Final Exam

Andus Kong

March 5, 2016

Part 1: Coordinate Descent Algorithm

Coordinate descent is an optimization algorithm. It can be used to find a local minimum of a function. To perform coordinate descent, you perform a line search along one coordinate direction to find the value that minimizes the function in that direction while the other values are held constant. Once the value for that direction is updated, you perform the same operation for the other coordinate directions. This repeats until it has been updated for all coordinate directions, at which point the cycle repeats.

Thus for a function of two variables $f(x, y)$, a simple version of the algorithm can be described as follows:

1) Start with some initial values of x and y. This is time 0, so we have $x^{(0)}$ and $y^{(0)}$.
2) Iterate:
    1) Update $x^{(t+1)}$ to be the value of x that minimizes $f(x, y = y^{(t)})$
    2) Update $y^{(t+1)}$ to be the value of y that minimizes $f(x = x^{(t+1)}, y)$
3) Stop when some convergence criterion has been met.

The "tricky" part of the algorithm is finding the value that minimizes the function along one of the directions.

Golden Section Search Method (with Video)

This unidimensional minimization be done in one of many ways, but for our purposes, we will use the golden section search method.

The premise of how the golden section search works is summarized very nicely in this video from CUBoulder-Computing: https://vimeo.com/86277921

I will provide the code for the golden section search method here. This is a modified version of Eric Cai's code (https://chemicalstatistician.wordpress.com). It has been modified so that the locations of x1 and x2 match the CUBoulderComputing video

```
##### A modifcation of code provided by Eric Cai
golden = function(f, lower, upper, tolerance = 1e-5)
{
   golden.ratio = 2/(sqrt(5) + 1)

   ## Use the golden ratio to find the initial test points
   x1 <- lower + golden.ratio * (upper - lower)
   x2 <- upper - golden.ratio * (upper - lower)

   ## the arrangement of points is:
   ## lower ----- x2 --- x1 ----- upper

   ### Evaluate the function at the test points
   f1 <- f(x1)
   f2 <- f(x2)
```

```r
  while (abs(upper - lower) > tolerance) {
      if (f2 > f1) {
      # the minimum is to the right of x2
      lower <- x2  # x2 becomes the new lower bound
      x2 <- x1     # x1 becomes the new x2
      f2 <- f1     # f(x1) now becomes f(x2)
      x1 <- lower + golden.ratio * (upper - lower)
      f1 <- f(x1)  # calculate new x1 and f(x1)
      } else {
      # then the minimum is to the left of x1
      upper <- x1  # x1 becomes the new upper bound
      x1 <- x2     # x2 becomes the new x1
      f1 <- f2
      x2 <- upper - golden.ratio * (upper - lower)
      f2 <- f(x2)  # calculate new x2 and f(x2)
      }
   }
   (lower + upper)/2 # the returned value is the midpoint of the bounds
}
```

We can thus use the golden search to find the minimizing value of a function. For example, the function $f(x) = (x - 3)^2$ has a minimum value at $x = 3$.

```r
f <- function(x){ (x - 3)^2 }
golden(f, 0, 10)
```

```
## [1] 3.000001
```

Back to Coordinate Descent

With our golden search function, we can now create our coordinate descent algorithm:

1) Start with some initial values of x and y. This is time 0, so we have $x^{(0)}$ and $y^{(0)}$.
2) Iterate:
    1) Update x:
        a. Find the function $f(x) = f(x, y = y^{(t)})$
        b. Use golden search to minimize $f(x)$
        c. Set $x^{(t+1)}$ be the result of the search.
    2) Update y
        a. Find the function $f(y) = f(x = x^{(t+1)}, y)$
        b. Use golden search to minimize $f(y)$
        c. Set $y^{(t+1)}$ be the result of the search.
3) Stop when some convergence criterion has been met.

Task 1

Write code to perform coordinate descent to minimize the following function:

$$g(x, y) = 5x^2 - 6xy + 5y^2$$

```
# do not modify this code
g <- function(x,y) {
    5 * x ^ 2 - 6 * x * y + 5 * y ^ 2
    }
x <- seq(-1.5,1.5, len=100)
y <- seq(-1.5,1.5, len=100)
z <- outer(x,y,g)
```

Requirements:

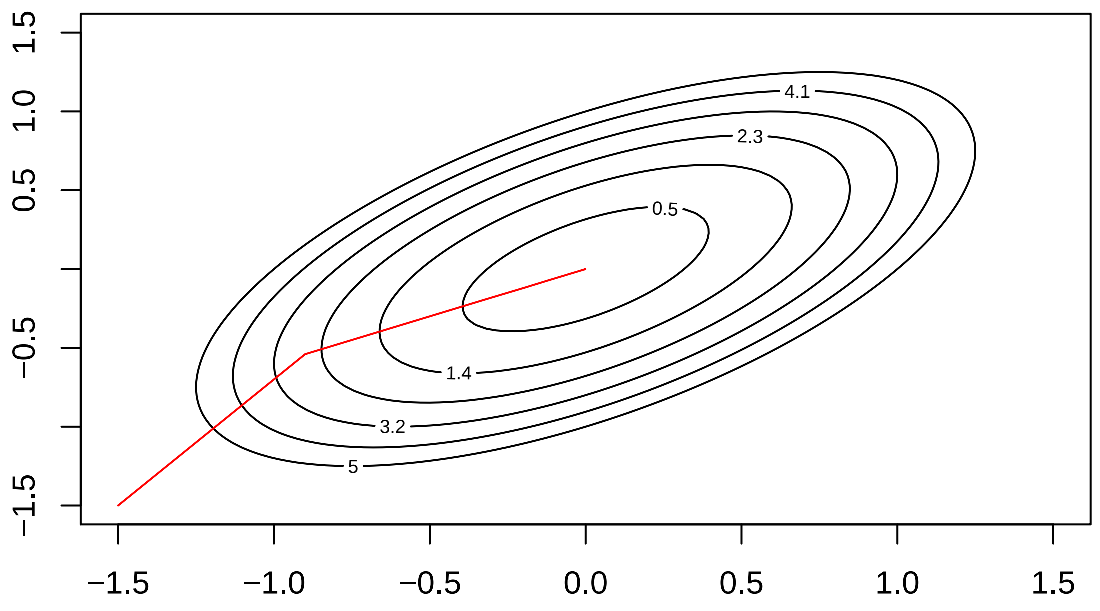
1) Your search space for the golden search can be limited to the range -1.5 to 1.5 for both the x and y directions.
2) For your starting point, use x = -1.5, and y = -1.5.
3) For the first step, hold y constant, and find the minimum in the x direction.
4) Plot the segments showing each 'step' of the algorithm onto the contour plot. See the example posted on CCLE with different starting coordinates.
5) After each full iteration, print out the current values of x and y. Hint: after your first full iteration, the next location should be (-0.9, -0.54).
6) Stop after 15 full iterations, or if the difference between one x and the next is less then `1e-5`. The true minimum is at (0,0). Your code should come close to that.

This shouldn't be too hard. I've already given you the golden search function. My complete solution is 12 lines. Of course yours might be longer or shorter, but that should give you an idea of how complicated or not complicated it needs to be.

```
contour(x,y,z, levels = seq(.5,5,by=.9)) # code to plot contour lines
x_i <- -1.5
y_i <- -1.5

# write your code here

### Build coordinate matrix and initialize starting points ###
coord <- matrix(nrow = 16, ncol = 2)
coord[1,1] <- x_i
coord[1,2] <- y_i

########################
# Turns g(x,y) unidimensional
#
# @param x: Variable to be held constant
# returns: unidimensional function
# NOTE: Does not matter if holding x or y constant
#       since the partial derivatives of g(x,y) are essentially equal
#########################
g1 <- function(x) function(y) g(x,y)

# Perform gradient descent using golden ratio as the minimizing function
for (i in 1:15){
  # Calculate new x and y
  x_n <- golden(g1(x_i),-1.5, 1.5)
  y_n <- golden(g1(x_n),-1.5, 1.5)
  coord[i + 1,1] <- round(x_n, 3)
  coord[i + 1,2] <- round(y_n, 3)

  # Check for convergence
  if(abs(x_n - x_i) < 1e-5) break
  x_i <- x_n
  print(c(x_n, y_n))
}
```

```
## [1] -0.9000005 -0.5399991
## [1] -0.5399991 -0.3239996
## [1] -0.3239996 -0.1943994
## [1] -0.1943994 -0.1166404
## [1] -0.11664038 -0.06998455
## [1] -0.06998455 -0.04199138
## [1] -0.04199138 -0.02519619
## [1] -0.02519619 -0.01511759
## [1] -0.015117592 -0.009071722
## [1] -0.009071722 -0.005444077
## [1] -0.005444077 -0.003266769
## [1] -0.003266769 -0.001961105
## [1] -0.001961105 -0.001174652
## [1] -0.0011746518 -0.0007038706
## [1] -0.0007038706 -0.0004213549
```

```
lines(coord, col = "red")
```



Part 2: Gradient Descent

We have already covered gradient descent in the context of minimizing the cost function for an artificial neural network. Here, you will perform gradient descent on a much simpler function.

Again, the gradient descent algorithm can be summarized as follows:

1) Find the gradient at the current estimate. Take a 'step' in the direction opposite of the gradient multiplied by the learning rate α.
2) Repeat until convergence

Task 2

Write code to perform gradient descent to minimize the following function:

$$g(x, y) = 5x^2 - 6xy + 5y^2$$

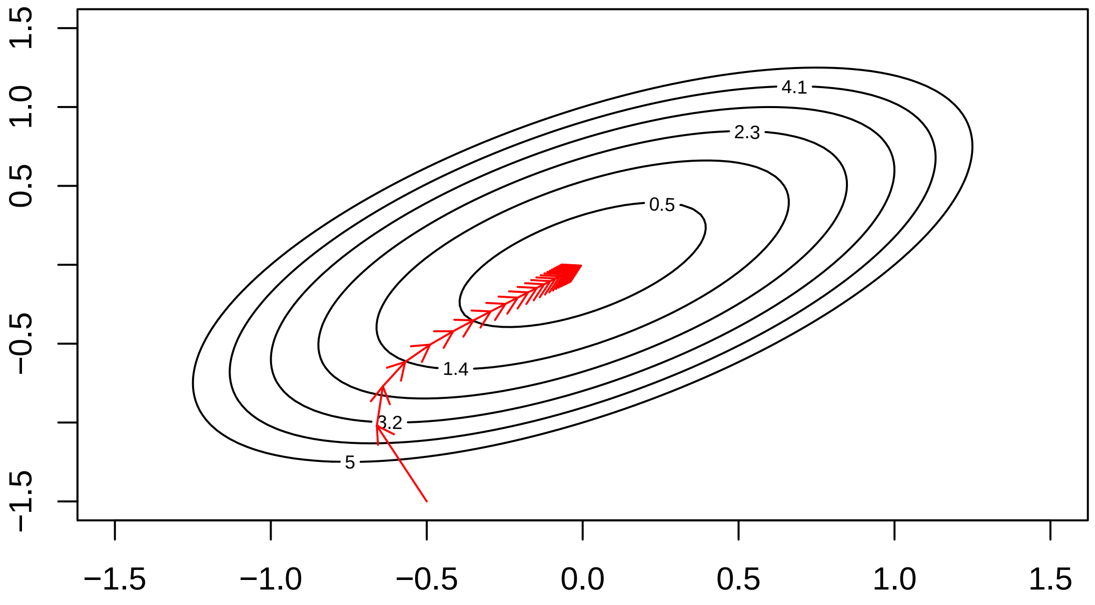
(This is the same function from the previous problem.)

Requirements:

1) For your starting point, use x = -0.5, and y = -1.5.
2) For each iteration, plot arrows indicating the location of the next estimate. I recommend using the command `arrows()` with `length` set to 0.1.
3) Have convergence criteria: stop when $|x^{(t)} - x^{(t+1)}| < 0.001$ and when $|y^{(t)} - y^{(t+1)}| < 0.001$ (`1e-3`)
4) Use a learning rate of $\alpha = 0.04$.

```r
# do not modify this code
g <- function(x,y) {
    5 * x ^ 2 - 6 * x * y + 5 * y ^ 2
    }
x <- seq(-1.5,1.5, len=100)
y <- seq(-1.5,1.5, len=100)
z <- outer(x,y,g)
contour(x,y,z, levels = seq(.5,5,by=.9)) # code to plot contour lines
x_i <- -0.5
y_i <- -1.5
alpha = 0.04

## write your code here

#####################
# Calculates partial derivative with respect to x
#
# @param x: x coordinate to be evaluated
# @param y: y coordinate to be evaluated
# returns the gradient with respect to x
#####################
x_grad <- function(x, y, alpha){
  x - alpha * (10 * x - 6 * y)
}

## write your code here
```

```r
#####################
# Calculates partial derivative with respect to y
#
# @param x: x coordinate to be evaluated
# @param y: y coordinate to be evaluated
# returns the gradient with respect to y
#####################
y_grad <- function(x, y, alpha) {
  y - alpha * (10 * y - 6 * x)
}

# Iterate through gradient descent until convergence
repeat{

  # Calculate new x and y
  x_n <- x_grad(x_i, y_i, alpha)
  y_n <- y_grad(x_i, y_i, alpha)

  # Check for convergence
  if(abs(x_i - x_n) < 1e-3 && abs(y_i - y_n) < 1e-3) break
  arrows(x_i, y_i, x_n, y_n, col = "red", length = 0.1)
  x_i <- x_n
  y_i <- y_n
}
```



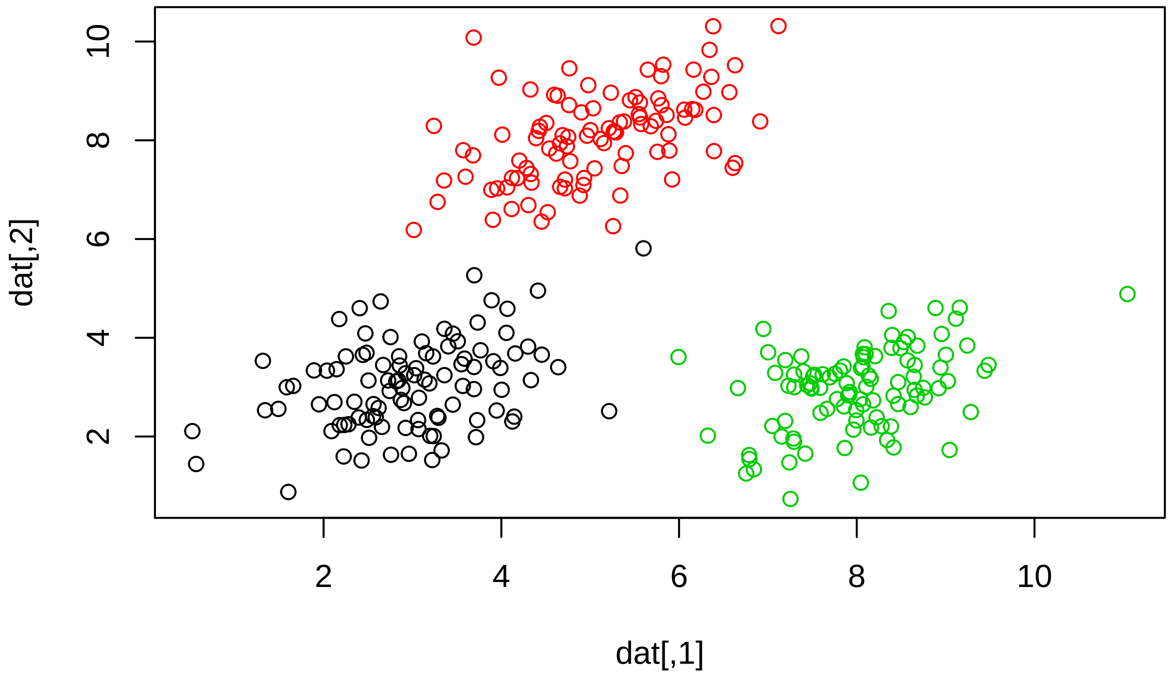
Part 3: K-means Clustering

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

1) Begin with a set of k random or arbitrary starting points. These are the 'means'
2) Clusters are created by associating values in the data set to the nearest (euclidean distance) mean.
3) Once all values have been assigned to a cluster, recalculate the means as the centroid of the values in each cluster.
4) Repeat steps 2 and 3 until convergence.

https://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm

```r
# Don't change this code. It will be used to generate the data.
set.seed(2016)
library(mvtnorm)
cv <- matrix(c(.8,.4,.4,.8), 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)
```



Task 3

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

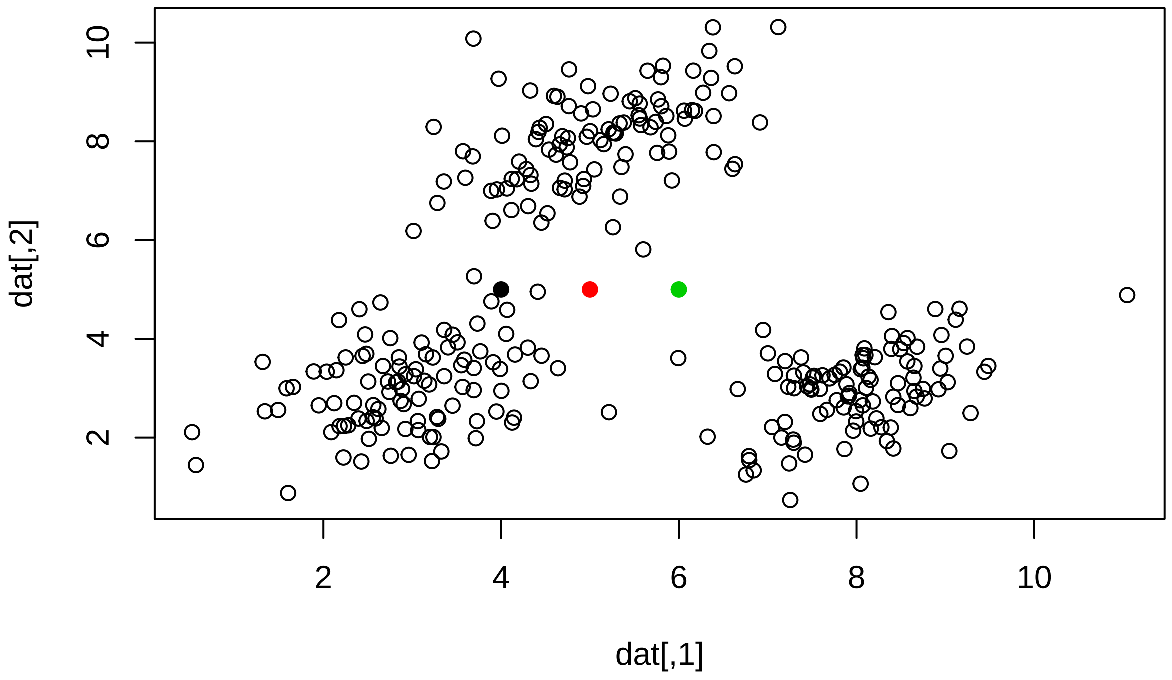
The true group labels are provided in the vector `true_groups`. Of course, you can't use that until the very end where you will perform some verification.

Requirements:

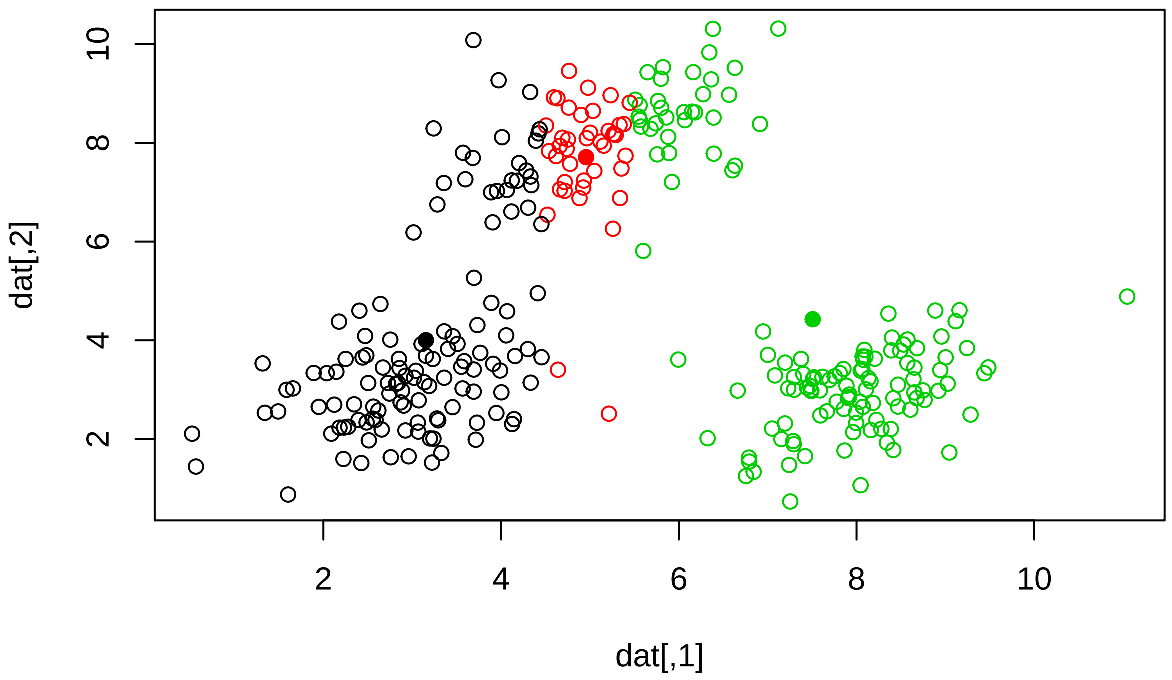
1) So everyone will get consistent results, I have selected the initial starting locations of the 3 'means' for you: (4,5), (5,5), (6,5).
2) With each iteration, plot the data, colored by their current groupings, and the updated means.
3) Convergence is reached when group assignments no longer change. Your k-means clustering algorithm should reach convergence fairly quickly, about 6 iterations or less.
4) Print out a 'confusion' matrix showing how well the k-means clustering algorithm classified the data.

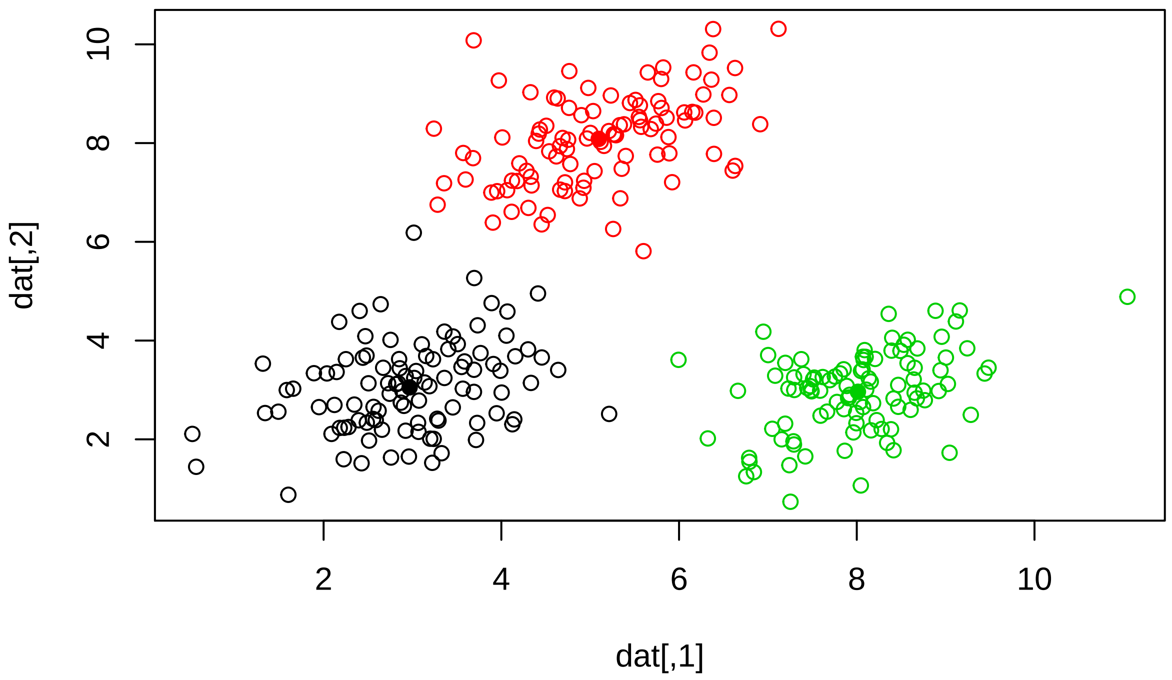
IMO, the code for k-means requires a bit more creativity. You'll probably want to write a function that will calculate distances from the three means. I also highly recommend avoiding the use of loops within each iteration. Instead try to use apply functions. My final code takes 15 lines. You don't need to write as efficiently as I do. You can write a hundred lines, but as long as it gets the job done correctly, it'll be accepted.

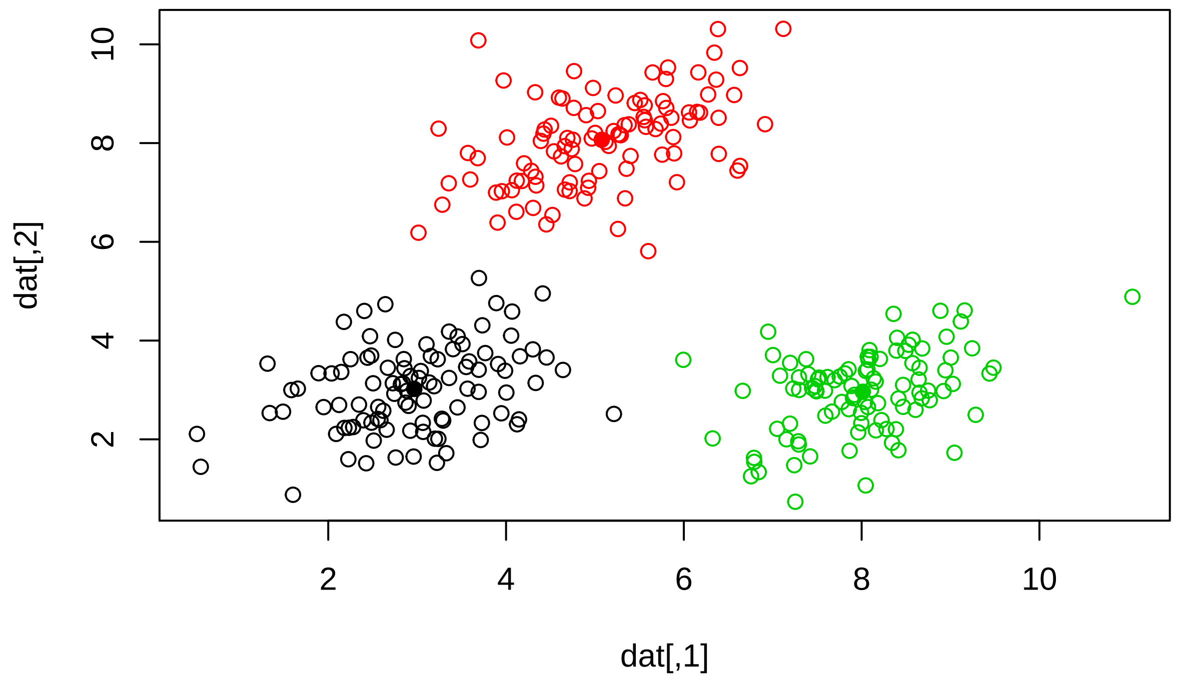
```r
# do not modify this code
means <- rbind(c(4,5), c(5,5), c(6,5)) # matrix of starting means
groupings = as.factor(rep(1,300))  #initial groupings that you will need to update
plot(dat, col = groupings)  # initial plot
points(means, col = as.factor(1:3), pch = 19) # add dots for the means
```



```r
# write your code here

################
# Calculates Euclidean distance
#
# @param x_1: point 1 (x,y)
# @param x_2: point 2
# returns: Euclidean distance between two points
################
dis <- function(x_1,x_2){
  x_i <- x_1[1]
  y_i <- x_1[2]
  x_f <- x_2[1]
  y_f <- x_2[2]
  sqrt((x_i - x_f)^2 + (y_i - y_f)^2)
}

# Iterate through K means clustering
repeat{
  distance <- list()

  # Calculate all distances from each point to the 3 means (3 distances per point)
  for (i in 1: 300)
  distance[[i]] <- apply(means, 1, FUN = function(x) (dis(x,dat[i,])))

  minimum_dist <- list()

  # Find smallest distance for each point

  minimum_dist<- lapply(distance, FUN = which.min)
  groupings_n <- as.factor(unlist(minimum_dist))

  # Check for convergence
  if (sum(as.integer(groupings_n) == as.integer(groupings)) == 300) break
  groupings <- groupings_n

  # Calculate new means
  means <- rbind(c(mean(dat[groupings == 1,1]), mean(dat[groupings == 1,2])),
                 c(mean(dat[groupings == 2,1]), mean(dat[groupings == 2,2])),
                 c(mean(dat[groupings == 3,1]), mean(dat[groupings == 3,2])))
  plot(dat, col = groupings)
  points(means, col = as.factor(1:3), pch = 19)
}
```


dat[,2]
dat[,1]
2
4
6
8
10
2
4
6
8
10


dat[,2]
dat[,1]
2
4
6
8
10
2
4
6
8
10



```r
# Confusion Matrix
table(groupings_n, true_groups) # misclassified one point
```

```
##            true_groups
## groupings_n   j   k   l
##           1  99   0   0
##           2   1 100   0
##           3   0   0 100
```

Part 4: EM Algorithm

The Expectation-Maximization algorithm is an iterative algorithm for finding maximum likelihood estimates of parameters when some of the data is missing.

The general form consists of alternating between an expectation step and a maximization step. 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 given the values of X under the current estimates of θ. In the maximization step, the values of θ are found that will maximize this expected log-likelihood.

The great thing is 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

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. A key difference between the k-means algorithm and the EM algorithm is that the EM algorithm is probabalistic. While 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.

A bonus is that, IMO, the EM algorithm in the context of gaussian mixtures is a bit easier to understand than the general form of the EM algorithm.

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

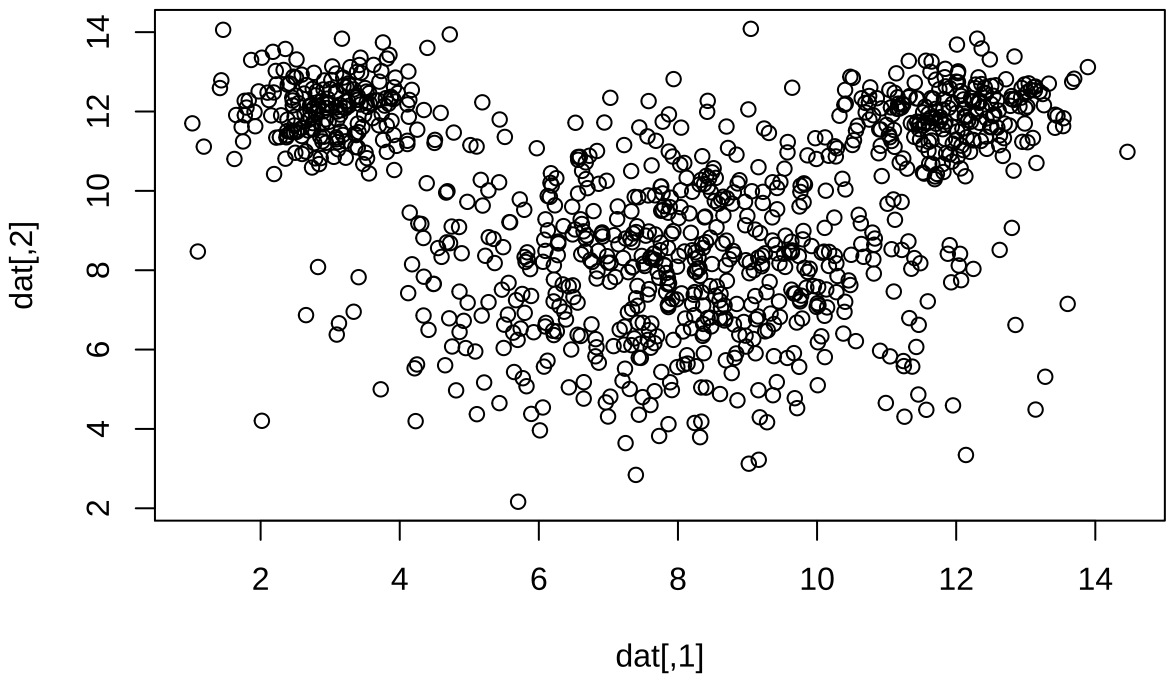
1) X is our observed data
2) Z is the missing data: the class or cluster to which the observations X belong.
3) X come from a normal distributions defined by the unknown parameters Θ (the mean μ and variance Σ).
4) Z is generated by a categorical distribution based on the unknown class mixing parameters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

```r
# Don't change this code. It will be used to generate the data.
set.seed(2016)
library(mvtnorm)
cv <- matrix(c(1,.1,.1,1), 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)
true_groups <- as.factor(c(rep("j",200),rep("k",600),rep("l",200) ))
plot(dat)
```



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.

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

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. 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 on the same data to estimate the clusters. (Your previous k-means code should work here as well. You'll probably just need to change a few lines: the number of data points. You won't be producing plots for each iteration either.)
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.

```r
# 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)

## write your code here
#########################
# EM ALGORITHM
#########################
# initialize groups variable
grouping = as.factor(rep(1,1000))

# Iterate through EM algorithm
repeat{
  mvd_j <- vector()
  mvd_k <- vector()
  mvd_l <- vector()
```

```r
# Calculate probablity of data for each component
for(i in 1:N){
  mvd_j[i] <- dmvnorm(dat[i,], mean = mu[1,], sigma = sig1) * alpha[1]
  mvd_k[i] <- dmvnorm(dat[i,], mean = mu[2,], sigma = sig2) * alpha[2]
  mvd_l[i] <- dmvnorm(dat[i,], mean = mu[3,], sigma = sig3) * alpha[3]
}
mvd <- data.frame(mvd_j, mvd_k, mvd_l)

# Calculate sum of probabilities of each component
total_mvd <- apply(mvd, 1, sum)

weight1 <- vector()
weight2 <- vector()
weight3 <- vector()

# calculate weights for each component
for(i in 1:N){
  weight1[i] <- mvd_j[i] / total_mvd[i]
  weight2[i] <- mvd_k[i] / total_mvd[i]
  weight3[i] <- mvd_l[i] / total_mvd[i]
}
weight <- data.frame(weight1, weight2, weight3)

# Determine new groups for each point
grouping_n <- apply(weight, 1, FUN = function(x) which.max(x))

# Check for convergence
if(sum(as.integer(grouping_n) == as.integer(grouping)) == 1000) break
grouping <- grouping_n

# calculate number of points in each component using probablistic weights
Nk <- apply(weight, 2, sum)

# Calculate weight over Nk for each component
constant <- matrix(ncol = 3, nrow = N)
constant[,1] <- weight[,1] / Nk[1]
constant[,2] <- weight[,2] / Nk[2]
constant[,3] <- weight[,3] / Nk[3]

# Calculate new alpha level
alpha <- Nk / 1000

# Calculate new means
mu <- t(constant) %*% (dat)

# Calculate standard deviation and variance covariance matrix for component 1
sd_j <- t(apply(dat,1, FUN = function(x) x - mu[1,]))
var_j <- t(apply(sd_j, 1, FUN = function(x) x %*% t(x)))

# Calculate standard deviation and variance covariance matrix for component 2
sd_k <- t(apply(dat,1, FUN = function(x) x - mu[2,]))
var_k <- t(apply(sd_k, 1, FUN = function(x) x %*% t(x)))
```

```
  # Calculate standard deviation and variance covariance matrix for component 3
  sd_l <- t(apply(dat,1, FUN = function(x) x - mu[3,]))
  var_l <- t(apply(sd_l, 1, FUN = function(x) x %*% t(x)))

  # Multiply variance covariance matrix by constants to find final sigma for all components
  sig1 <- matrix(t(constant[,1]) %*% var_j, ncol = 2)
  sig2 <- matrix(t(constant[,2]) %*% var_k, ncol = 2)
  sig3 <- matrix(t(constant[,3]) %*% var_l, ncol = 2)
}

# Final Nk
Nk
```

```
##  weight1  weight2  weight3
## 200.2496 585.5457 214.2047
```

```
# Final mu
mu
```

```
##           [,1]      [,2]
## [1,]  3.009942 12.052176
## [2,]  7.965885  8.013878
## [3,] 11.919635 11.913995
```

```
# Final sigma of component 1
sig1
```

```
##           [,1]      [,2]
## [1,] 0.5065032 0.0135643
## [2,] 0.0135643 0.5369009
```

```
# Final sigma of component 2
sig2
```

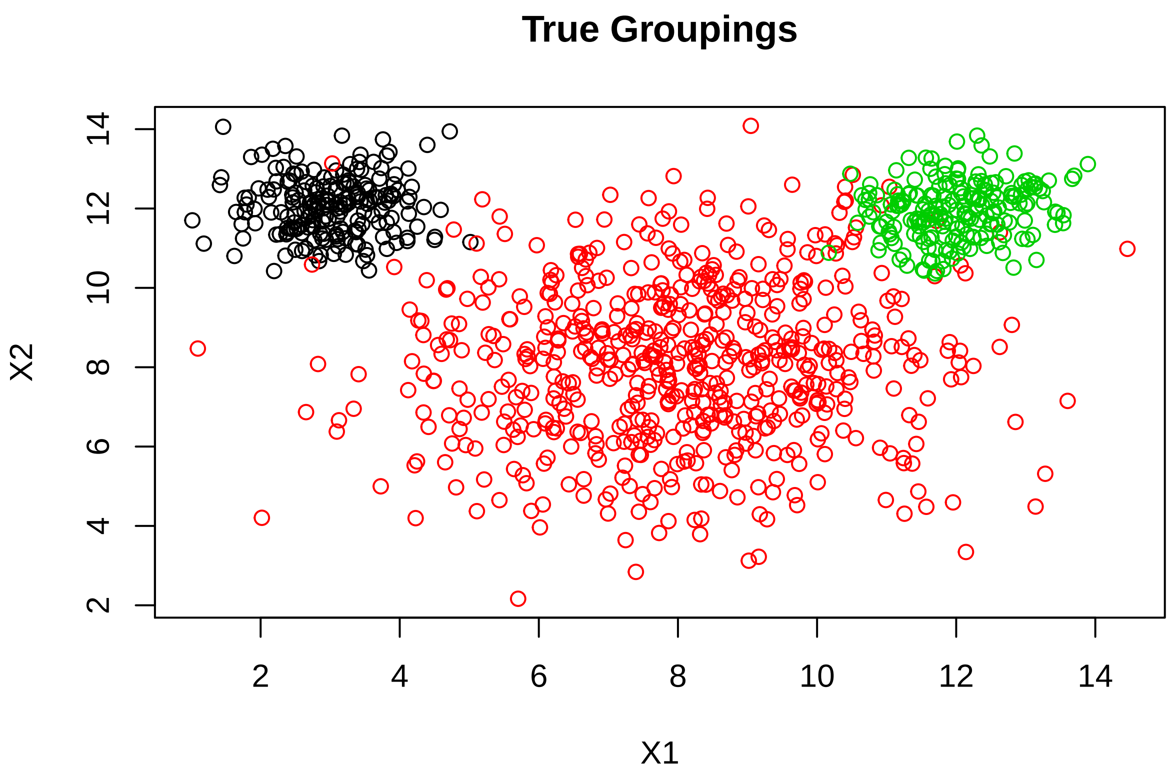
```
##            [,1]       [,2]
## [1,] 4.00187210 0.05948532
## [2,] 0.05948532 3.79799104
```

```
# Final sigma of component 3
sig3
```

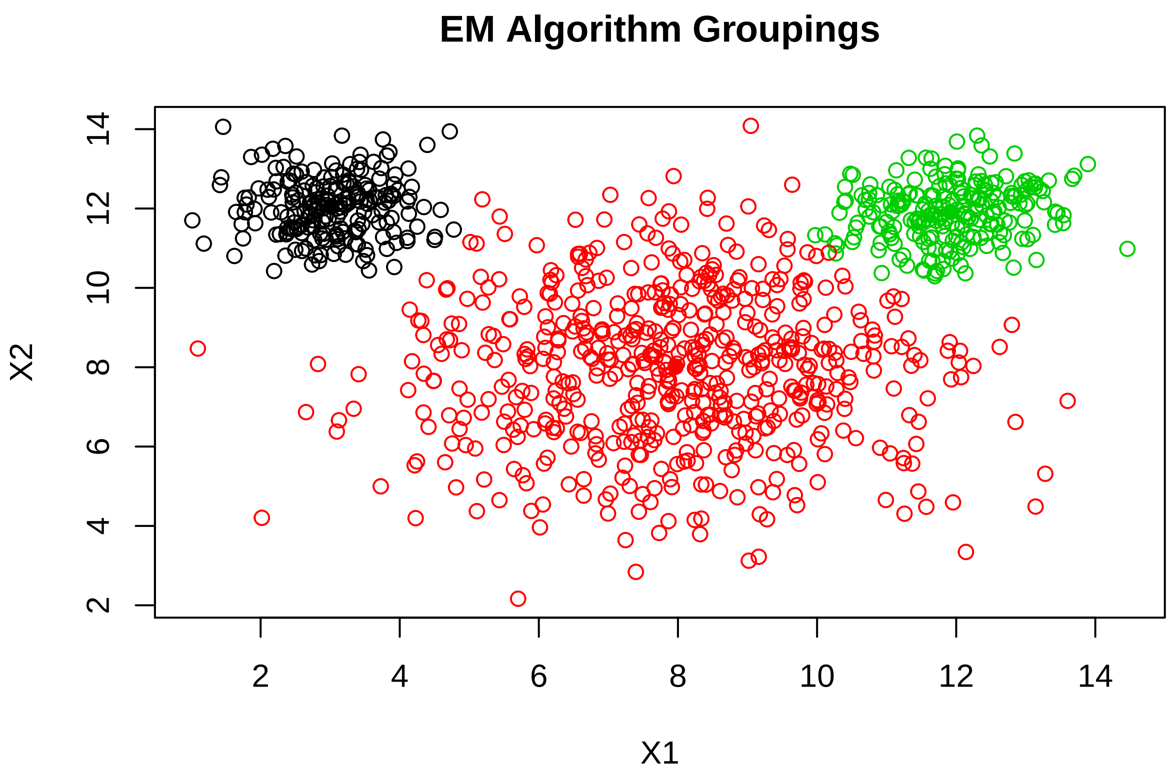
```
##           [,1]      [,2]
## [1,] 0.7036510 0.1140455
## [2,] 0.1140455 0.5160063
```

```
###########################
# K MEANS ALGORITHM
###########################

# Initialize means and groups
```

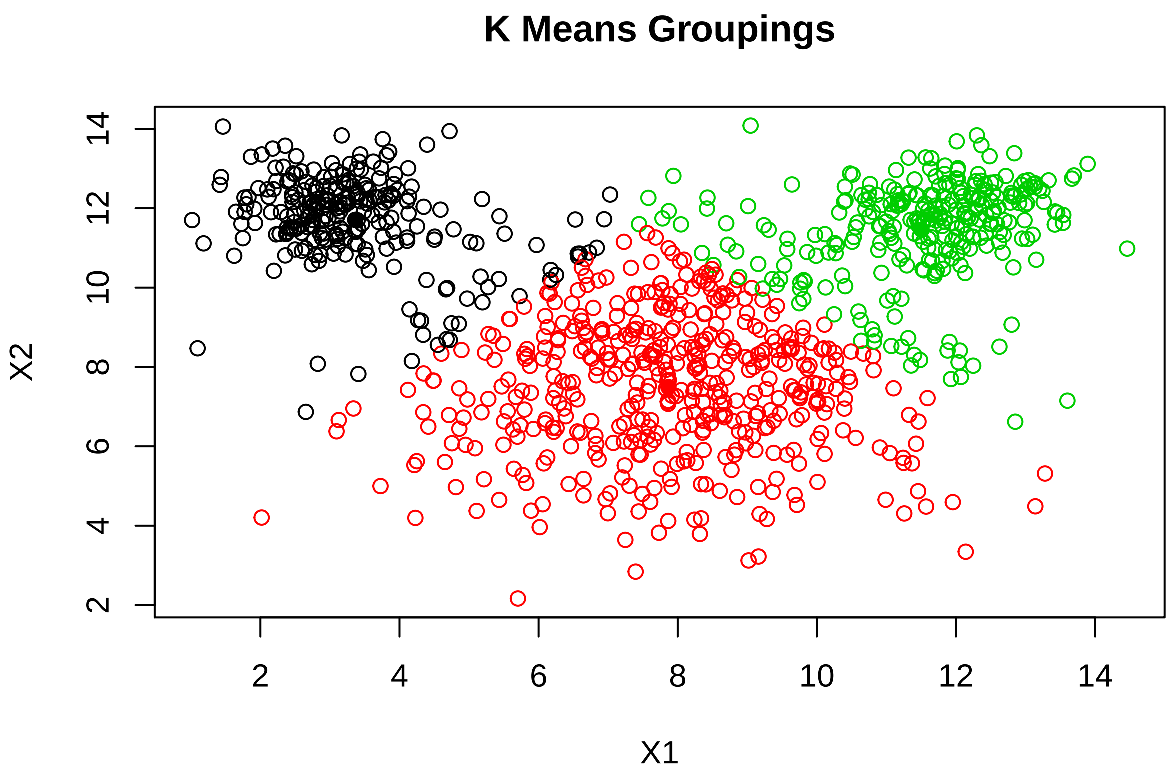
```r
means <- rbind(c(5,8), c(7,8), c(9,8))
groupings = as.factor(rep(1,1000))

# Iterate through K Means algorithm
repeat{
  distance <- list()

  # Calculate all distances from each point to the 3 means (3 distances per point)
  for (i in 1: 1000)
  distance[[i]] <- apply(means, 1, FUN = function(x) (dis(x,dat[i,])))

  minimum_dist <- list()

  # Find smallest distance for each point
  minimum_dist<- lapply(distance, FUN = which.min)
  groupings_n <- as.factor(unlist(minimum_dist))

  # Check for convergence
  if (sum(as.integer(groupings_n) == as.integer(groupings)) == 1000) break
    groupings <- groupings_n

  # Calculate new means
    means <- rbind(c(mean(dat[groupings == 1,1]), mean(dat[groupings == 1,2])),
                c(mean(dat[groupings == 2,1]), mean(dat[groupings == 2,2])),
                c(mean(dat[groupings == 3,1]), mean(dat[groupings == 3,2])))
}

# True Groupings
plot(dat, col = true_groups, main = "True Groupings", xlab = "X1", ylab = "X2")
```



```r
# EM Algorithm Groupings
plot(dat, col = grouping, main = "EM Algorithm Groupings", xlab = "X1", ylab = "X2")
points(mu, col = as.factor(1:3), pch = 19)
```



```r
# K Means Algorithm Groupings
plot(dat, col = groupings, main = "K Means Groupings", xlab = "X1", ylab = "X2")
points(means, col = as.factor(1:3), pch = 19)
```


K Means Groupings
X1
X2
2
4
6
8
10
12
14
2
4
6
8
10
12
14