An Example of k-Means using Iris Data

Alan Montgomery

10/27/2019

Introduction

In this script we illustrate R's implement of k-Means using Fisher's Iris data. This dataset has four different measurements for 150 irises measured by a botanist on a particular day in a field outside of Montreal. This is very popular dataset and you will see it frequently.

Our first step is to input the dataset and then summarize it.

```
###############################################################################
# setup
###############################################################################

# prepare data
if (!require(datasets)) {install.packages("datasets"); library(datasets)}
data("iris")    # load the data into memory

# summarize the dataset
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Notice we have five variables in the iris dataframe. `Sepal.Length` and `Septal.Width` refer to the length and width of the sepal (the green part of the flower) in centimeters, `Petal.Length` and `Petal.Width` refer to the length and width of the petal (the showy part of the flower) in centimeters, and finally `Species` is a factor that identifies the species of flower. In this dataset the expert has classified each flower, but later in this analysis we will pretend that the species is unknown to us.

Using R to understand the Data Structure

There are ways of doing the same thing in R. For example, to list the first six records use head.

```
###############################################################################
# understand the structure of the data [optional]
###############################################################################

# first look at the data
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

By default `head` will print the first six records and all variables. Alternatively, we could select the first six records using `iris[1:6,]`. Consider the following methods of showing the first two rows and first two variables using different types of selections.

```
# R makes extensive use of the select operator
# multiple ways to list the first 6 rows and first 2 columns
head(iris[,1:2])  # select the first two columns and then use head
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

```
iris[1:6,1:2]     # here we think of the data frame as a table and get the first 6 rows and 2 columns
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

```
iris[1:6,c(1,2)]  # notice that 1:2=seq(1,2)=c(1,2)
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

```
iris[1:6,c("Sepal.Length","Sepal.Width")]  # another way to do the same thing using the names of the co
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

```
varnames=c("Sepal.Length","Sepal.Width")   # create a vector of the variable names
iris[1:6,varnames]  # another way using another object that has the variables
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

If the rows have names we can also select the appropriate rows using the character strings that correspond to the names, just as we did with the columns.

```
# some more variations
iris[c("1","2","3"),varnames]  # sometimes rows have names (see rownames) and we can use these names to
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
```

```
iris$Sepal.Length[1:6]   # yet another way if we just want to access a column directly
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

```
subset(iris,subset=(rownames(iris) %in% c("1","2","3")),select=varnames)  # and yet another, see also d
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
```

```
# notice we use %in% which is another notation for match(rownames(iris),c("1","2","3"),nomatch=0)>0
```

Visualizing the data

A simple graphic for visualizing the distribution of the data is the boxplot. `boxplot` (as well as many other functions in R) use what is known as formula syntax the basics of formula is left hand side is the dependent variance and the right side are the independent variables. So for instance `Petal.Length~Species` tells boxplot to have `Petal.Length` in the y-axis against `Species` in the x-axis. One nice thing is that when use formula syntax R knows that `Species` is a factor variable so it knows to automatically create dummy variables for each of its levels "setosa", "versicolor" and "virginica". Notice if you execute the command `boxplot(Petal.Length)` then we only get a single boxplot that has all species.

```
###########################################################################################
# visualize the data
###########################################################################################

# plot iris data in boxplot
par(mfrow=c(4,1),mar=c(3,4,1,1))  # mfrow=c(4,1) tells R to plot 4 rows and 1 column, and mar is the ma
boxplot(Petal.Length~Species,data=iris,ylab="Petal Length")
boxplot(Petal.Width~Species,data=iris,ylab="Petal Width")
boxplot(Sepal.Length~Species,data=iris,ylab="Sepal Length")
boxplot(Sepal.Width~Species,data=iris,ylab="Sepal Width")
```

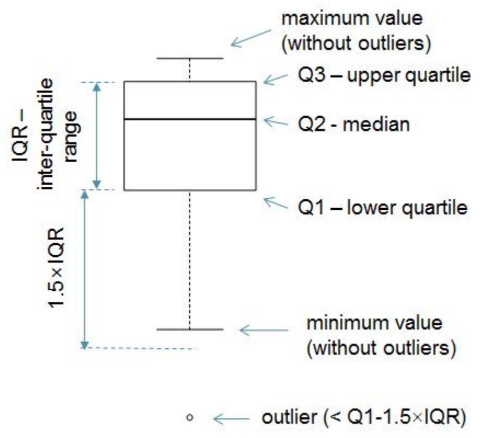
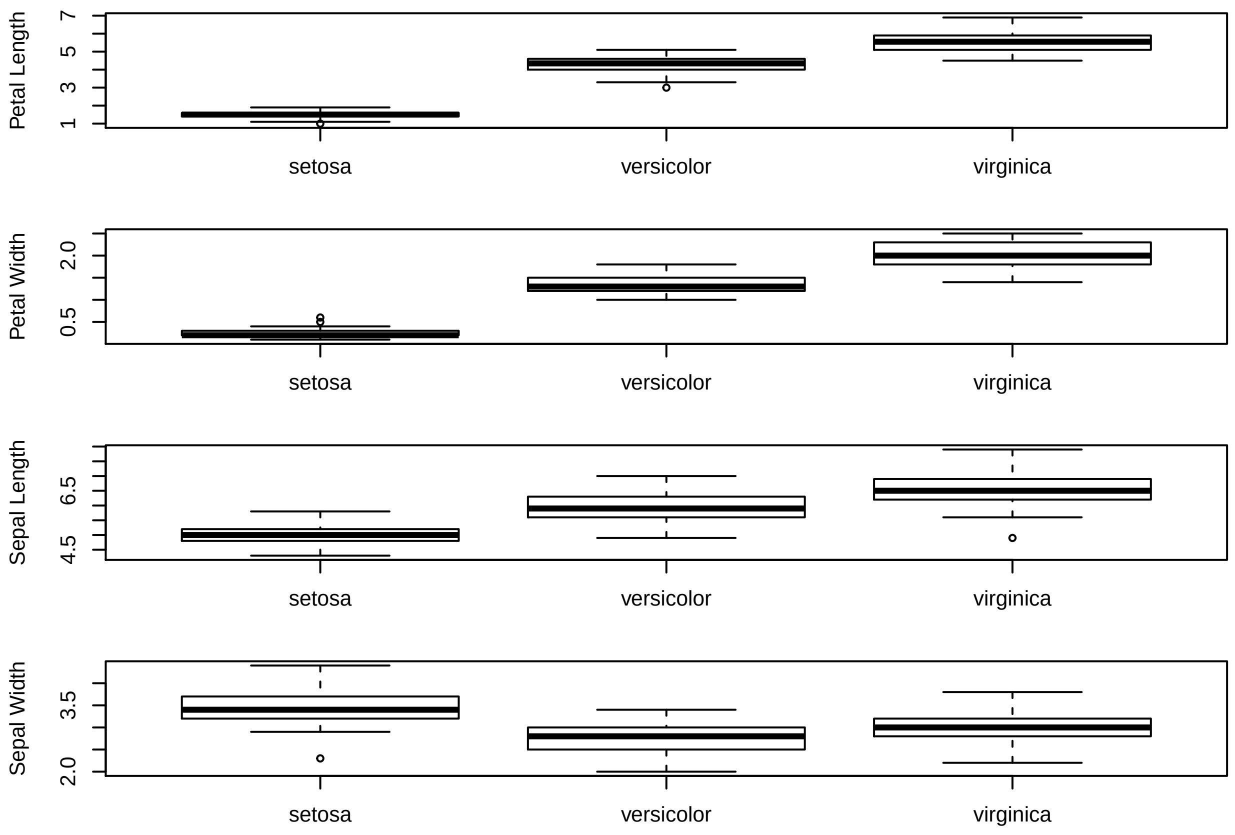


Figure 1: Boxplots provide a graphical illustration of five summary statistics: minimum, lower quartile, median, upper quartile, and maximum. By default R also separates out outliers from the minimum and maximum values and denotes them with circles.



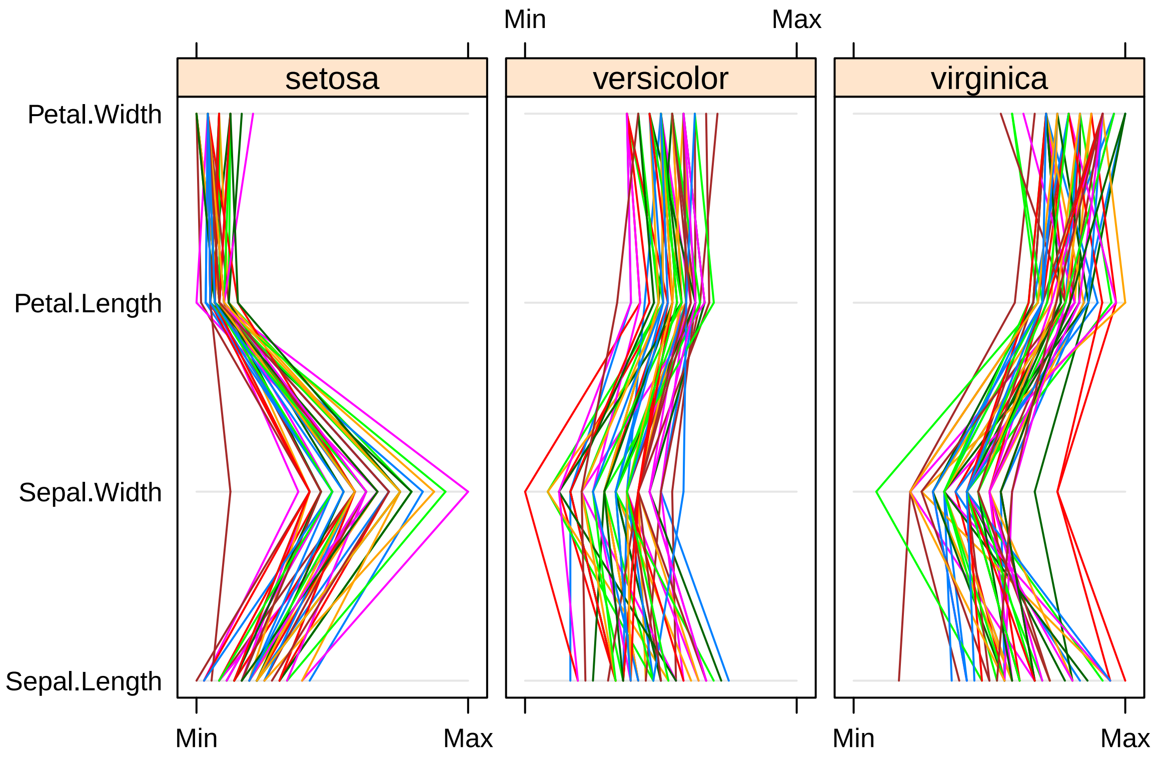
```r
par(mfrow=c(1,1))   # reset to one graph in the panel
```

A challenge in visualization is representing high dimensional data in a two-dimensional plot. A `parallelplot` is one way of trying to plot multivariate data in two-dimensional space. In this type of plot each variable is given its own axis, which are represented as a horizontal line. Each observation is then marked as a point on each axis, and then all the points are connected using a line. Typically, the lines run parallel to one another. A challenge with parallel plots is that they can become over-cluttered with information and become very dense and difficult to read. To ease the representation we use the formula operator `|` to show we want the formula applied to each of the levels of Species separately.

```
# one more visualization is a parallel lines plot
if (!require(lattice)) {install.packages("lattice"); library(lattice)}
```

```
## Loading required package: lattice
```

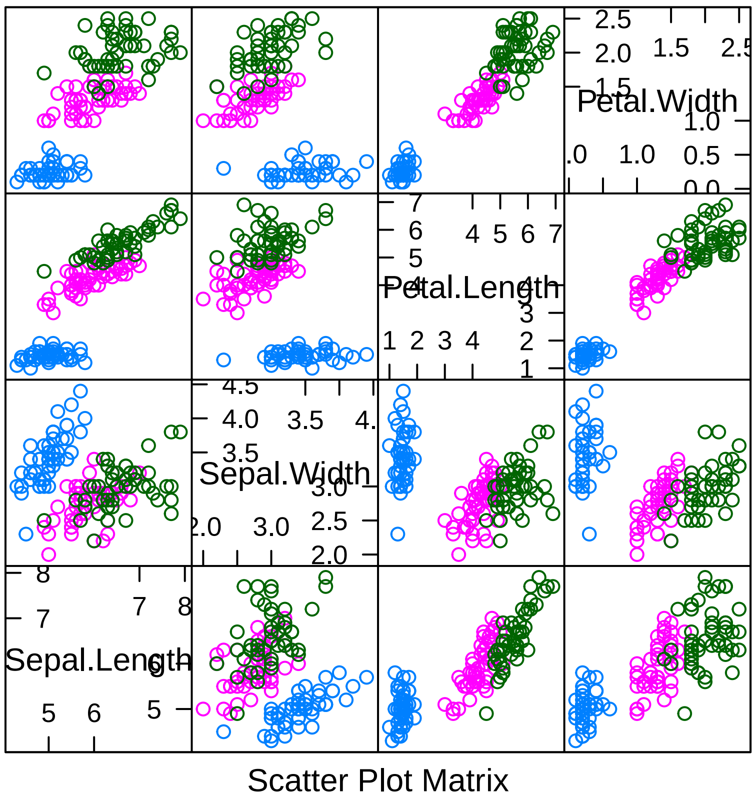
```
# the ~ ... | ... in the following line is a formula in R, and says the input variables
# are the first four columns of iris and are conditional upon the category Species
parallelplot(~iris[1:4]|Species,data=iris)
```



We can observe from these parallelplots that `virginica` tend be larger, `setosa` are smaller except for `Sepal.Width`, and `versicolor` falls somewhere in between.

Another visualization of multivariate data is to represent each pair of variables using the familiar scatter plot. The variable in the "X-axis" is identified by the column, and the variables in the "Y-axis" is identified by the row. In the following plot consider the fourth row and second column plots `Sepal.Width` (x) versus `Sepal.Length` (y). Additionally, we have used the `groups=Species` option in splom to color each of the different `Species` a different color point.

```
# scatterplot matrix plots an array of scatterplots
splom(~iris[1:4], groups = Species, data = iris)
```



Cluster the data

```
################################################################################
# cluster the data
################################################################################

# before performing k-means clusters with iris data
# remove the Species variable by setting it to NULL
newiris=iris
newiris$Species = NULL

# apply kmeans and save the clustering result to kc.
# the parantheses tell R to print/evaluate the object, alternatively we could enter
# kc=kmeans(newiris,3); print(kc)  but this gives us a simpler way to do both in one line
# there are two inputs to kmeans the dataset of newiris and setting K to the value 3
# to understand the inputs and outputs you can ask for help from R using help(kmeans) or ?kmeans
# however, the help is really meant to be syntax help not help in understanding the algorithm
set.seed(1248765792)   # set the seed so the random initialization of the clusters is the same
( kc = kmeans(newiris, 3) )
```

```
## K-means clustering with 3 clusters of sizes 62, 38, 50
##
## Cluster means:
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     5.901613    2.748387     4.393548    1.433871
## 2     6.850000    3.073684     5.742105    2.071053
## 3     5.006000    3.428000     1.462000    0.246000
##
```

```
## Clustering vector:
##   [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [36] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2
## [106] 2 1 2 2 2 2 2 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 2 2 2 2 1 2
## [141] 2 2 1 2 2 2 1 2 2 1
##
## Within cluster sum of squares by cluster:
## [1] 39.82097 23.87947 15.15100
##  (between_SS / total_SS =  88.4 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

```
str(kc)
```

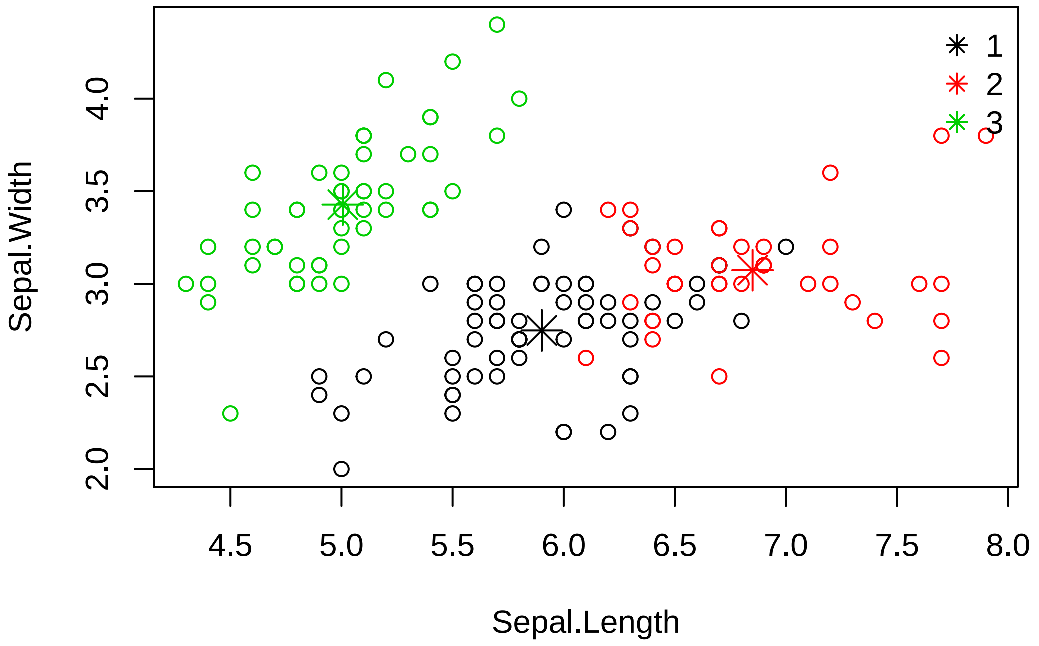
```
## List of 9
##  $ cluster     : int [1:150] 3 3 3 3 3 3 3 3 3 3 ...
##  $ centers     : num [1:3, 1:4] 5.9 6.85 5.01 2.75 3.07 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "1" "2" "3"
##   .. ..$ : chr [1:4] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
##  $ totss       : num 681
##  $ withinss    : num [1:3] 39.8 23.9 15.2
##  $ tot.withinss: num 78.9
##  $ betweenss   : num 603
##  $ size        : int [1:3] 62 38 50
##  $ iter        : int 3
##  $ ifault      : int 0
##  - attr(*, "class")= chr "kmeans"
```

`kmeans` returns a list which we name `kc`. Quite frequently R returns lists as the result, so that it can organize many different variables into a single group. In this case kmeans returns:

Variable	Description
`kc$cluster`	a vector of integers (from `1:K`) indicating the cluster to which each point is allocated
`kc$centers`	a matrix of cluster centers (the rows correspond to the clusters, columns are variables used for kmeans)
`kc$totss`	total sum of squares, which says how much variation that originally was in the dataset
`kc$withinss`	within-cluster sum of squares, vector with K dimensions, each element corresponds to a cluster
`kc$betweenss`	the between-cluster sum of squares which equals is the difference between `totss` and `betweenss`
`kc$size`	the number of pionts in each cluster
`kc$iter`	the number of iterations
`kc$ifault`	integer that indicates a possible algorithm problem

We can plot the clusters and their centers. We have color coded each of the points according to their cluster. Note that there are four dimensions in the data and that only the first two dimensions are used to draw the plot below. As you look at the plot notice that some of the points in cluster 2 are actually closer to the centroid for cluster 1. The reason is that we are only looking at two of the four dimensions. If we observe the other two dimensions we should see that these same points must be much closer to their respective clusters.

```
# scatter plot of each cluster with a different color for each cluster
par(mar=c(5,5,5,5))
plot(newiris[c("Sepal.Length", "Sepal.Width")], col=kc$cluster)
points(kc$centers[,c("Sepal.Length", "Sepal.Width")], col=1:3, pch=8, cex=2) # overlay the cluster cent
legend("topright",legend=as.character(1:3),col=1:3,pch=8,bty='n') # add legend to tell which colors cor
```



```
# look at the cluster solution
print(kc$centers)  # the centers or centroids are the averages of observation within a cluster
```

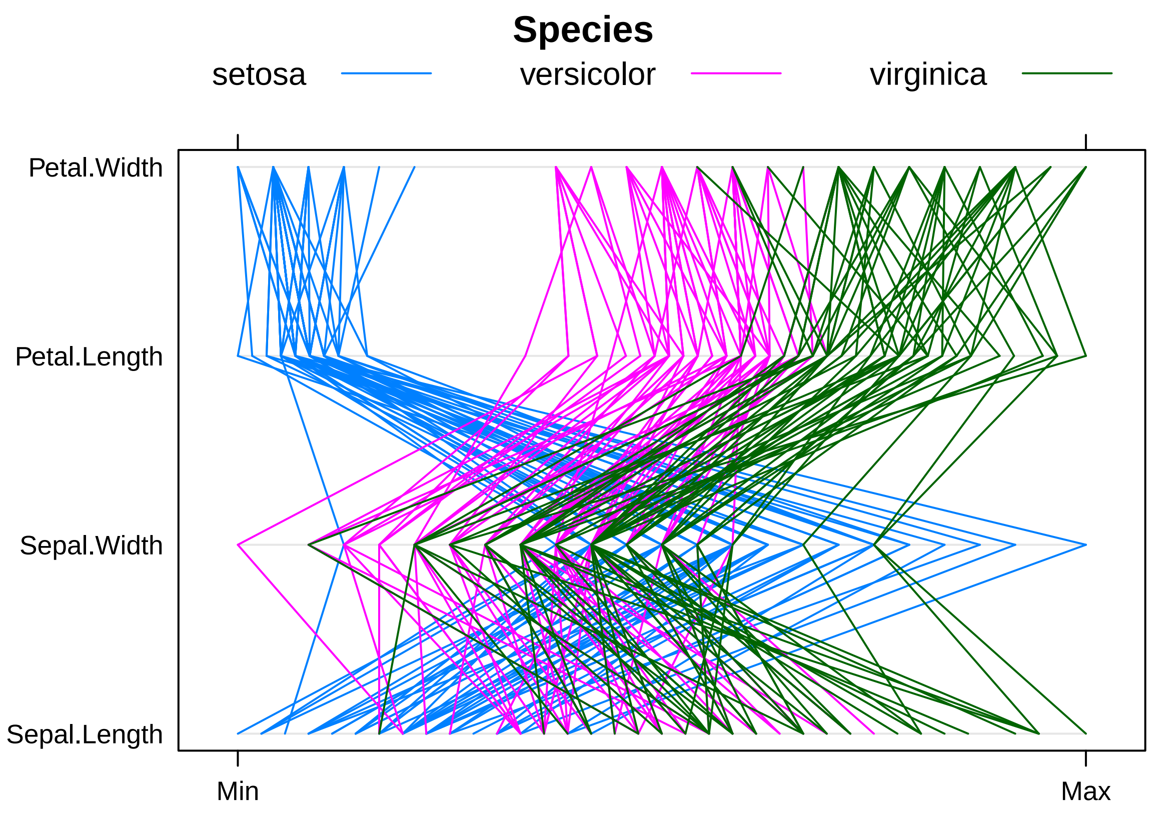
```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     5.901613    2.748387     4.393548    1.433871
## 2     6.850000    3.073684     5.742105    2.071053
## 3     5.006000    3.428000     1.462000    0.246000
```

We can manually check that the elements of the `kc$centers` match with the mean of the data. In this example we compute the mean of `Sepal.Length` for `kc$cluster==1`.

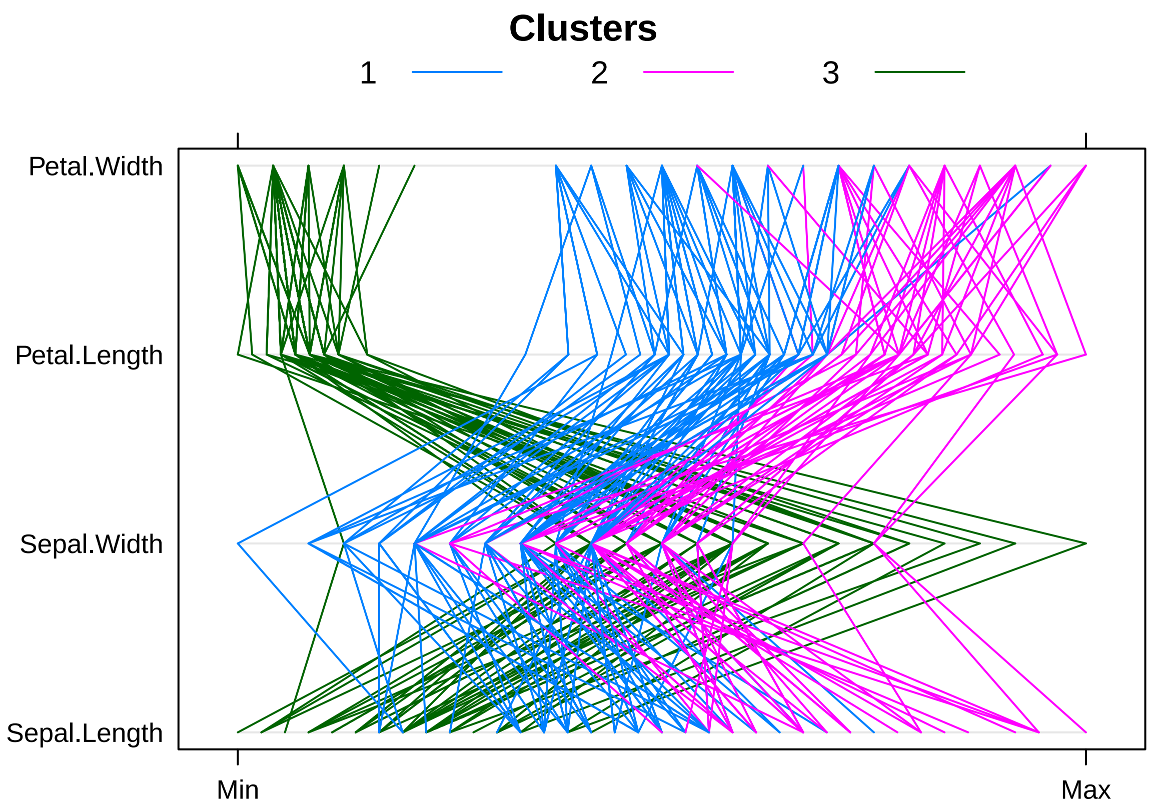
```
#mean(newiris$Sepal.Length[kc$cluster==1])   # example: compute average of variable for specific cluste
```

We can compare the parallel plot from the original data and the cluster. Observe that the patterns are quite similar.

```
# we can use parallel plot to see the effects, the auto.key plots a legend above the parallellines
parallelplot(iris[1:4],groups=iris$Species,main="Species",auto.key=list(space="top",columns=3,lines=T))
```



```
parallelplot(iris[1:4],groups=kc$cluster,main="Clusters",auto.key=list(space="top",columns=3,lines=T))
```



We can compute a cross tabulation to directly compare the Species with the cluster.

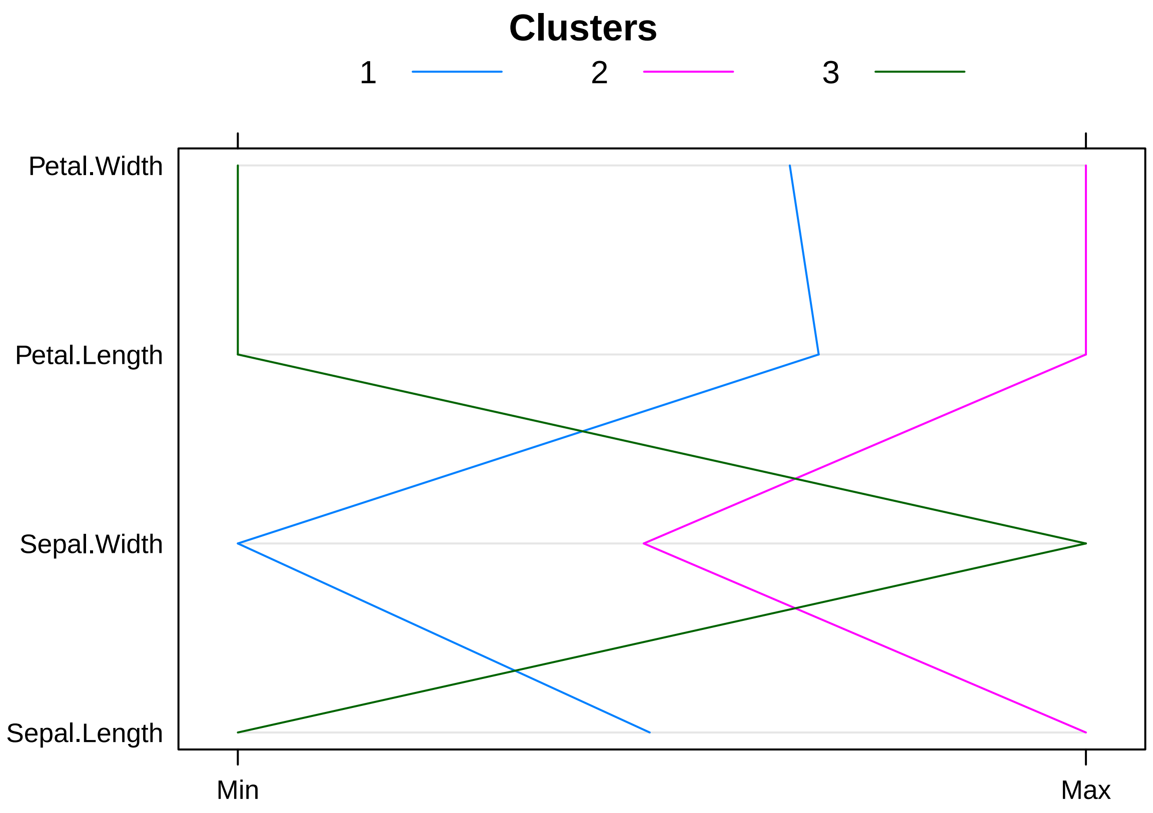
```r
# compare the Species label with the cluster result
table(iris$Species, kc$cluster)
```

```
##
##               1  2  3
##   setosa      0  0 50
##   versicolor 48  2  0
##   virginica  14 36  0
```

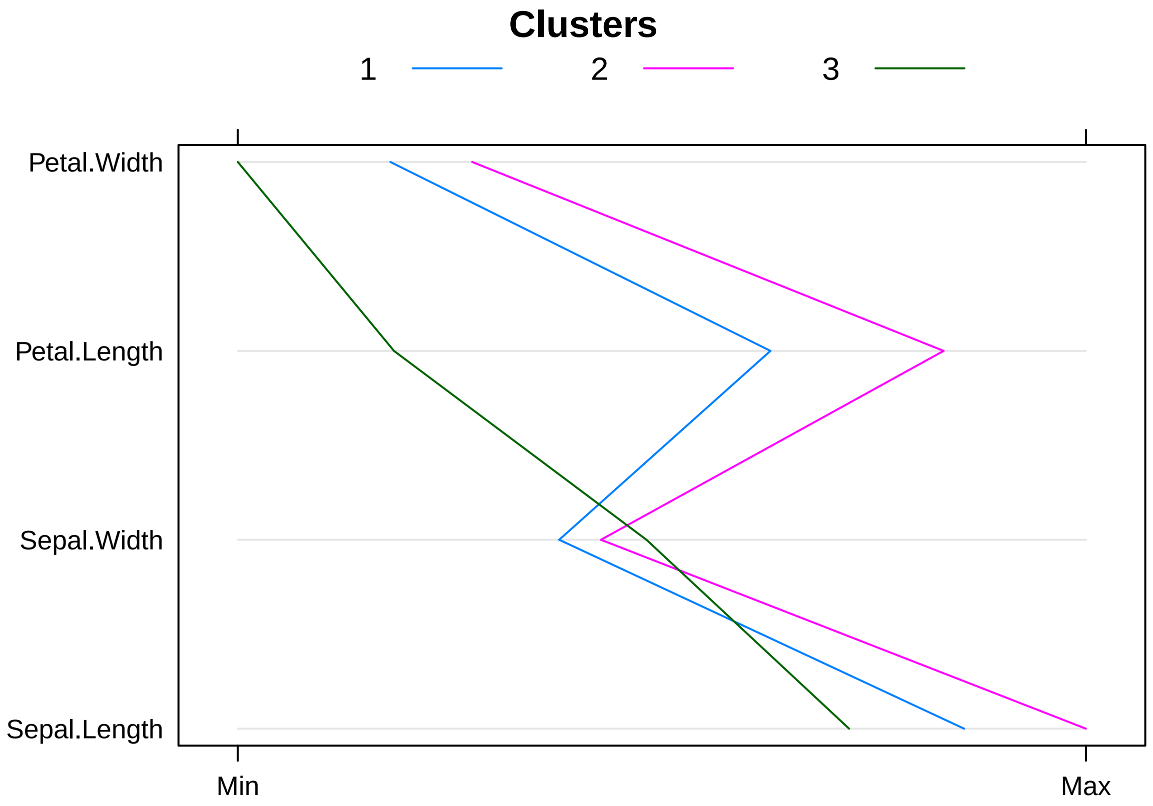
```r
#xtabs(~Species+kc$cluster,data=iris)  # another version of same table, xtabs using formula notation
```

A final thought is instead of printing a table of the centers we could plot the centers using `parallelplot`. A difficulty with the default setting is that each of the variables have very different means, but since each of the variables are rescaled we cannot compare scale differences across the variables. Instead we use the option `common.scale=TRUE` to tell `parallelplot` to use the minimum and maximum across all the variables. This allows us to see the differences across the variables more clearly.

```r
# we can also compare the cluster centroids
parallelplot(kc$centers,main="Clusters",auto.key=list(text=c("1","2","3"),space="top",columns=3,lines=T
```



```r
# notice the scales are relative for each variable, try it again it the same scale
parallelplot(kc$centers,main="Clusters",common.scale=TRUE,auto.key=list(text=c("1","2","3"),space="top"
```

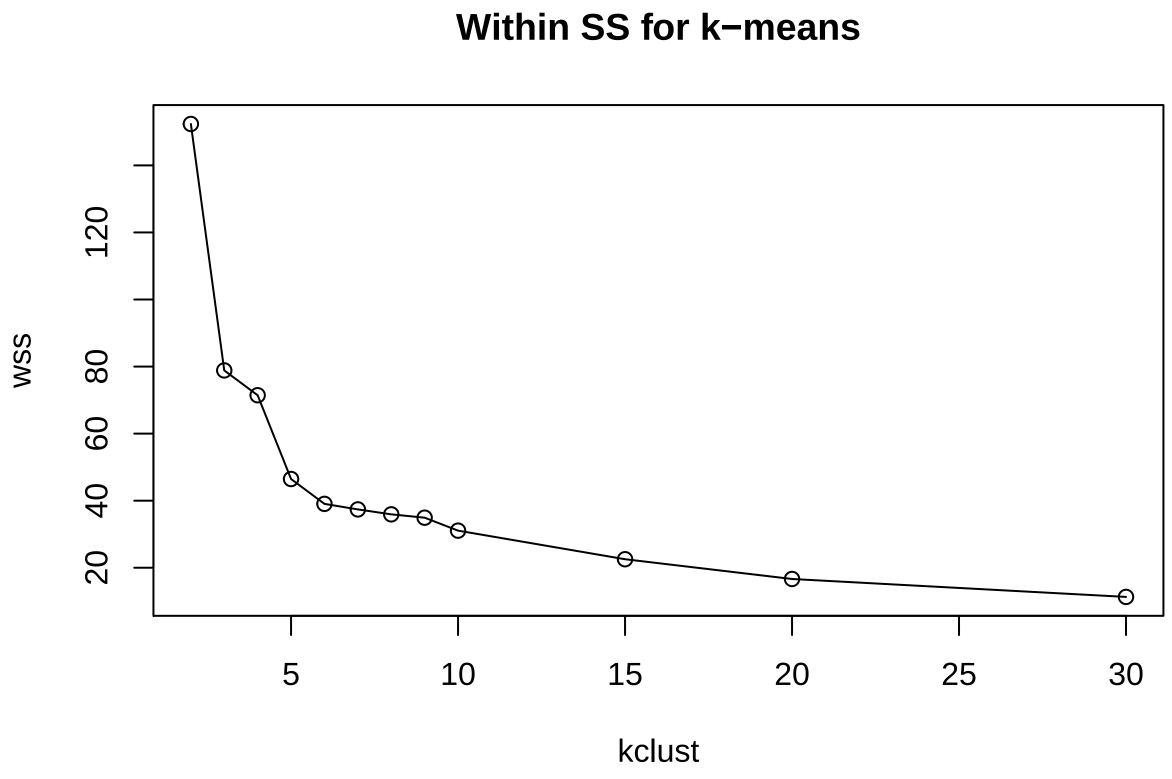


Build a screen plot to determine the number of clusters

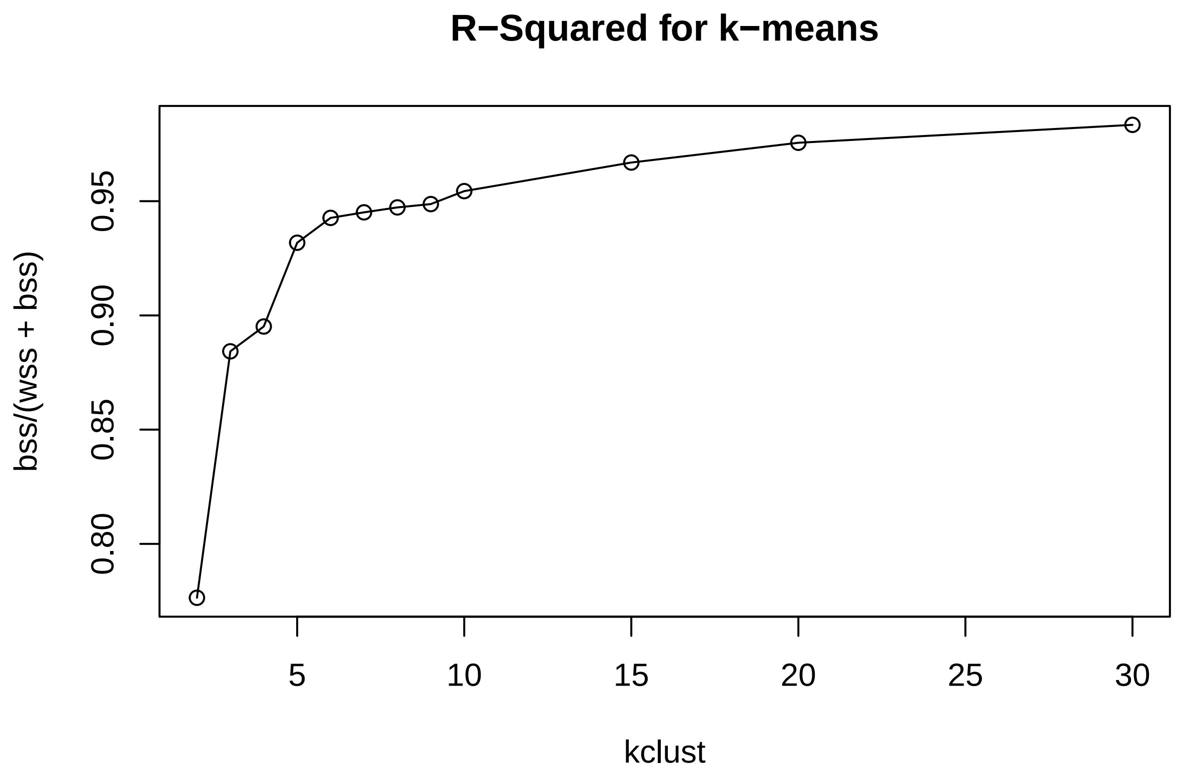
In the previous k-means analysis we used `k=3`. A justification is that we knew that there are three species, but in general there is no reason to think we know what the value of `k` to use beforehand. One suggestion is to create a scree plot to determine the number of clusters. In this plot we create many different solutions and then plot a metric like `wss`, `bss`, or a ratio of `bss` to `tss` (which gives us R-squared) to determine an appropriate value. As we look at the scree plots it appears that there are elbows at either k=3 or k=5. These points may be good candidate values of `k`.

```
################################################################################
### Build a scree plot to determine the number of clusters
################################################################################

# set the random number seed so the samples will be the same if regenerated
set.seed(34612)

# compute multiple cluster solutions
grpA2=kmeans(newiris,centers=2)
grpA3=kmeans(newiris,centers=3)
grpA4=kmeans(newiris,centers=4)
grpA5=kmeans(newiris,centers=5)
grpA6=kmeans(newiris,centers=6)
grpA7=kmeans(newiris,centers=7)
grpA8=kmeans(newiris,centers=8)
grpA9=kmeans(newiris,centers=9)
grpA10=kmeans(newiris,centers=10)
grpA15=kmeans(newiris,centers=15)
grpA20=kmeans(newiris,centers=20)
grpA30=kmeans(newiris,centers=30)
```

```
# compute between and within SS
kclust=c(2:10,15,20,30)
bss=c(grpA2$betweenss,
      grpA3$betweenss,grpA4$betweenss,grpA5$betweenss,grpA6$betweenss,
      grpA7$betweenss,grpA8$betweenss,grpA9$betweenss,grpA10$betweenss,
      grpA15$betweenss,grpA20$betweenss,grpA30$betweenss)
wss=c(grpA2$tot.withinss,
      grpA3$tot.withinss,grpA4$tot.withinss,grpA5$tot.withinss,grpA6$tot.withinss,
      grpA7$tot.withinss,grpA8$tot.withinss,grpA9$tot.withinss,grpA10$tot.withinss,
      grpA15$tot.withinss,grpA20$tot.withinss,grpA30$tot.withinss)

# plot the results and look for the "Hockey-Stick" effect
par(mfrow=c(1,1))
plot(kclust,wss,type="l",main="Within SS for k-means")  # Within SS is variation of errors
points(kclust,wss)
```



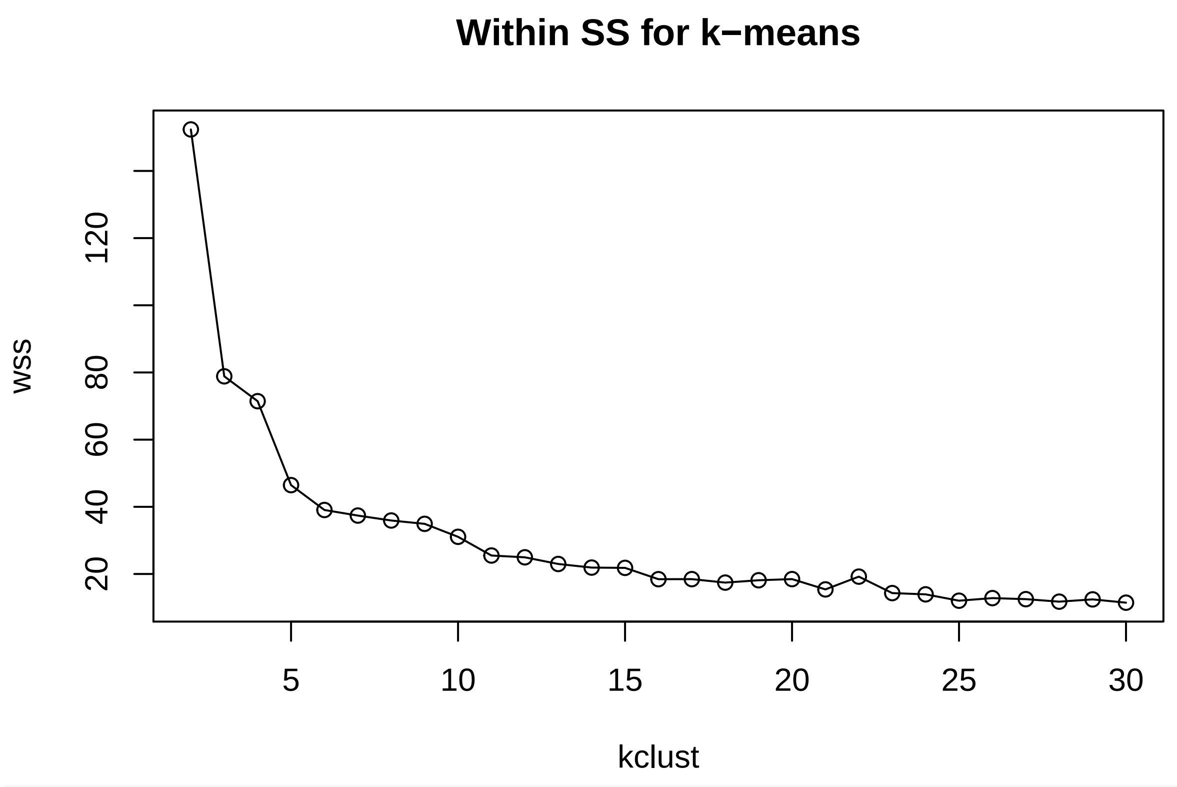
```
plot(kclust,bss/(wss+bss),type="l",main="R-Squared for k-means")  # R-Squared is ratio of explained var
points(kclust,bss/(wss+bss))
```



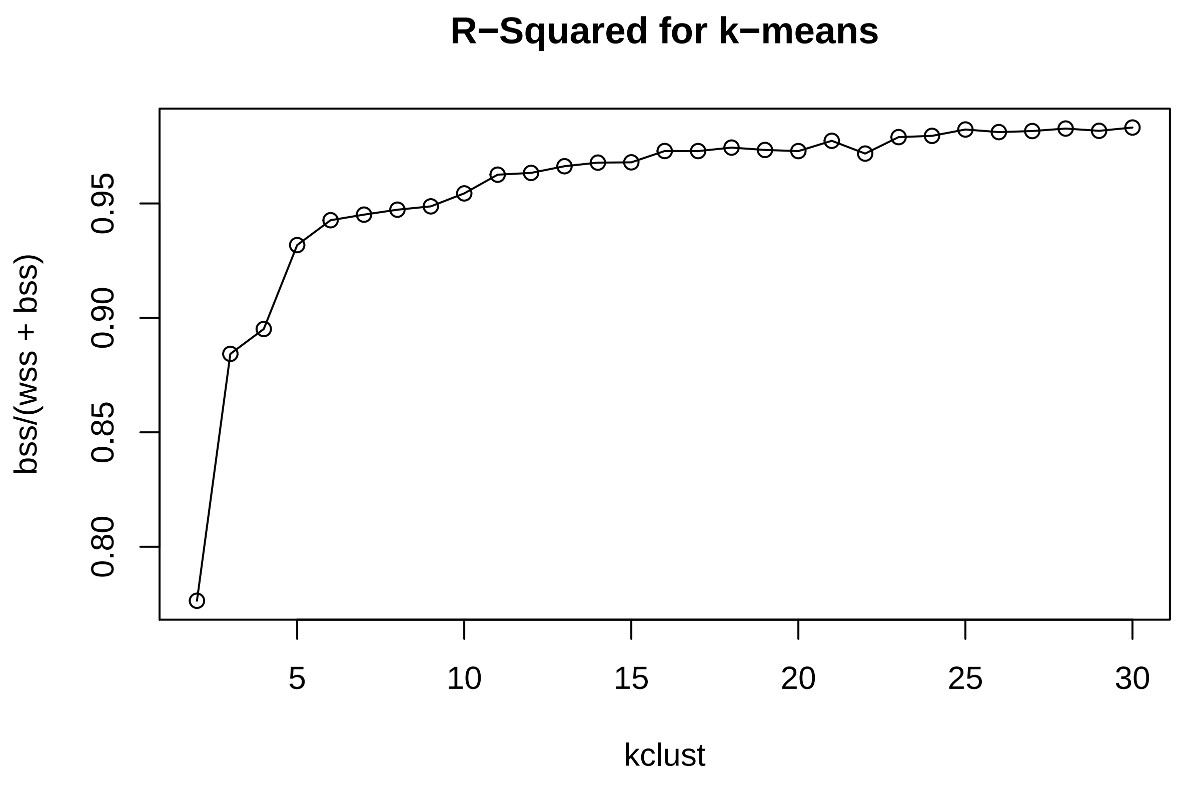
Screen plot using a for loop

To improve the efficiency of our code we could employ a for loop. Notice that `grpQ` is a list that has the same length as the number of cluster values that we evaluate. Each position of `grpQ` stores our values of `kmeans`. In other words we have a list whose elements are lists.

```
###############################################################################
### Build a scree plot to determine the number of clusters
### Alternative method using a for loop
###############################################################################

# compute multiple cluster solutions
kclust=2:30                                  # create a vector of k values to try
nclust=length(kclust)                        # number of kmeans solutions to compute
bss=wss=rep(0,nclust)                        # initialize vectors bss and wss to zeroes
set.seed(34612)                              # set the seed so we can repeat the results
grpQ=as.list(rep(NULL,nclust))               # create empty list to save results
# compute SS for each cluster
for (i in 1:nclust) {
   grpQ[[i]]=kmeans(newiris,kclust[i])  # compute kmeans solution, !! try adding nstart=100 to try many
   wss[i]=grpQ[[i]]$tot.withinss        # save the within SS
   bss[i]=grpQ[[i]]$betweenss           # save the between SS
}

# plot the results and look for the "Hockey-Stick" effect
par(mfrow=c(1,1))
plot(kclust,wss,type="l",main="Within SS for k-means")  # Within SS is variation of errors
points(kclust,wss)
```



```
plot(kclust,bss/(wss+bss),type="l",main="R-Squared for k-means")  # R-Squared is ratio of explained var
points(kclust,bss/(wss+bss))
```



Using nstart option to improve kmeans

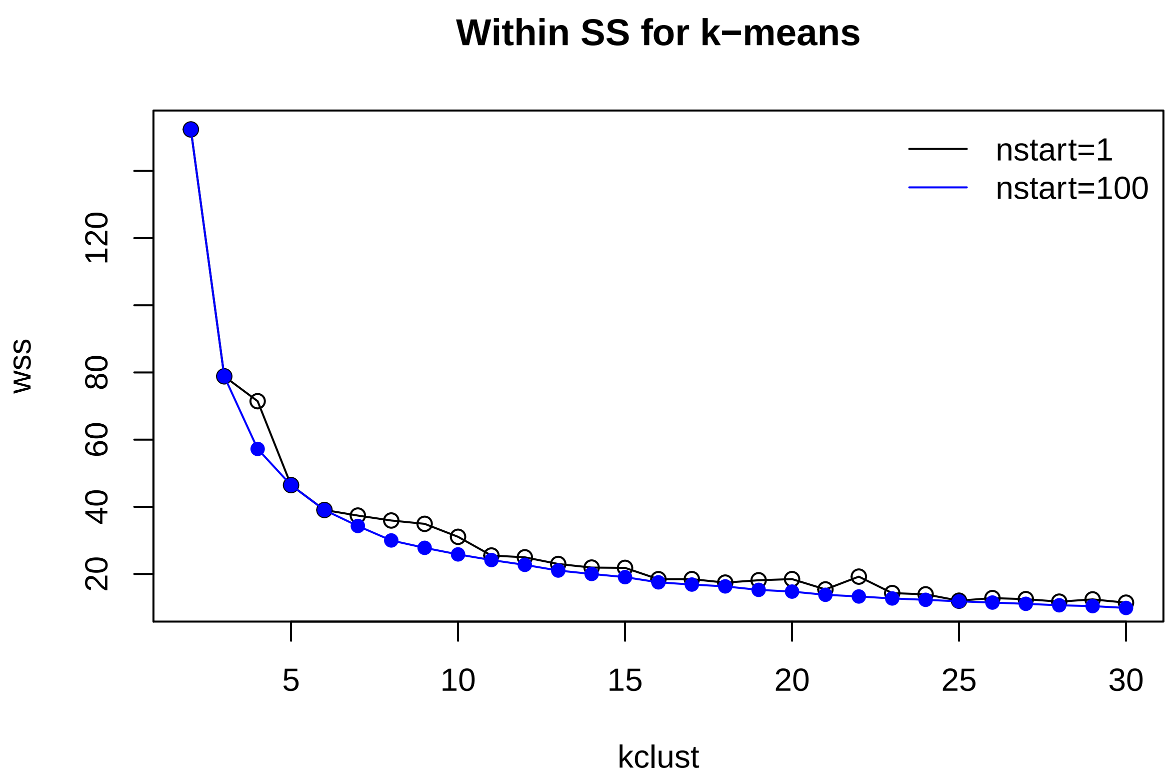
The default for `kmeans` is to randomly draw a set of `k` points to start as the initial centroids. Unfortunately there is no guarantee that this random draw will give us an optimal starting point. One suggestion is to try to run `kmeans` with many random starting values. In this example we modify the previous code with the option `nstart=100` to `kmeans`. The 100 value means that 100 sets of random initial values are drawn, but only the best one is returned (i.e., the one that returns the smallest `wss` value). In the following screen plot you can notice that our solutions are always better than with a single starting value. It smooths out the random bumps that we might see due to an unlikely initial cluster assignment. If you use `nstart` make sure that you include it in all `kmeans` evaluations.

We still can observe that there is a substantial drop in `wss` for `k=3` which suggests that this value of `k=4` would be good candidates. However, the decrease at `k=5` does not look as good as it did in the previous evaluation.

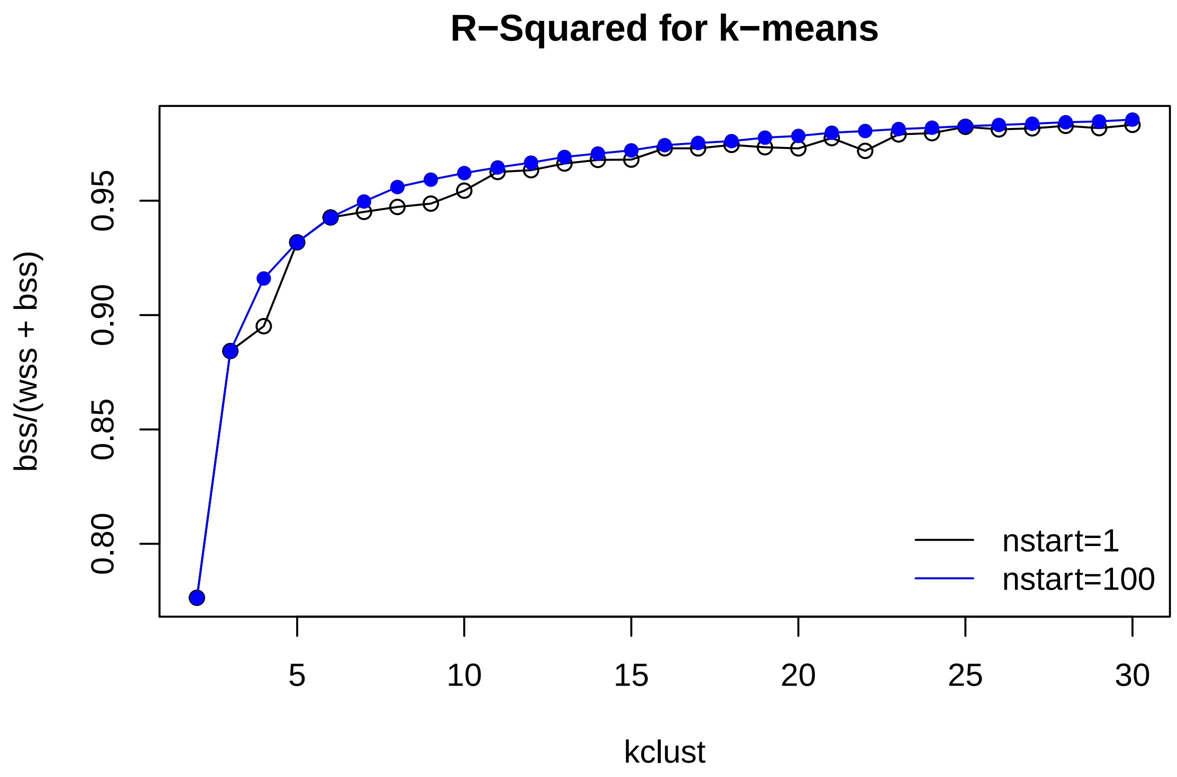
```
###############################################################################
### Build a scree plot to determine the number of clusters
### Use the for loop, but instead of choosing one initial starting point for
### each value of k, let's try nstart=100.  The nstart option tells kmeans
### to randomly choose many randomly choosen values.  kmeans will only return
### the "best" solution (or the one that minimizes the SSE)
###############################################################################

# compute multiple cluster solutions
kclust=2:30                          # create a vector of k values to try
nclust=length(kclust)                # number of kmeans solutions to compute
bss.nstart=wss.nstart=rep(0,nclust) # initialize vectors bss and wss to zeroes
set.seed(34612)                      # set the seed so we can repeat the results
grpQ=as.list(rep(NULL,nclust))       # create empty list to save results
# compute SS for each cluster
for (i in 1:nclust) {
   grpQ[[i]]=kmeans(newiris,kclust[i],nstart=100)  # compute kmeans solution with nstart value, nstart=
   wss.nstart[i]=grpQ[[i]]$tot.withinss        # save the within SS
   bss.nstart[i]=grpQ[[i]]$betweenss           # save the between SS
}
```

```
## Warning: did not converge in 10 iterations
```

```
# plot the results and look for the "Hockey-Stick" effect
par(mfrow=c(1,1))
# plot the old solution
plot(kclust,wss,type="l",main="Within SS for k-means")  # Within SS is variation of errors
points(kclust,wss)
# overlay our improved solutions
lines(kclust,wss.nstart,col="blue")
points(kclust,wss.nstart,col="blue",pch=16)
legend("topright",c("nstart=1","nstart=100"),lty=1,col=c("black","blue"),bty="n")
```



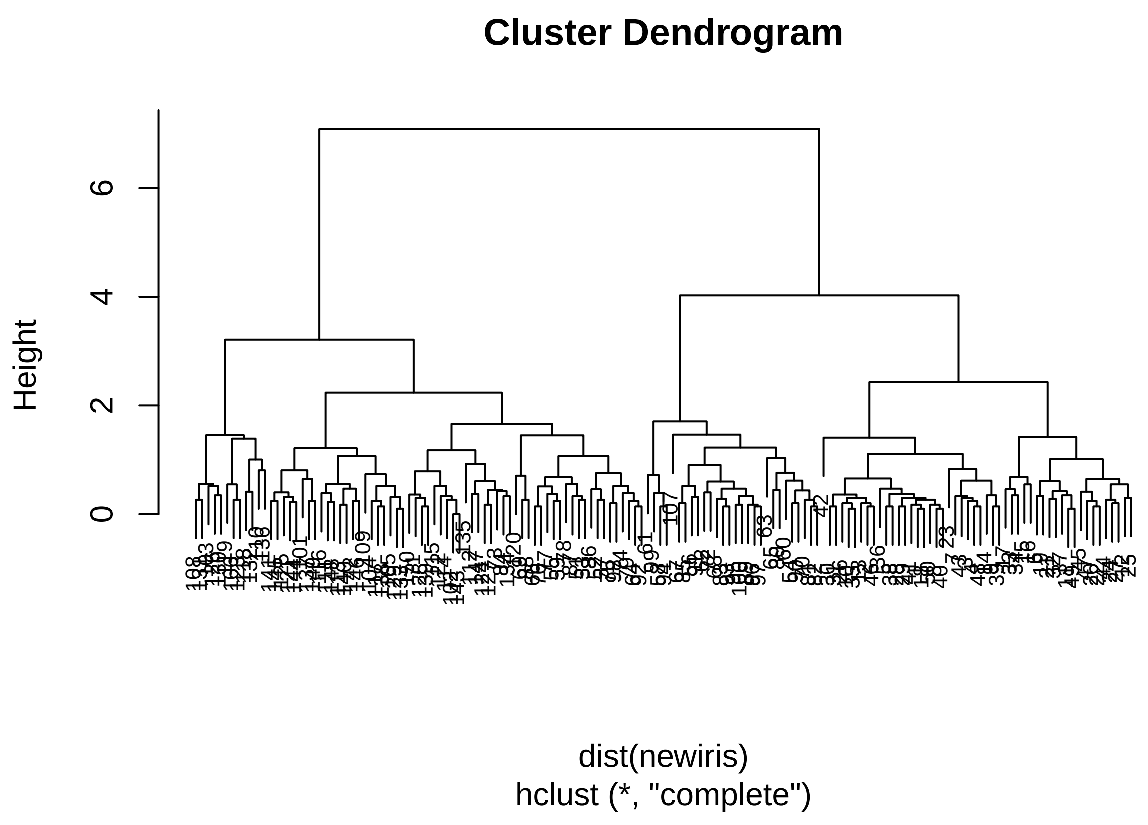
```
# plot the old solution for r-squared
plot(kclust,bss/(wss+bss),type="l",main="R-Squared for k-means")  # R-Squared is ratio of explained var
points(kclust,bss/(wss+bss))
# overlay our improved solutions
lines(kclust,bss.nstart/(wss.nstart+bss.nstart),col="blue")
points(kclust,bss.nstart/(wss.nstart+bss.nstart),col="blue",pch=16)
legend("bottomright",c("nstart=1","nstart=100"),lty=1,col=c("black","blue"),bty="n")
```



Optional Hierarchical Cluster Analysis

Instead of grouping observations into K clusters, we build up the cluster by finding individual observations that are most like another observation. Like our previous grouping you can turn a hierarchical cluster into K clusters by deciding what is the distance at which you want to define your clusters. To do so we use the `cutree` command to turn our hierarchical cluster into a divisive clustering scheme.

```
###############################################################################
### (Optional) Create a hierarchical cluster analysis
###############################################################################

# try a hierarchical cluster on the flowers
hc=hclust(dist(newiris),method="complete")
plot(hc,cex=.7)
```



```
hc3id = cutree(hc,k=3)  # divide the tree into three clusters
print(hc3id)  # these are the clusters if with divide into three clusters
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 3 2 3 2 3 2 3 3 3 3 2 3 2 3 3 2 3
##  [71] 2 3 2 2 2 2 2 2 2 3 3 3 3 2 3 2 2 2 3 3 3 2 3 3 3 3 3 2 3 3 2 2 2 2 2
## [106] 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [141] 2 2 2 2 2 2 2 2 2 2
```

```
table(hc3id,kc$cluster)  # notice that the division is similar
```

```
##
## hc3id  1  2  3
##     1  0  0 50
##     2 34 38  0
##     3 28  0  0
```

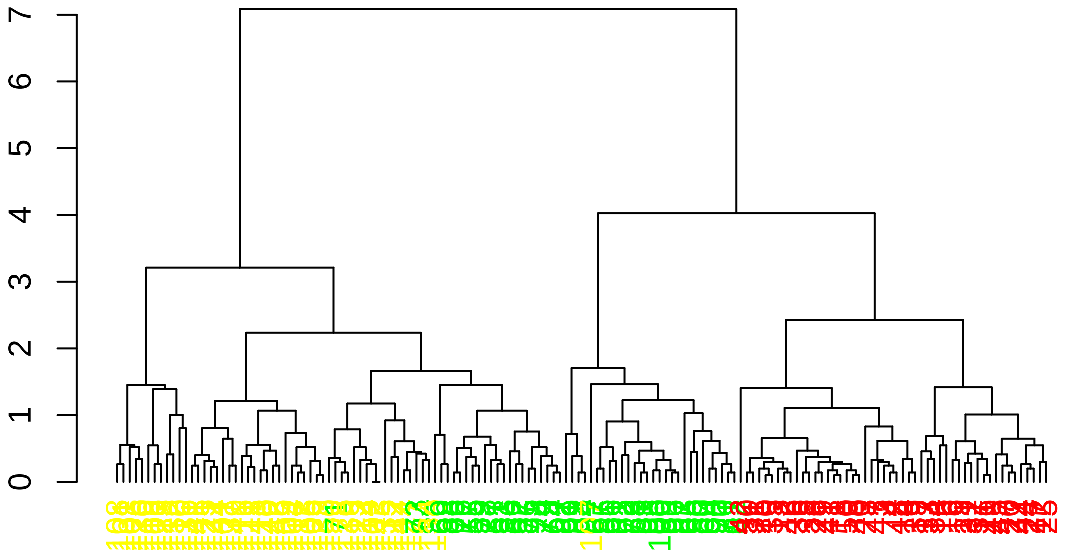
```
# if we want to color the labels according to the species
if (!require(dendextend)) {install.packages("dendextend"); library(dendextend)}  # need new package
```

```
## Loading required package: dendextend

## Registered S3 methods overwritten by 'ggplot2':
##   method         from
##   [.quosures     rlang
##   c.quosures     rlang
##   print.quosures rlang

##
## ---------------------
## Welcome to dendextend version 1.12.0
## Type citation('dendextend') for how to cite the package.
##
## Type browseVignettes(package = 'dendextend') for the package vignette.
## The github page is: https://github.com/talgalili/dendextend/
##
## Suggestions and bug-reports can be submitted at: https://github.com/talgalili/dendextend/issues
## Or contact: <tal.galili@gmail.com>
##
##  To suppress this message use:  suppressPackageStartupMessages(library(dendextend))
## ---------------------
```

```
##
## Attaching package: 'dendextend'
```

```
## The following object is masked from 'package:stats':
##
##     cutree
```

```
dend=as.dendrogram(hc)  # create a new object that stores the dendogram
colorCodes = c(setoas="red",versicolor="green",virginica="yellow")  # the elements of the list correspo
labels_colors(dend) = colorCodes[iris$Species][order.dendrogram(dend)]  # notice we have to reorder the
plot(dend)  # let's plot the denodogram again
```



```
# for an extended analysis see https://cran.r-project.org/web/packages/dendextend/vignettes/Cluster_Ana
```