Stats_102B_HW_5_Charles_Liu

Charles Liu (304804942)

5/28/2020

Homework 5 Requirements

There is no separate instruction file. This file is the instructions and you will modify it for your submission.

You will submit two files.

The files you submit will be:

1. `102b_hw5_output_First_Last.Rmd` Take this R Markdown file and make the necessary edits so that it generates the requested output.
2. `102b_hw5_output_First_Last.pdf` OR `102b_hw_5_output_First_Last.html` Your output file. This can be a PDF or an HTML file. This is the primary file that will be graded. **Make sure all requested output is visible in the output file.**

Failure to submit all files will result in an automatic 40 point penalty.

Academic Integrity

Modifying the following statement with your name.

"By including this statement, I, Charles Liu, declare that all of the work in this assignment is my own original work. At no time did I look at the code of other students nor did I search for code solutions online. I understand that plagiarism on any single part of this assignment will result in a 0 for the entire assignment and that I will be referred to the dean of students."

If you collaborated verbally with other students, please also include the following line to credit them.

"At no point did I show another student my code, nor did I look at another student's code." I only utilized the sweep() function from the TA, who taught us how it works during discussion. Homework questions and text copyright Miles Chen. For personal use only. Do not distribute. Do not post or share your solutions.

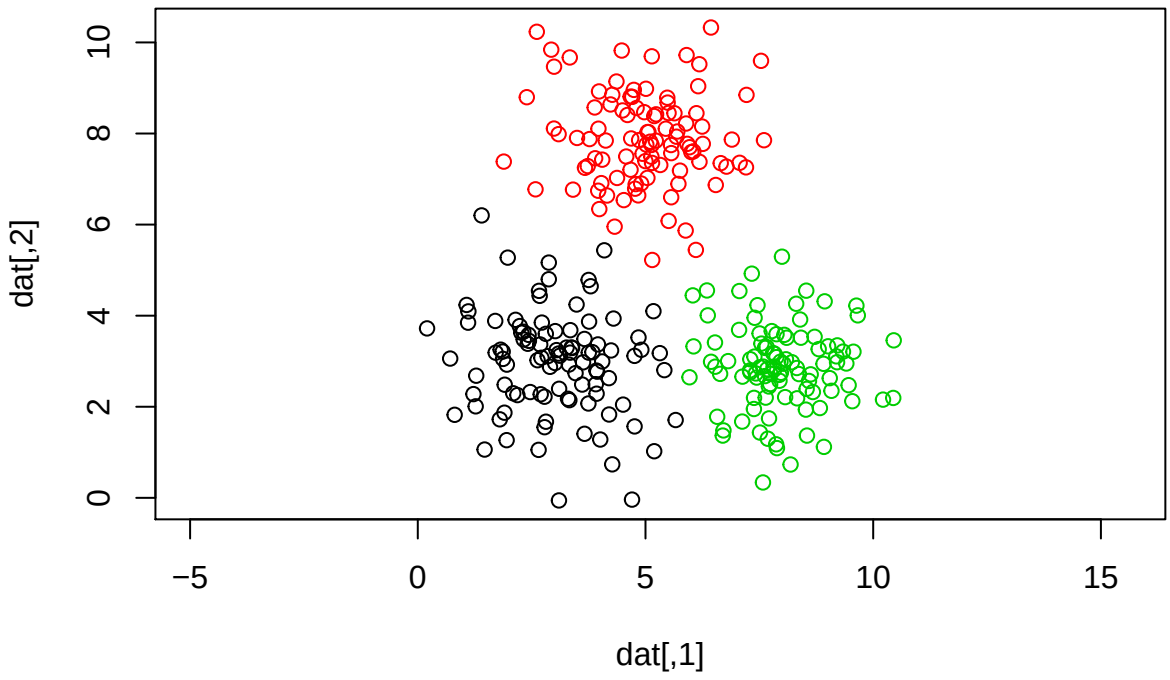
Part 1: K-means Clustering

Read section 6.2 in the text and https://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm

K-means clustering is a clustering method. The algorithm can be described as follows:

0) Determine how many (k) clusters you will search for.
1) Randomly assign points in your data to each of the clusters.
2) Once all values have been assigned to a cluster, calculate the means or the centroid of the values in each cluster.
3) Reassign values to clusters by associating values in the data set to the nearest (Euclidean distance) centroid.
4) Repeat steps 2 and 3 until convergence. Convergence occurs when no values are reassigned to a new cluster.

```
# Don't change this code. It will be used to generate the data.
set.seed(2020)
RNGkind(sample.kind = "Rejection")
library(mvtnorm)
cv <- matrix(c(1, 0, 0, 1), ncol = 2)
j <- rmvnorm(100, mean = c(3, 3), sigma = cv)
k <- rmvnorm(100, mean = c(5, 8), sigma = cv)
l <- rmvnorm(100, mean = c(8, 3), sigma = cv)
dat <- rbind(j, k, l)
true_groups <- as.factor(c(rep("j", 100), rep("k", 100), rep("l", 100)))
plot(dat, col=true_groups, asp = 1)
```



Task 1

Write code to perform k-means clustering on the values in the matrix `dat`.

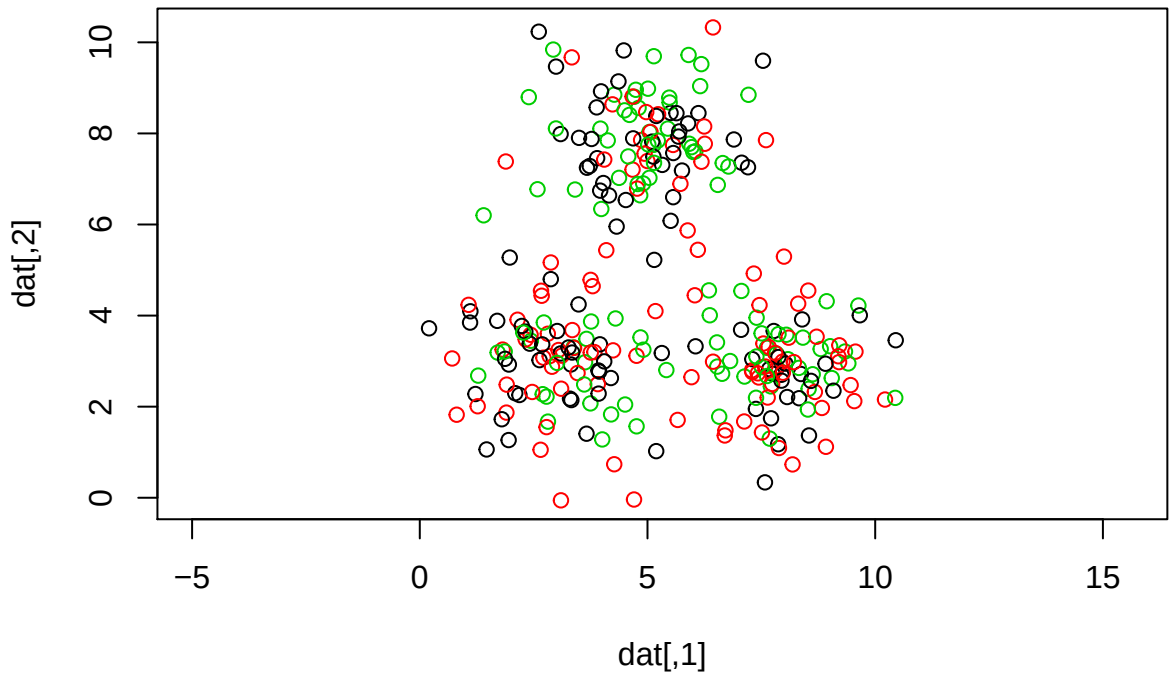
The true group labels are provided in the vector `true_groups`. Of course, you can't use that until the very end where you will perform some verification.

Requirements:

1) So everyone will get consistent results, I have performed the initial assignment of points to clusters.
2) With each iteration, plot the data, colored by their current groupings, and the updated means.
3) Convergence is reached when group assignments no longer change. Your k-means clustering algorithm should reach convergence fairly quickly.
4) Print out a 'confusion' matrix showing how well the k-means clustering algorithm grouped the data vs the 'true labels.'

One suggestion is to write a function that will calculate the distances from a point to each of the three means. You can apply this function to the matrix of points (n x 2) and get back another matrix of distances (n x 3) where the columns are distance to centroid A, dist to centroid B, dist to centroid C.

```
# do not modify this code
set.seed(2020)
assignments <- factor(sample(c(1, 2, 3), 300, replace = TRUE)) # initial groupings that you will need t
plot(dat, col = assignments, asp = 1)  # initial plot
```




```
# Using piping for conversion
library(tidyverse)
```

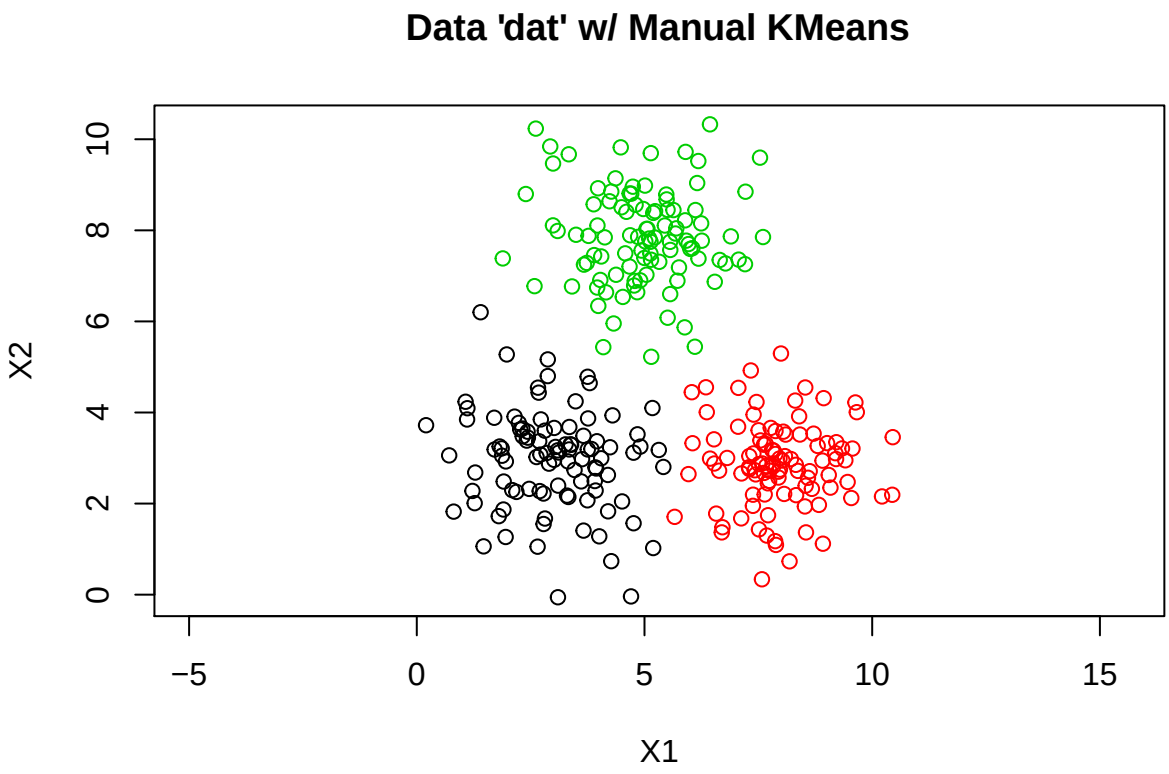
```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.0     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## Warning: package 'ggplot2' was built under R version 3.6.3
```

```
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
# dat has 600 observations [300x2], so set it up with names (X1,X2) and as dataframe
colnames(dat) <- c("X1","X2")
dat <- data.frame(dat, assignments)

# Distance Formula
distances <- function(point, means){
  dist_to_centroid_A <- (point[,1]-means[1,1])^2 + (point[,2]-means[1,2])^2
  dist_to_centroid_B <- (point[,1]-means[2,1])^2 + (point[,2]-means[2,2])^2
  dist_to_centroid_C <- (point[,1]-means[3,1])^2 + (point[,2]-means[3,2])^2
  return(matrix(c(A = dist_to_centroid_A, B = dist_to_centroid_B,
           C = dist_to_centroid_C), nrow(point), 3))
}

# Compute the while() loop
converged = FALSE
while(!converged){
  centroids <- dat %>%
    group_by(assignments) %>%
    summarise(x1=mean(X1), x2=mean(X2))
  centroids <- as.data.frame(centroids)[,-1]
  new_assignments <- factor(apply(distances(dat, centroids), 1, which.min))
  if(sum(abs(as.integer(dat[,3]) - as.integer(new_assignments))) == 0) {
    converged <- TRUE
  }
  else {
    dat[,3] <- new_assignments
  }
}

# Plot
plot(dat[,-3], col=dat[,3], asp=1, main="Data 'dat' w/ Manual KMeans")
```

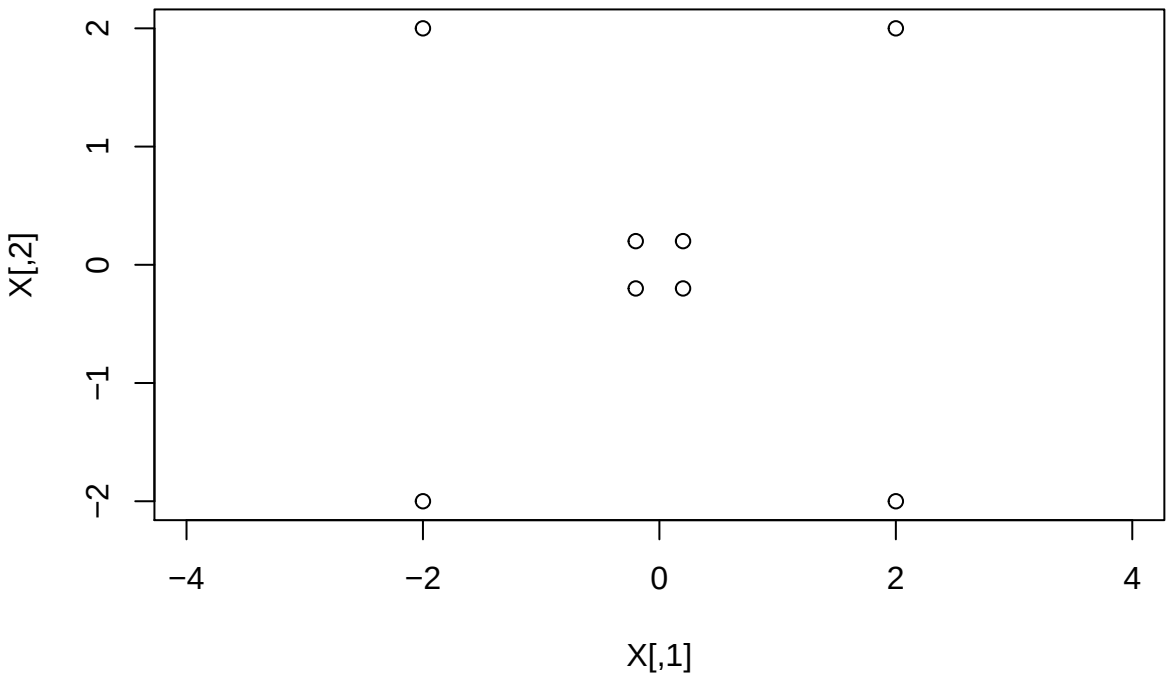


Part 2: Kernelized K-means Clustering

Read section 6.2.3 in the text.

Our very simple data. 8 points. 4 near the origin. 4 that are spaced away

```
X <- matrix(c(
  0.2,  0.2,
  0.2, -0.2,
 -0.2,  0.2,
 -0.2, -0.2,
    2,  2,
    2, -2,
   -2, -2,
   -2,  2),
  byrow = TRUE,
  ncol = 2)
plot(X, asp = 1)
```



To perform Kernelized K-means clustering, we will use Kernel Functions. A Kernel transforms our 2D point into another point in a higher dimension and returns the inner (dot) product of pair of points in the higher dimension. (helpful to review what a dot product signifies: https://youtu.be/LyGKycYT2v0)

The function that tranforms the data from 2D to the higher dimension is called phi ($\phi(\mathbf{x})$). For this example, Phi will transform our 2D data into 3D data.

We transform a 2D coordinate $\phi(x_1, x_2)$ to the 3d coordinate: $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$ (equivalent to using a polynomial kernel with C = 0 and gamma = 2) (See: https://en.wikipedia.org/wiki/Polynomial_kernel)

```
phi <- function(x){
  c(x[1]^2, x[2]^2, sqrt(2)*x[1]*x[2]) # change this
}

transformed <- t(apply(X, 1, FUN = phi))
colnames(transformed) <- c("X1^2", "X2^2", "(2*X1*X2)^0.5")
transformed
```

```
##      X1^2 X2^2 (2*X1*X2)^0.5
## [1,] 0.04 0.04    0.05656854
## [2,] 0.04 0.04   -0.05656854
## [3,] 0.04 0.04   -0.05656854
## [4,] 0.04 0.04    0.05656854
## [5,] 4.00 4.00    5.65685425
## [6,] 4.00 4.00   -5.65685425
## [7,] 4.00 4.00    5.65685425
## [8,] 4.00 4.00   -5.65685425
```

Task 2a:

Instead of random assignments, place all points into the same cluster, with the exception of the first value, which will be placed in the second cluster.

Using the coordinates of the data projected in 3 dimensions, calculate the (3-dimensional) centroids of each cluster.

```
assignments <- factor(c(2, rep(1,7)), levels=c(1, 2))
colnames(transformed) <- c("X1", "X2", "X3")
df <- data.frame(transformed, assignments)
centroids <- df %>%
  group_by(assignments) %>%
  summarise(x1=mean(X1), x2=mean(X2), x3=mean(X3))
centroids <- as.data.frame(centroids)[,-1]
centroids
```

```
##         x1       x2          x3
## 1 2.302857 2.302857 -0.00808122
## 2 0.040000 0.040000  0.05656854
```

Task 2b:

Measure the squared Euclidean distance from each of the 8 values to both centroids. Produce a matrix that records these distances. It will be an 8 x 2 matrix. The first column will be the distance from each point to the first centroid (center of the 7 points in cluster 1). The second column will be the distances from each point to the second centroid (this cluster has only the first coordinate in it).

```
distances <- function(point, means){
  dist_to_centroid_A <- (point[,1]-means[1,1])^2 + (point[,2]-means[1,2])^2 +
    (point[,3]-means[1,3])^2
  dist_to_centroid_B <- (point[,1]-means[2,1])^2 + (point[,2]-means[2,2])^2 +
    (point[,3]-means[2,3])^2
  return(matrix(c(A = dist_to_centroid_A, B = dist_to_centroid_B),
                nrow(point), 2))
}

dist_overall <- distances(df, centroids)
colnames(dist_overall) <- c("X1", "X2")

dist_overall
```

```
##            X1      X2
## [1,] 10.24522  0.0000
## [2,] 10.24340  0.0128
## [3,] 10.24340  0.0128
## [4,] 10.24522  0.0000
## [5,] 37.85208 62.7264
## [6,] 37.66922 64.0064
## [7,] 37.85208 62.7264
## [8,] 37.66922 64.0064
```

Task 2c:

Assign each point to the cluster whose centroid is nearer. Update your Z values accordingly.

```r
# Set up the phi_assignments and the z1 & z2
assignments_phi <- apply(dist_overall, 1, which.min)
z1 <- ifelse(assignments_phi == 1, 1, 0)
z2 <- ifelse(assignments_phi == 2, 1, 0)
z_overall <- cbind(z1, z2)

z_overall
```

```
##      z1 z2
## [1,]  0  1
## [2,]  0  1
## [3,]  0  1
## [4,]  0  1
## [5,]  1  0
## [6,]  1  0
## [7,]  1  0
## [8,]  1  0
```

This takes us through just one iteration. Normally, we would iterate between the assignment step and the centroid calculation step. However, for this data set, one iteration is enough to properly cluster the points.

Using Kernels

The use of a kernel function allows us to bypass the need to calculate the coordinates of each point in the higher-dimensional transformed space. We do not even need to know the function Phi. The Kernel function will simply return the dot product (a scalar value) of two points in the transformed space.

With some algebraic manipulation (show in the textbook), we can get the equation for the distances from each point to the cluster centroids in terms of the Kernel function, completely avoiding the use of phi.

Task 3a:

To reduce computational time, we will create a matrix of all the products resulting from applying the Kernel function to every pair of points. (We find $K(\mathbf{x}_m, \mathbf{x}_m)$ for all pairs of m, r). This is similar to creating a multiplication table, except we are using the Kernel function. The calculations in the future will use these Kernel function products, and rather than having to redo the calculation each time, we will simply look up the product in the table.

$$K(\mathbf{x}_m, \mathbf{x}_r) = (\mathbf{x}_m^T \mathbf{x}_r)^2$$

The Kernel matrix is equivalent to finding the inner products of the transformed data. However, do not use the transformed data, as the whole point of this part of the exercise is to show that we can achieve the same thing without transforming our data. Find the kernel results using the kernel function itself.

```r
# Initialize the for()loop
N = nrow(X)
K = matrix(0, nrow=N, ncol=N)

# Start the for() loop
for(m in 1:N) {
  for(r in 1:N) {
    K[m,r] = (t(X[m,]) %*% X[r,])^2
  }
}
```

```
K
```

```
##        [,1]   [,2]   [,3]   [,4]  [,5]  [,6]  [,7]  [,8]
## [1,] 0.0064 0.0000 0.0000 0.0064  0.64  0.00  0.64  0.00
## [2,] 0.0000 0.0064 0.0064 0.0000  0.00  0.64  0.00  0.64
## [3,] 0.0000 0.0064 0.0064 0.0000  0.00  0.64  0.00  0.64
## [4,] 0.0064 0.0000 0.0000 0.0064  0.64  0.00  0.64  0.00
## [5,] 0.6400 0.0000 0.0000 0.6400 64.00  0.00 64.00  0.00
## [6,] 0.0000 0.6400 0.6400 0.0000  0.00 64.00  0.00 64.00
## [7,] 0.6400 0.0000 0.0000 0.6400 64.00  0.00 64.00  0.00
## [8,] 0.0000 0.6400 0.6400 0.0000  0.00 64.00  0.00 64.00
```

Task 3c:

Use equation 6.3 from the textbook to calculate the kernelized distance matrix. The results of this should be equivalent to your results in task 2b.

Note that the calculation of this distance matrix did not require the use of the transformation function phi at all.

```
# Set up the z1 and z2
z1 <- as.integer(assignments == 1)
z2 <- as.integer(assignments == 2)

# Initialize the for() loop
dist1 <- rep(0, N)
dist2 <- rep(0, N)

# Start for() loop
for(n in 1:N) {
  dist1[n] = K[n,n] - 2 / sum(z1) * sum(z1*K[n,]) + 1/(sum(z1)^2) *
    sum((z1 %*% t(z1))*K)
  dist2[n] = K[n,n] - 2 / sum(z2) * sum(z2*K[n,]) + 1/(sum(z2)^2) *
    sum((z2 %*% t(z2))*K)
}

kernel_distance <- cbind(dist1, dist2)
kernel_distance
```

```
##         dist1   dist2
## [1,] 10.24522  0.0000
## [2,] 10.24340  0.0128
## [3,] 10.24340  0.0128
## [4,] 10.24522  0.0000
## [5,] 37.85208 62.7264
## [6,] 37.66922 64.0064
## [7,] 37.85208 62.7264
## [8,] 37.66922 64.0064
```

Task 3d:

Assign each point to the cluster whose centroid is nearer. Update your Z matrix (the assignment matrix) accordingly.

```r
# Set up the kernel_assignments and the z1 & z2
kernel_assignments <- apply(kernel_distance, 1, which.min)
z1 <- as.integer(kernel_assignments == 1)
z2 <- as.integer(kernel_assignments == 2)
z_overall <- cbind(z1, z2)

z_overall
```

```
##      z1 z2
## [1,]  0  1
## [2,]  0  1
## [3,]  0  1
## [4,]  0  1
## [5,]  1  0
## [6,]  1  0
## [7,]  1  0
## [8,]  1  0
```

This takes us through just one iteration. Normally, we would iterate between the assignment step and the centroid calculation step. However, for this data set, one iteration is enough to properly cluster the points.

More information about the Kernel

In this exercise, we used a simple 2D to 3D transformation function and its corresponding Kernel function. In real life, the most commonly used kernel function is the Radial Basis Function (RBF) Kernel, also called a Gaussian Kernel Function. https://en.wikipedia.org/wiki/Radial_basis_function_kernel

Like other kernel functions, the RBF kernel returns the inner product of a pair of points in a transformed space. The interesting thing is that the transformed space is infinite dimensional. It uses the (infinite) Taylor Expansion of the exponential function. http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf

Thus, getting the kernelized distances to the cluster centroids is only possible using the Kernel function. We would not actually be able to compute the transformation into infinite dimensional space.

Part 3: EM Algorithm

The Expectation-Maximization algorithm is an iterative algorithm for finding maximum likelihood estimates of parameters when some of the data is missing. In our case, we are trying to estimate the model parameters (the means and sigma matrices) of a mixture of multi-variate Gaussian distributions, but we are missing the group information of the data points. That is, we do not know if a value belongs to group A, group B, or group C.

The general form of the EM algorithm consists of alternating between an expectation step (E) and a maximization step (M).

In the expectation step, a function is calculated. The function is the expectation of the log-likelihood of the joint distribution of the data X along with the missing values of Z (cluster assignments) given the values of X under the current estimates of θ. (θ is the umbrella parameter that encompasses the means and sigma matrices)

In the maximization step, the values of θ are found that will maximize this expected log-likelihood.

We can take advantage of the fact that the solution to the maximization step can often be found analytically (versus having to search for it via a computational method.) For example, the estimate of the mean that maximizes the likelihood of the data is just the sample mean.

EM Algorithm for Gaussian Mixtures

See EM Algorithm Notes Handout. Section 6.3.3 in the text also covers the EM algorithm but uses different notation.

This (brilliant) algorithm can be applied to perform clustering of Gaussian mixtures (among many other applications) in a manner similar to the k-means algorithm and Bayes classifier. A key difference between the k-means algorithm and the EM algorithm is that the EM algorithm is probabilistic. The k-means algorithm assigned a value to the group with the nearest mean. The EM algorithm calculates the probability that a point belongs to a certain group (much like the Bayes classifier).

In the context of a Gaussian mixture, we have the following components:

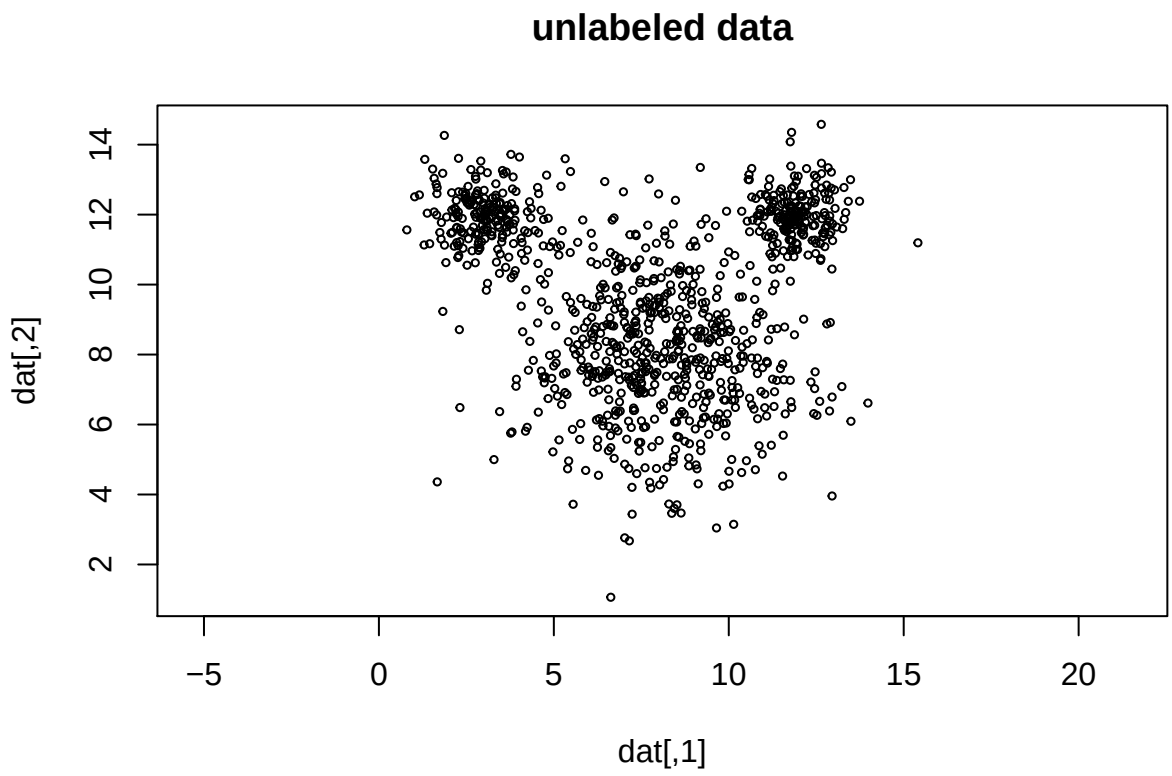
1) X is our observed data
2) Z is the missing data: the cluster to which the observations X belong.
3) X come from a normal distributions defined by the unknown parameters Θ (the mean μ and variance Σ).
4) Z is generated by a categorical distribution based on the unknown class mixing parameters α. ($\sum \alpha_i = 1$)

Thus,

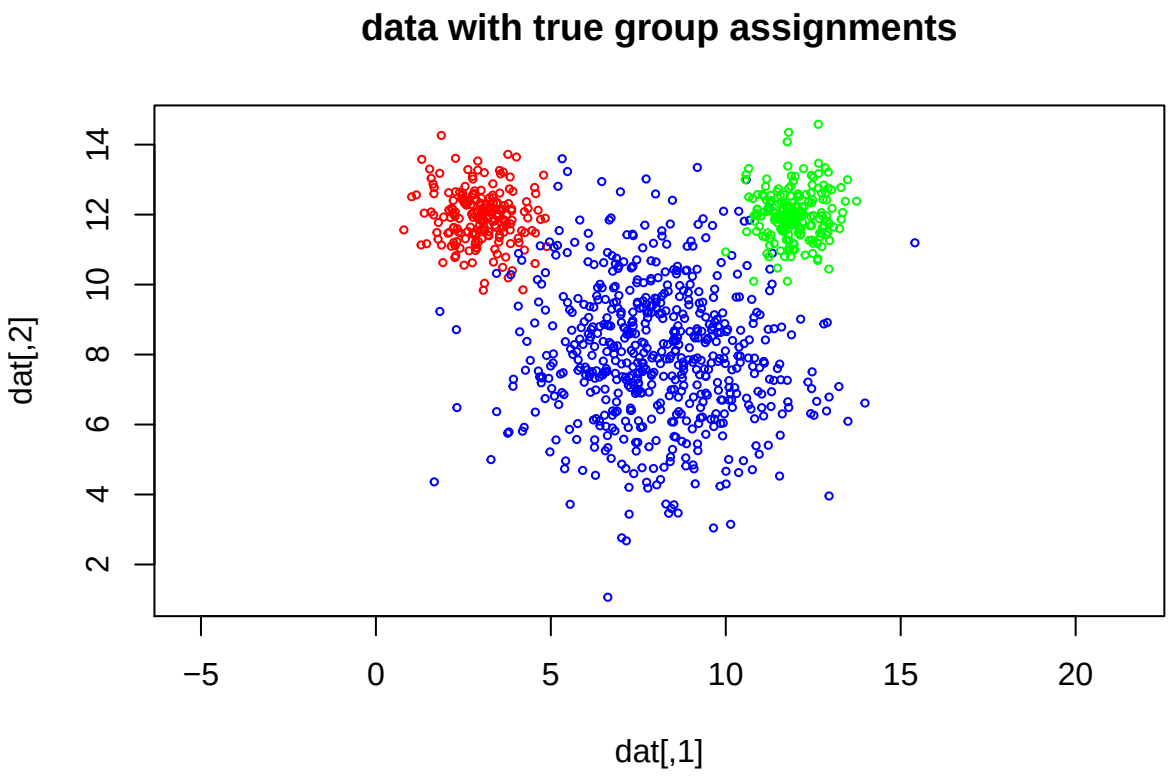
$$P(x|\Theta) = \sum_{k=1}^{K} \alpha_k P(X|Z_k, \theta_k)$$

We will use the following code to generate our data. It generates 1000 points.

```
# Don't change this code. It will be used to generate the data.
set.seed(2020)
library(mvtnorm)
cv <- matrix(c(1,0,0,1), ncol=2)
j <- rmvnorm(200, mean = c(3,12), sigma = .5*cv)
k <- rmvnorm(600, mean = c(8,8), sigma = 4*cv)
l <- rmvnorm(200, mean = c(12,12), sigma = .5*cv)
dat <- rbind(j,k,l)
em_true_groups <- as.factor(c(rep("j",200),rep("k",600),rep("l",200) ))
plot(dat, main = "unlabeled data", asp = 1, cex = 0.5)
```




```
col = c("red", "blue", "green")
plot(dat, col = col[em_true_groups], main = "data with true group assignments", asp = 1, cex = 0.5)
```



The EM algorithm for Gaussian Mixtures will behave as follows:

1) Begin with some random or arbitrary starting values of Θ and α.

2) E-Step. In the E-step, we will use Bayes' theorem to calculate the posterior probability that an observation i belongs to component k.

$$w_{ik} = p(z_{ik} = 1|x_i, \theta_k) = \frac{p(x_i|z_k, \theta_k)p(z_k = 1)}{\sum_{j=1}^{K} p(x_i|z_j, \theta_j)p(z_j = 1)}$$

We will define α_k as that the probability that an observation belongs to component k, that is $p(z_k = 1) = \alpha_k$.

We also know that the probability of our x observations follow a normal distribution. That is to say $p(x_i|z_k, \theta_k) = N(x_i|\mu_j, \Sigma_j)$. Thus, the above equation simplifies to:

$$w_{ik} = \frac{N(x_i|\mu_k, \Sigma_k)\alpha_k}{\sum_{j=1}^{K} N(x_i|\mu_j, \Sigma_j)\alpha_j}$$

This is the expectation step. It essentially calculates the 'weight' or the 'responsibility' that component k has for observation i. This reflects the expectations about the missing values of z based on the current estimates of the distribution parameters Θ.

3) M-step. Based on the estimates of the 'weights' found in the E-step, we will now perform Maximum Likelihood estimation for the model parameters.

This turns out to be fairly straightforward, as the MLE estimates for a normal distribution are fairly easy to obtain analytically.

For each component, we will find the mean, variance, and mixing proportion based on the data points that are "assigned" to the component. The data points are not actually "assigned" to the components like they are in k-means, but rather the components are given a "weight" or "responsibility" for each observation.

Thus, our MLE estimates are:

$$\alpha_k^{new} = \frac{N_k}{N}$$

$$\mu_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} w_{ik} x_i$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} w_{ik} (x_i - \mu_k^{new})(x_i - \mu_k^{new})^T$$

4. Iterate between steps 2 and 3 until convergence is reached.

Coding the EM algorithm for Gaussian Mixtures

Coding the algorithm is a matter of turning the above steps into code.

The package `mvtnorm` handles multivariate normal distributions. The function `dmvnorm()` can be used to find the probability of the data $N(x_i|\mu_k, \Sigma_k)$. It can even be applied in vector form, so you can avoid loops when trying to find the probabilities.

You are dealing with a 1000 x 2 matrix of data points.

A few key things to remember / help you troubleshoot your code:

1) Your matrix of 'weights' will be 1000 x 3. (one row for each observation, one column for each cluster)
2) N_k is a vector of three elements. It is effectively the column sums of the weight matrix w.
3) α is a vector of three elements. The elements will add to 1.
4) μ is a 3 x 2 matrix. One row for each cluster, one column for each x variable.
5) Each covariance matrix sigma is a 2x2 matrix. There are three clusters, so there are three covariance matrices.

Tip for the covariance matrices Σ As I was coding, I struggled a bit with creating the covariance matrices. I ended up having to implement the formula almost exactly as it was written. I wrote a loop to calculate each covariance matrix. My loop went through the data matrix, row by row. The operation $(x_i - \mu_k^{new})(x_i - \mu_k^{new})^T$ taxes a 2x1 matrix and matrix-multiplies it by a 1x2 matrix, resulting in a 2x2 matrix. You need to do this for every row. Multiply the resulting 2x2 matrices by w_{ik}, and then add all of them together to form one 2x2 matrix. Then divide those values by N_k. That should give you Σ_k for one of the clusters.

Other tips I also suggest running through your code one iteration at a time until you are pretty sure that it works.

Another suggestion:

IMO, implementing the covariances is the hardest part of the code. Before trying to update the covariances, you can leave the covariance matrices as the identity matrix, or plug in the actual known covariance matrices for `sig1` `sig2` and `sig3`. This way you can test out the rest of the code to see if the values of the means are updating as you would expect.

Output Requriements

1) Run your EM algorithm until convergence is reached. Convergence can be deemed achieved when the mu and/or sigma matrices no longer changes.

2) Print out the resulting estimates of N_k, the μ and the Σ values.

3) Run the k-means clustering algorithm (not kernelized k-means) on the same data to estimate the clusters. (Your previous k-means code could work here, but you can also just use `kmeans()` which is probably faster.)

4) Produce three plots:

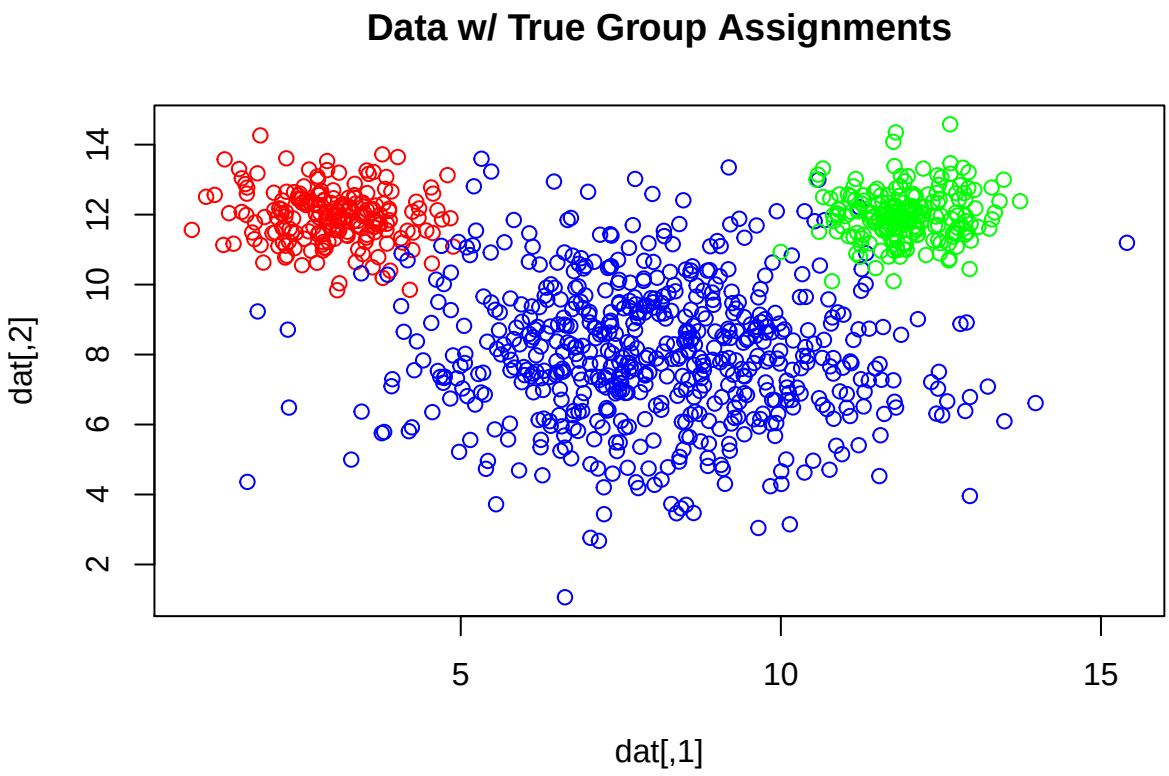
- Plot 1: Plot the original data, where the data is colored by the true groupings.
- Plot 2: Using the weight matrix, assign the data points to cluster that has the highest weight. Plot the data, colored by the estimated group membership.
- Plot 3: Using the results from the k-means clustering algorithm, plot the data colored by the k-means group membership.

```
# use these initial arbitrary values
N <- dim(dat)[1]  # number of data points
alpha <- c(0.2,0.3,0.5)  # arbitrary starting mixing parameters
mu <- matrix(  # arbitrary means
    c(5,8,
      7,8,
      9,8),
    nrow = 3, byrow=TRUE
)
sig1 <- matrix(c(1,0,0,1), nrow=2)  # three arbitrary covariance matrices
sig2 <- matrix(c(1,0,0,1), nrow=2)
sig3 <- matrix(c(1,0,0,1), nrow=2)

# Compute the while() loop
converged <- FALSE
while (!converged) {
  # E-Step:
  like_A <- dmvnorm(dat, mean=mu[1,], sigma=sig1)
  like_B <- dmvnorm(dat, mean=mu[2,], sigma=sig2)
  like_C <- dmvnorm(dat, mean=mu[3,], sigma=sig3)
  likelihood <- cbind(like_A, like_B, like_C)
  # utilize sweep() function taught by TA during discussion
  numerator <- sweep(likelihood, 2, alpha, `*`)
  marginal <- rowSums(numerator)
  w <- numerator/marginal

  # M-Step:
  Nk <- colSums(w)
  alpha_new <- Nk/N
  mu_new <- (1/Nk) * t(w) %*% dat
  mean_diff_A <- sweep(dat, 2, mu_new[1,])
  mean_diff_B <- sweep(dat, 2, mu_new[2,])
  mean_diff_C <- sweep(dat, 2, mu_new[3,])
  sig1_new <- (1/Nk[1]) * (t(mean_diff_A) %*% (w[,1] * mean_diff_A))
  sig2_new <- (1/Nk[1]) * (t(mean_diff_B) %*% (w[,2] * mean_diff_B))
  sig3_new <- (1/Nk[1]) * (t(mean_diff_C) %*% (w[,3] * mean_diff_C))

  # Convergence Criteria
```

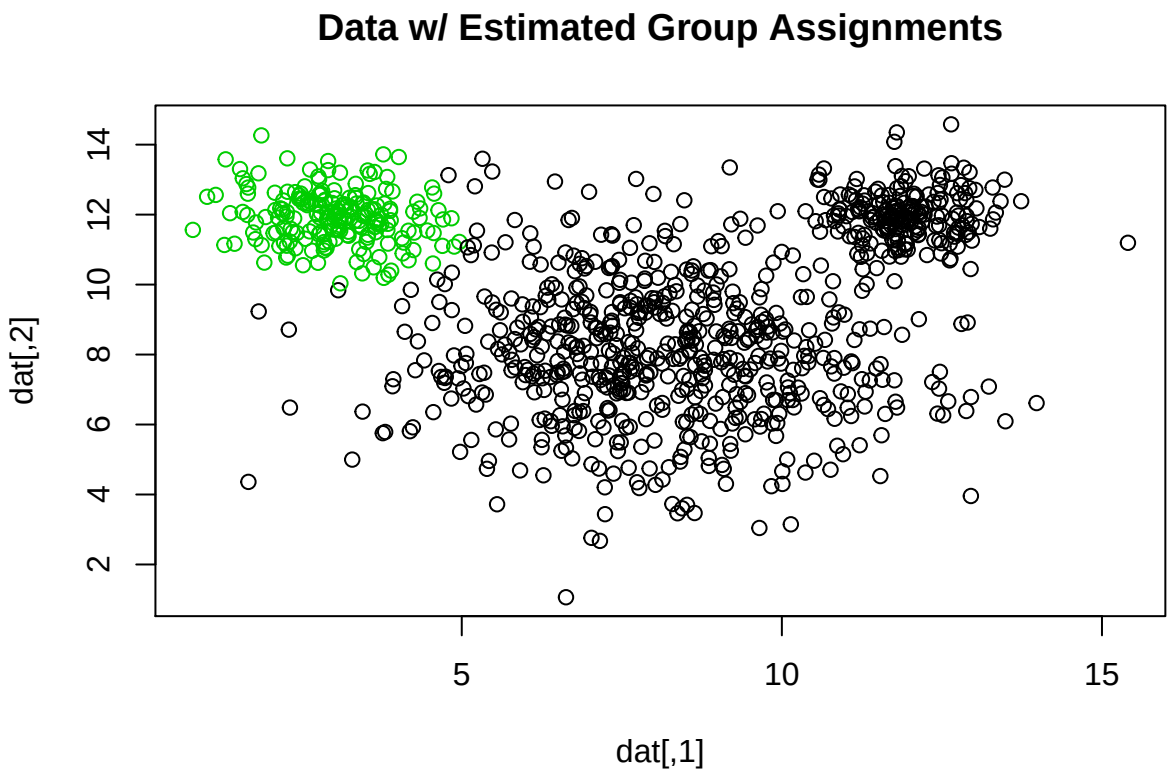
```
  if(sum(abs(mu_new - mu)) < 1e-13) {
    converged = TRUE
    print(Nk)
    print(mu_new)
    print(sig1_new)
    print(sig2_new)
    print(sig3_new)
  }
  else {
    alpha <- alpha_new
    mu <- mu_new
    sig1 <- sig1_new
    sig2 <- sig2_new
    sig3 <- sig3_new
    }
}
```

```
##   like_A   like_B   like_C
## 187.5751 406.2125 406.2125
##            [,1]      [,2]
## like_A 2.996486 11.950768
## like_B 8.912242  9.049215
## like_C 8.912242  9.049215
##             [,1]        [,2]
## [1,]  0.65161888 -0.08336452
## [2,] -0.08336452  0.49758865
##           [,1]      [,2]
## [1,] 14.171358  5.253569
## [2,]  5.253569 13.171875
##           [,1]      [,2]
## [1,] 14.171358  5.253569
## [2,]  5.253569 13.171875
```

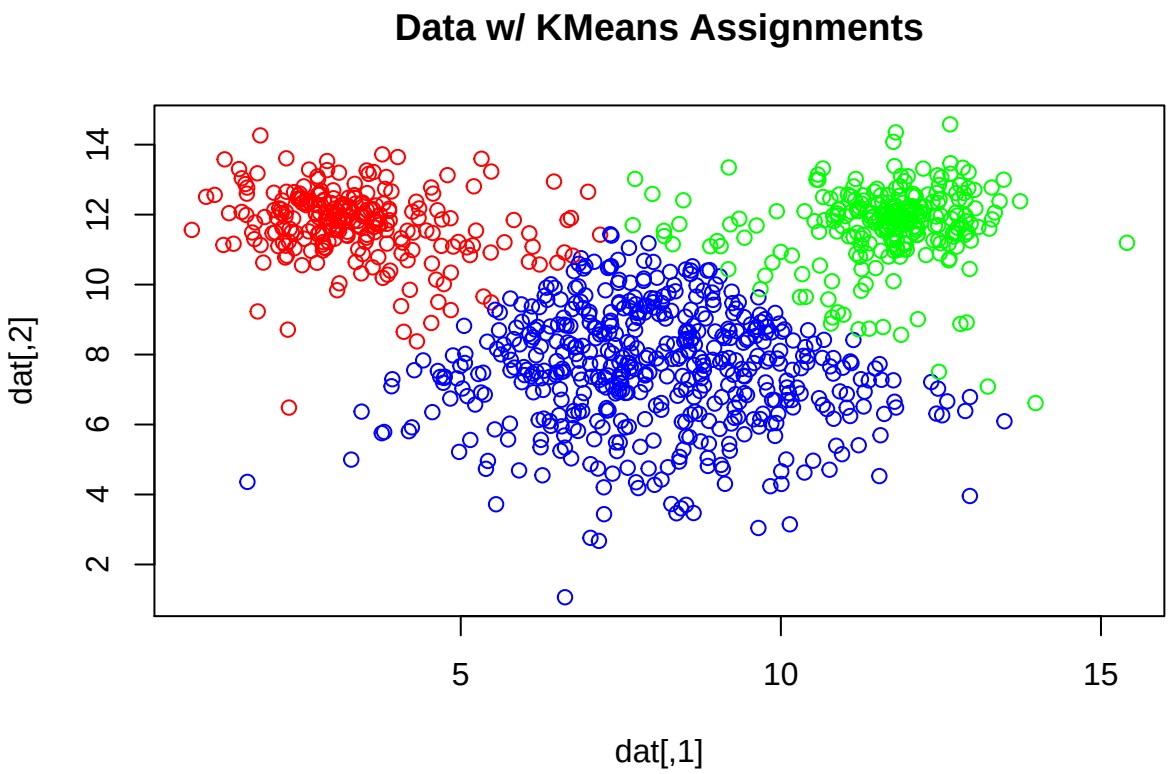
```
# Plot
plot(dat, col=col[em_true_groups], main="Data w/ True Group Assignments")
```




```
assignments <- apply(w, 1, which.min)
plot(dat, col=assignments, main="Data w/ Estimated Group Assignments")
```




```
plot(dat, col=col[kmeans(dat, 3)$cluster], main="Data w/ KMeans Assignments")
```


Data w/ KMeans Assignments
dat[,2]
2
4
6
8
10
12
14
5
10
15
dat[,1]