---|---|---|
| setosa | 50 | 0 | 0 |
| versicolor | 0 | 48 | 2 |
| virginica | 0 | 4 | 46 |



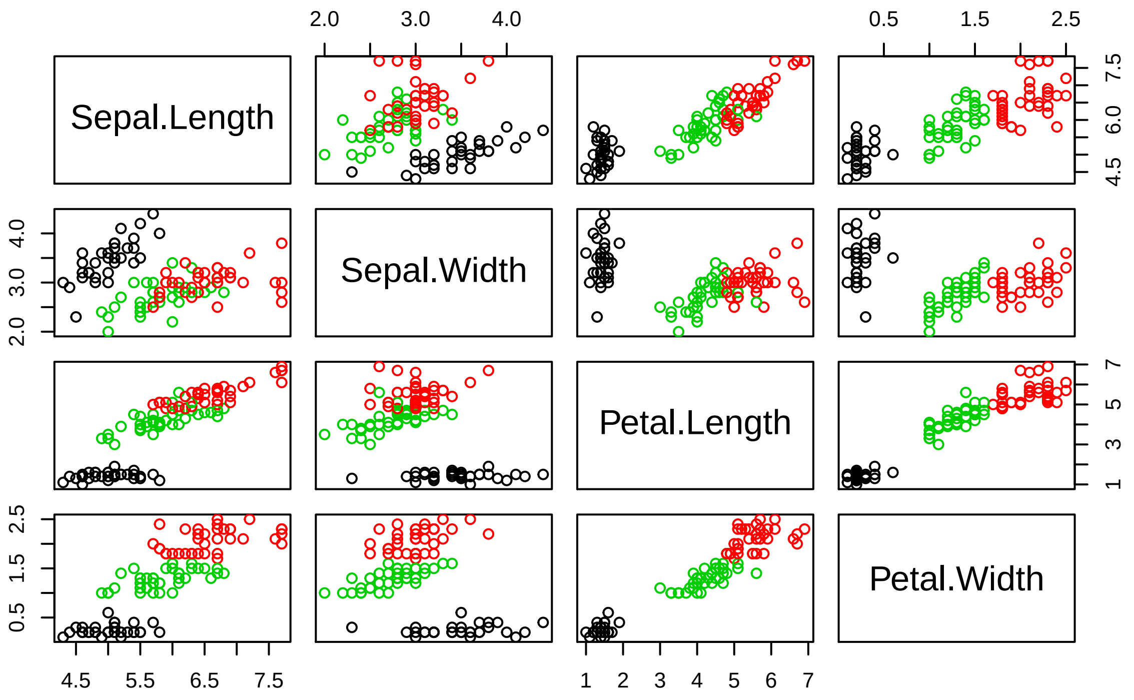
The Kmeans model using "Petal.Length, Petal Width" looks optimal.

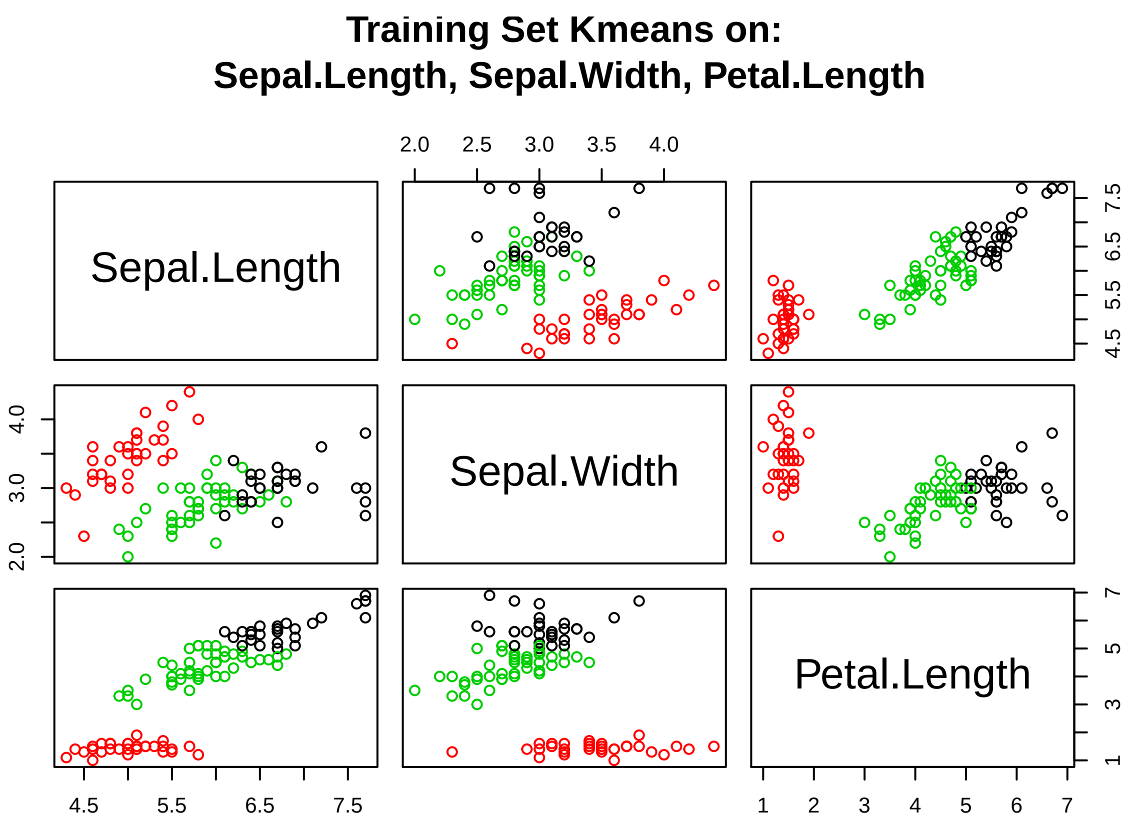
## Appendix

The code snippet below loops through our kmeans models using all available dimension combinations to create all the models analyzed for this question. See the models on the pages following this code snippet to see all the model details considered for the above summary.

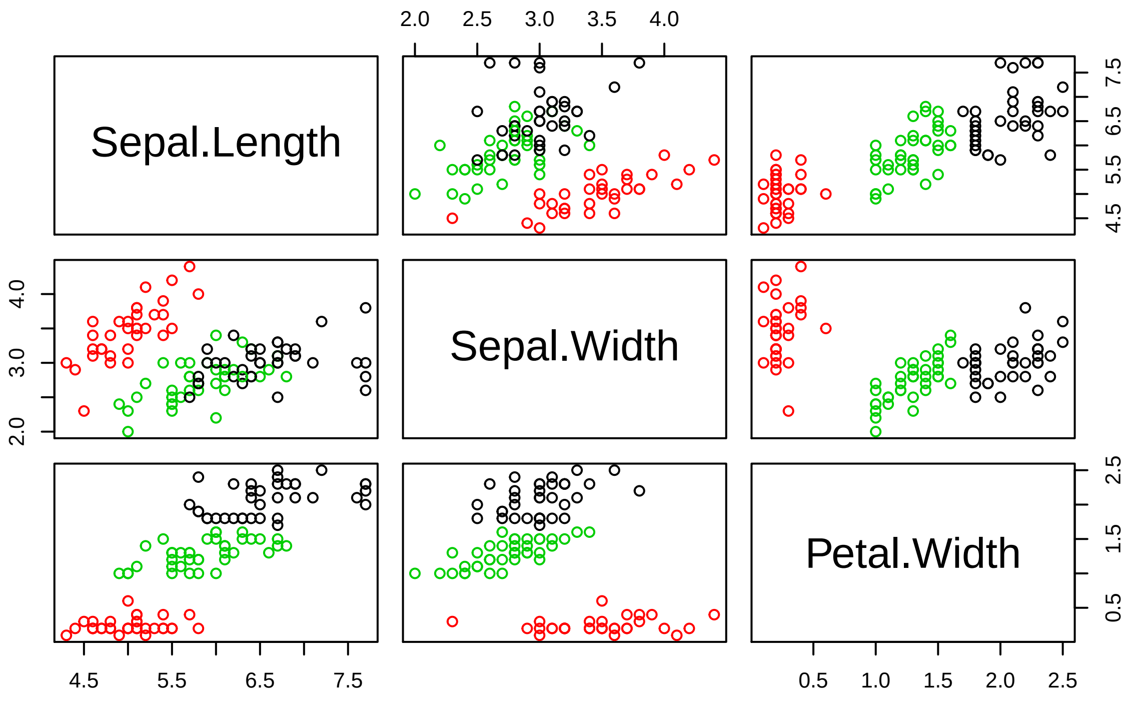
```
x <- iris_train[,1:4]
species <- iris_train[, 5]
# create combinations of column numbers to select
comb3 = t(combn(4,3))
comb2 = t(combn(4,2))
l = nrow(comb3) + nrow(comb2) + 1
combinations = matrix(tuple(), nrow=l, ncol=1)
# choose all 4 data columns
combinations[1] <- tuple(combn(4,1))
# loop through our combinations and add to the combinations array
i = 2
for(row in seq(1:nrow(comb3))) {
  combinations[i] <- tuple(comb3[row,])
  i = i + 1
}
for(row in seq(1:nrow(comb2))) {
  combinations[i] <- tuple(comb2[row,])
  i = i + 1
}
# loop through the built combinations array and run our kmeans function
for(columns in combinations) {
  kmeans_runplot(x, species, columns, 'Training Set', displaytable=FALSE, paginate=TRUE)
}
```

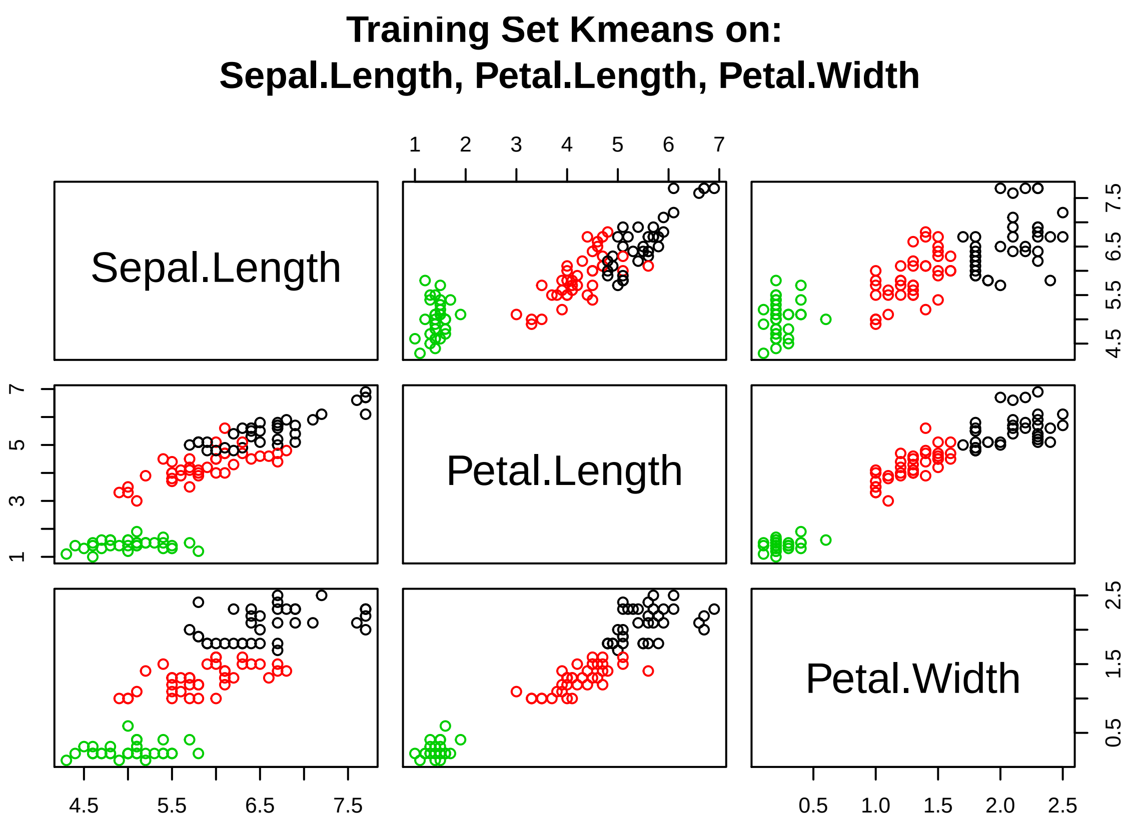
## Training Set Kmeans on:
## Sepal.Length, Sepal.Width, Petal.Length, Petal.Width



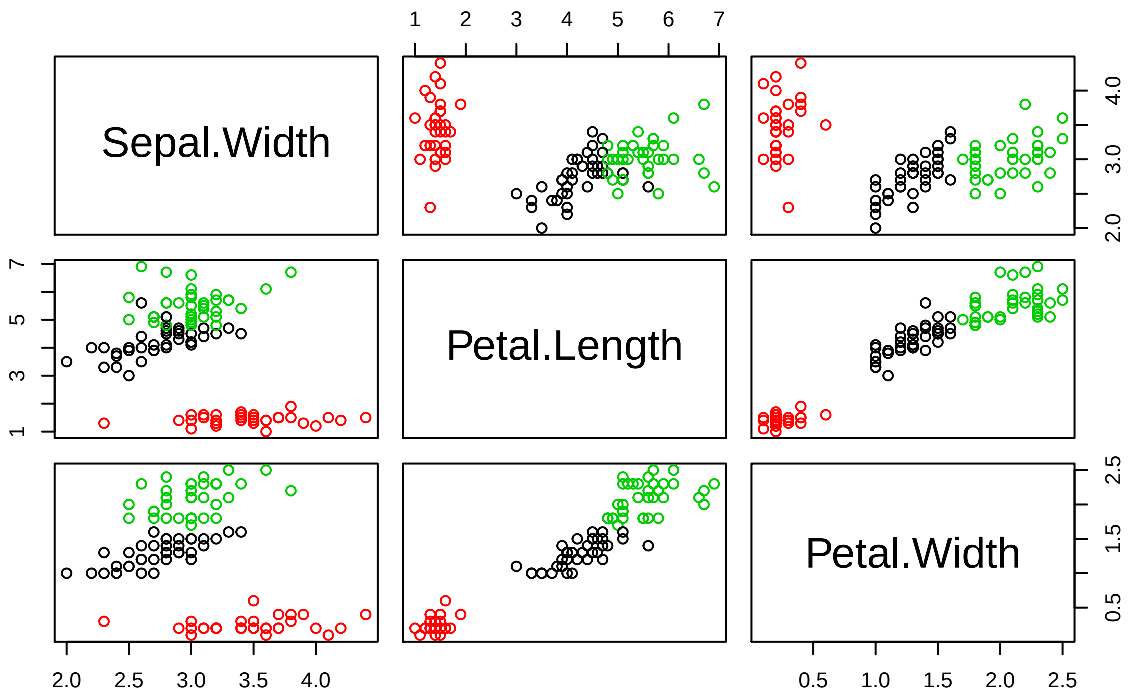

Training Set Kmeans on:
Sepal.Length, Sepal.Width, Petal.Length
Sepal.Length
Sepal.Width
Petal.Length
2.0
2.5
3.0
3.5
4.0
4.5
5.5
6.5
7.5
1
2
3
4
5
6
7

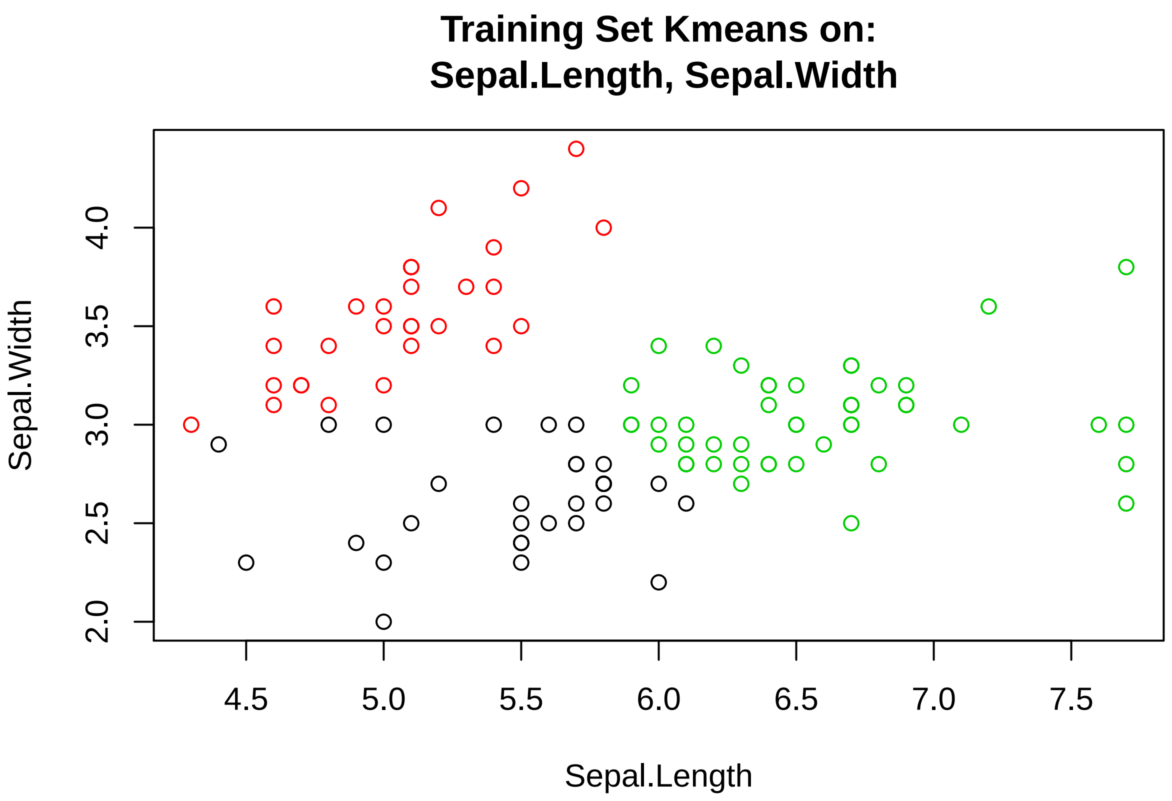
**Training Set Kmeans on:**
**Sepal.Length, Sepal.Width, Petal.Width**

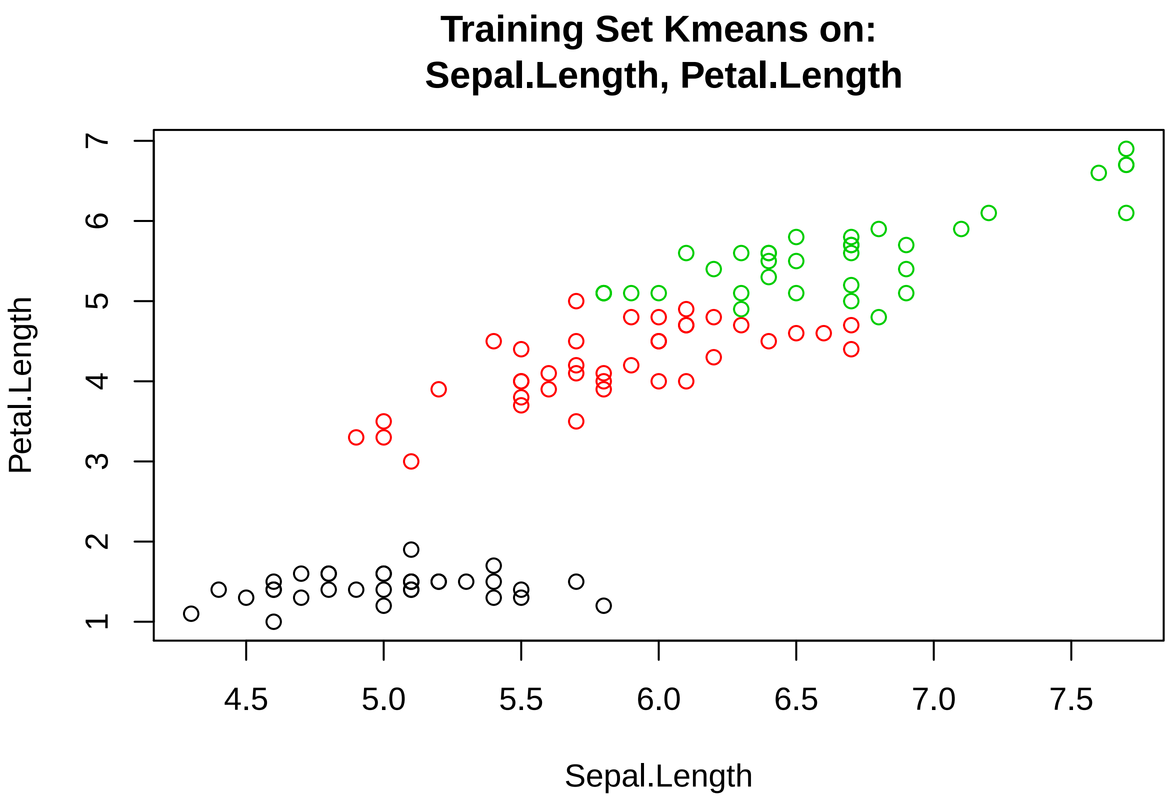


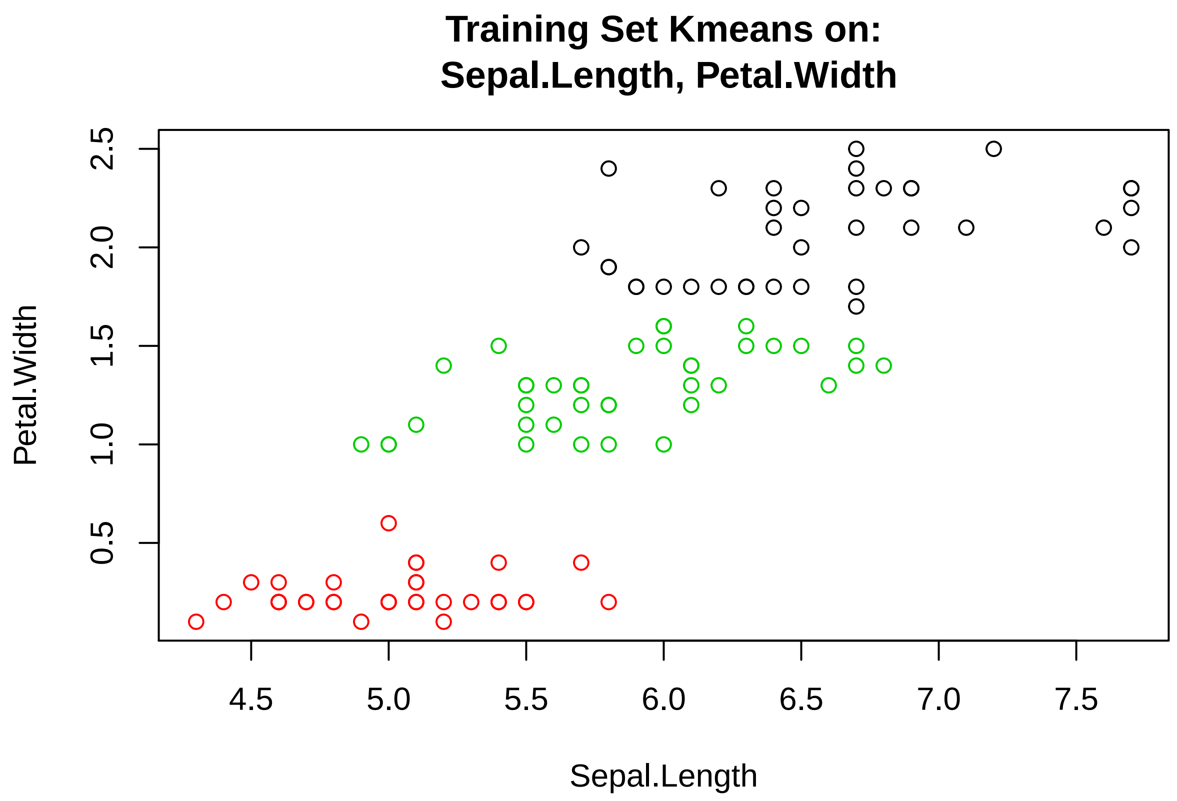

Training Set Kmeans on:
Sepal.Length, Petal.Length, Petal.Width
Sepal.Length
Petal.Length
Petal.Width
1
2
3
4
5
6
7
4.5
5.5
6.5
7.5
0.5
1.0
1.5
2.0
2.5

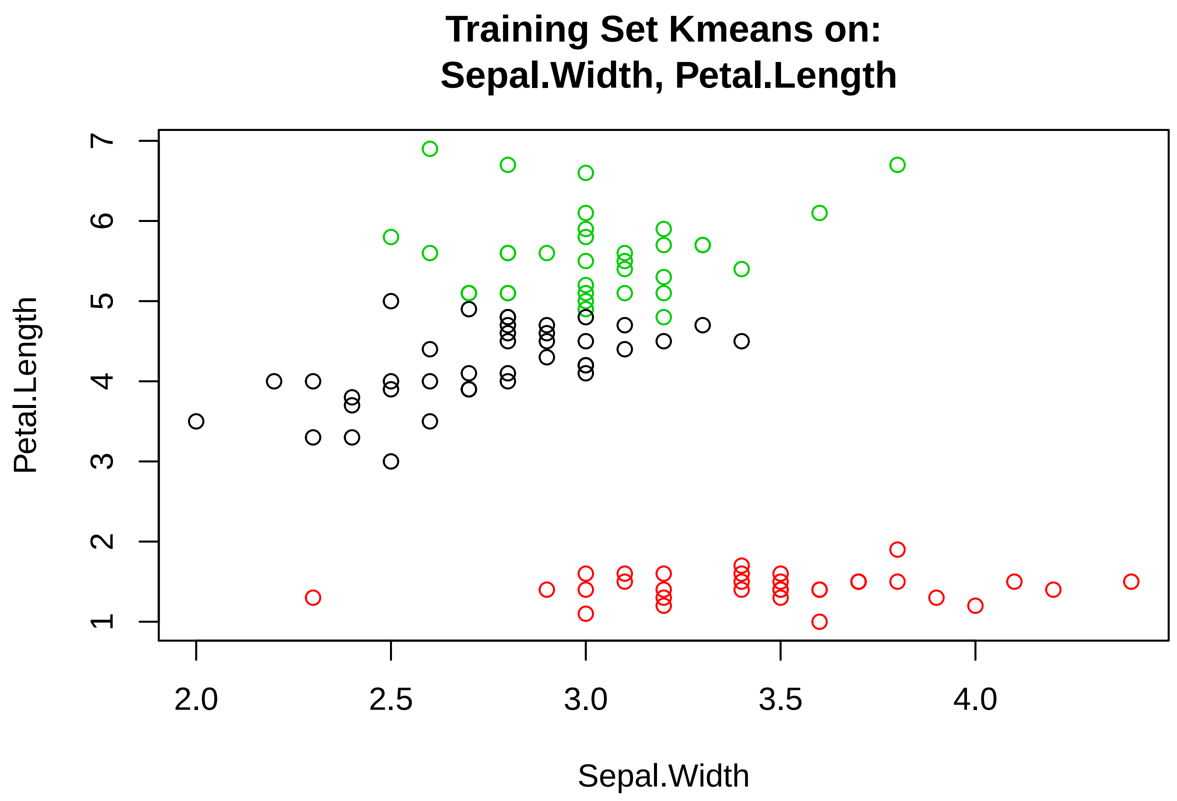
## Training Set Kmeans on: Sepal.Width, Petal.Length, Petal.Width

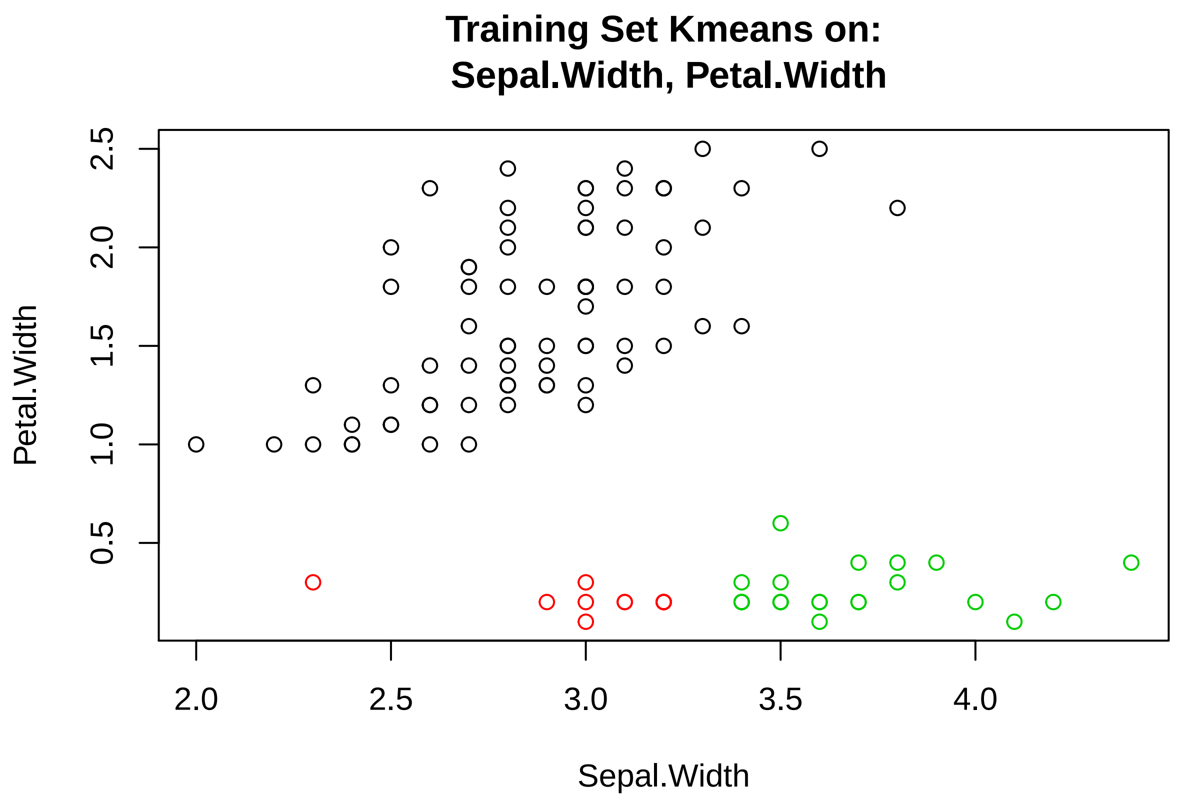


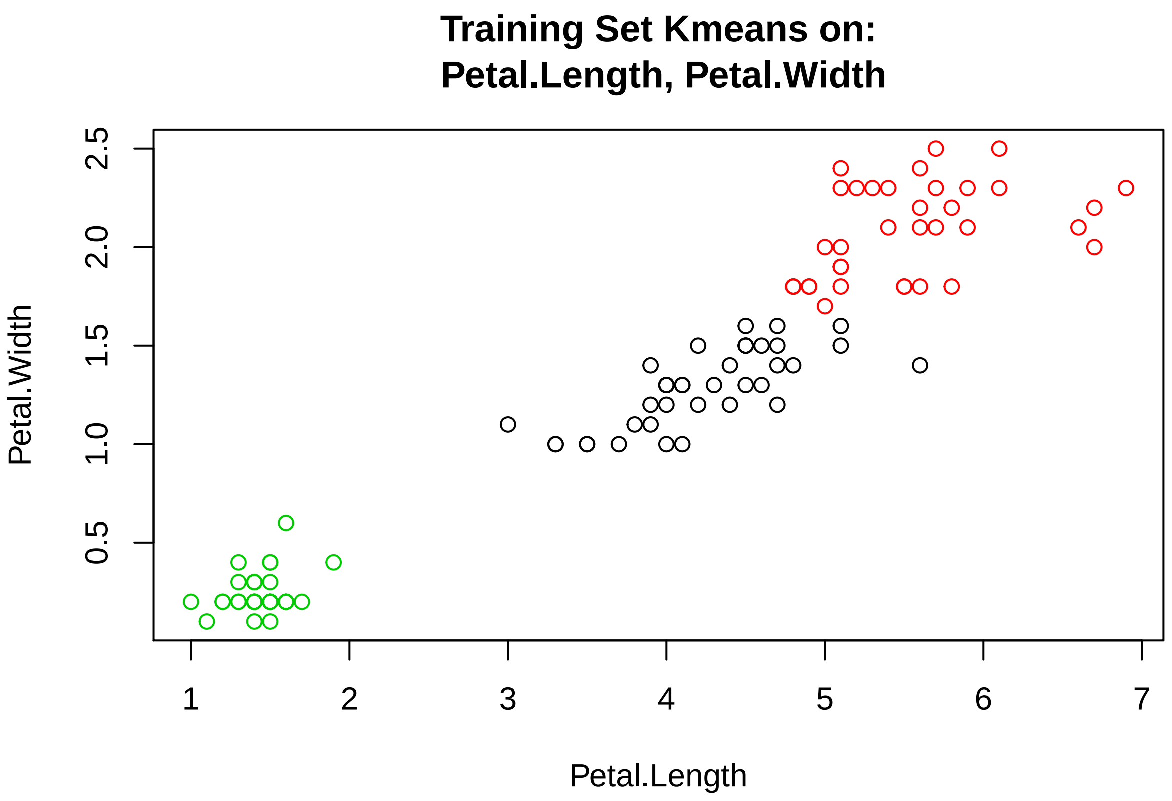

Training Set Kmeans on:
Sepal.Length, Sepal.Width
Sepal.Width
Sepal.Length
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5


Training Set Kmeans on:
Sepal.Length, Petal.Length
Petal.Length
Sepal.Length
1
2
3
4
5
6
7
4.5
5.0
5.5
6.0
6.5
7.0
7.5


Training Set Kmeans on:
Sepal.Length, Petal.Width
Petal.Width
0.5
1.0
1.5
2.0
2.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
Sepal.Length


Training Set Kmeans on:
Sepal.Width, Petal.Length
Petal.Length
Sepal.Width
1
2
3
4
5
6
7
2.0
2.5
3.0
3.5
4.0


Training Set Kmeans on:
Sepal.Width, Petal.Width
Petal.Width
0.5
1.0
1.5
2.0
2.5
2.0
2.5
3.0
3.5
4.0
Sepal.Width


Training Set Kmeans on:
Petal.Length, Petal.Width
Petal.Width
Petal.Length
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7