Homework 4 - Neural Networks and Clustering

Miles Chen

Part 1: Neural Networks

For this homework assignment, we will build and train a simple neural network, using the famous "iris" dataset. We will take the four variables `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width` to create a prediction for the species.

We will train the network using gradient descent, a commonly used tool in machine learning.

Data Preparation:

I have split the iris data into a training and testing dataset. I have also scaled the data so the numeric variables are all between 0 and 1.

```r
# split between training and testing data
set.seed(1)
n <- dim(iris)[1]
rows <- sample(1:n, 0.8 * n)
colmax <- apply(iris[,1:4], 2, max)

train <- iris[rows, ]
train <- as.matrix(train[ , 1:4])
train <- t( t(train) / colmax)

test <- iris[-rows, ]
test <- as.matrix(test[ , 1:4])
test <- t( t(test) / colmax)
```

Setting up our network

Our neural network will have four neurons in the input layer - one for each numeric variable in the dataset. Our output layer will have three outputs - one for each species. There will be a `Setosa`, `Versicolor`, and `Virginica` node. When the neural network is provided 4 input values, it will produce an output where one of the output nodes has a value of 1, and the other two nodes have a value of 0. This is a similar classification strategy we used for the classification of handwriting digits.

I have arbitrarily chosen to have 3 nodes in our hidden layer.

We will add bias values before applying the activation function at each of our nodes in the hidden and output layers.

Task 1:

How many parameters are present in our model? List how many are present in: weight matrix 1, bias values for the hidden layer, weight matrix 2, and bias values for output layer.

Your answer:

Notation

We will define each matrix of values as follows:

$W^{(1)}$ the weights applied to the input layer.

$B^{(1)}$ are the bias values added before activation in the hidden layer.

$W^{(2)}$ the weights applied to the values coming from the hidden layer.

$B^{(2)}$ are the bias values added before the activation function in the output layer.

Task 2:

To express the categories correctly, we need to turn the factor labels in species column into vectors of 0s and 1s. For example, an iris of species *setosa* should be expressed as `1 0 0`. Write some code that will do this. Hint: you can use `as.integer()` to turn a factor into numbers, and then use a bit of creativity to turn those values into vectors of 1s and 0s.

```
# your code goes here
```

Forward Propagation

We will use the sigmoid function as our activation function.

$$f(t) = \frac{1}{1 + e^{-t}}$$

Task 3:

Write the output ($\hat{y}$) of the neural network as a function or series of functions of the parameters ($W^{(1)}, B^{(1)}, W^{(2)}, B^{(2)}$).

In the language of neural networks, this step is called forward propagation. It's the idea of taking your input values and propagating the changes forward until you get your predictions.

You can visit https://github.com/stephencwelch/Neural-Networks-Demystified/blob/master/Part%202%20Forward%20Propagation.ipynb to see how the series of functions would be written if we did not use bias values in our calculations.

Your answer: Write your answer here. I recommend typesetting with Latex to express the mathematics. You can learn about writing mathematics in latex at: https://en.wikibooks.org/wiki/LaTeX/Mathematics

$$Z^{(2)} = XW^{(1)}$$

Task 4:

Express the forward propagation as R code using the training data. For now use random uniform values as temporary starting values for the weights and biases.

```r
# your code goes here
```

Back Propagation

The cost function that we will use to evaluate the performance of our neural network will be the squared error cost function:

$$J = 0.5 \sum (y - \hat{y})^2$$

Task 5 (the hard task):

Find the gradient of the cost function with respect to the parameters.

You will create four partial derivatives, one for each of $(W^{(1)}, B^{(1)}, W^{(2)}, B^{(2)})$.

This is known as back propagation. The value of the cost function ultimately depends on the data and our predictions. Our predictions are just a result of a series of operations which you have defined in task 2. Thus, when you calculate the derivative of the cost function, you will be applying the chain rule for derivatives as you take the derivative with respect to an early element.

Your answer:

$$\frac{\partial J}{\partial W^{(2)}} =$$

$$\frac{\partial J}{\partial B^{(2)}} =$$

$$\frac{\partial J}{\partial W^{(1)}} =$$

$$\frac{\partial J}{\partial B^{(1)}} =$$

Task 6:

Turn your partial derivatives into R code. This step might require some shuffling around of terms because the elements are all matrices and matrix multiplication requires that the inner dimensions match.

```r
# your code goes here
```

Gradient Descent

Task 7:

We will now apply the gradient descent algorithm to train our network. This simply involves repeatedly taking steps in the direction opposite of the gradient.

With each iteration, you will calculate the predictions based on the current values of the model parameters. You will also calculate the values of the gradient at the current values. Take a 'step' by subtracting a scalar multiple of the gradient. And repeat.

I will not specify what size scalar multiple you should use, or how many iterations need to be done. Just try things out. A simple way to see if your model is performing 'well' is to print out the predicted values of y-hat and see if they match closely to the actual values.

```
# your code goes here
```

Testing our trained model

Now that we have performed gradient descent and have effectively trained our model, it is time to test the performance of our network.

Task 8

Using the testing data, create predictions for the 30 observations in the test dataset. Print those results.

```
# your code goes here
```

How many errors did your network make?

Using package nnet

(You don't have to do anything for this part. Just read the documentation for the function `nnet()`.)

While instructive, the manual creation of a neural network is seldom done in production environments.

[Install the `nnet` package and NeuralNetTools] I've created a neural network for predicting the iris species based on the four numeric variables. We use the same training data to train the network. The function `nnet()` is smart enough to recognize that the values in the species column are a factor and will need to expressed in 0s and 1s as we did in our manually created network.

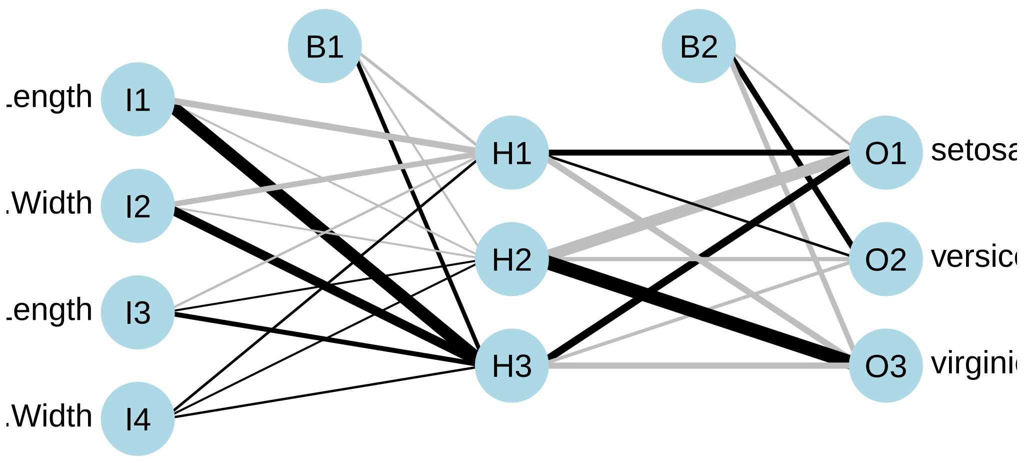
```
set.seed(1)
n <- dim(iris)[1]
rows <- sample(1:n, 0.8 * n)
train <- iris[rows, ]

library(nnet)
library(NeuralNetTools)
irismodel <- nnet(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
                    Petal.Width, size=3, data = train)
```

```
## # weights:  27
## initial  value 144.247567
## iter  10 value 51.531244
## iter  20 value 11.693157
## iter  30 value 4.794318
## iter  40 value 4.644015
## iter  50 value 4.642486
## iter  60 value 4.632611
## iter  70 value 4.625519
## iter  80 value 4.618919
## iter  90 value 4.616588
## iter 100 value 4.612555
## final  value 4.612555
## stopped after 100 iterations
```

Once we have created the network with nnet, we can use the predict function to make predictions for the test data.

```
plotnet(irismodel) # a plot of our network
```



```
# we can see that the predicted probability of each class matches the actual label
results <- predict(irismodel, iris[-rows,])
head(data.frame(results, actual = iris[-rows, 5]), 5)
```

```
##       setosa   versicolor    virginica actual
## 4  0.9999847 1.528832e-05 7.240628e-34 setosa
## 5  0.9999975 2.488093e-06 3.782015e-36 setosa
## 8  0.9999937 6.288797e-06 5.536253e-35 setosa
## 9  0.9999814 1.859289e-05 1.275695e-33 setosa
## 11 0.9999973 2.704618e-06 4.815116e-36 setosa
```

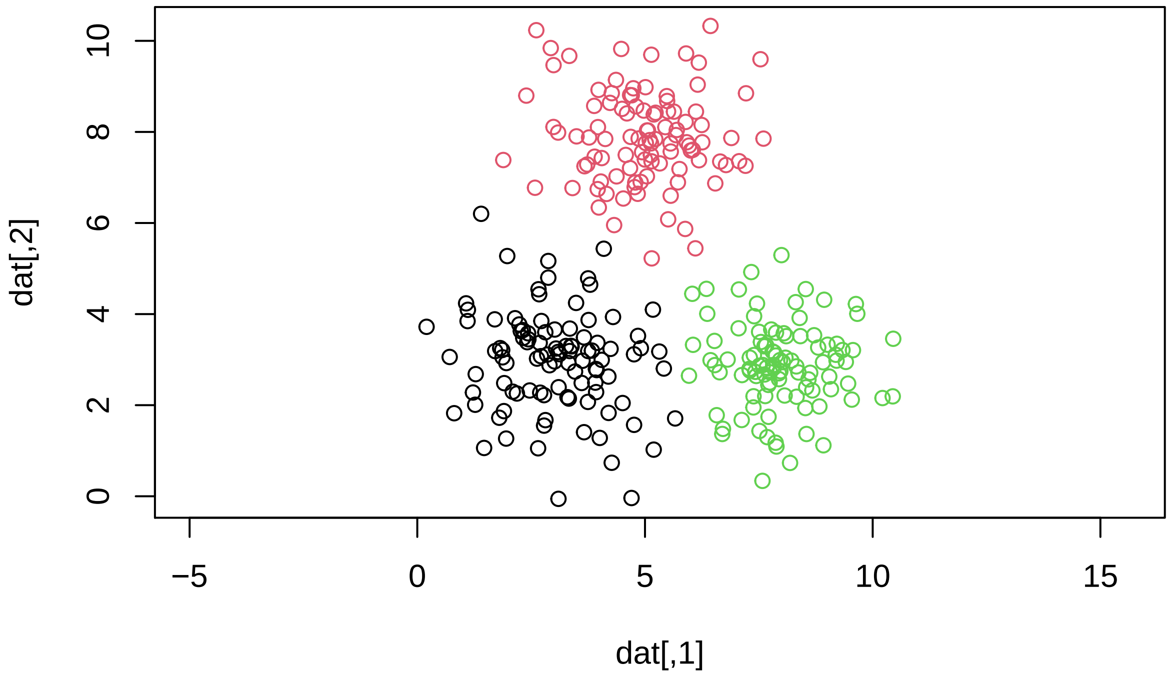
Part 2: K-means Clustering

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.

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

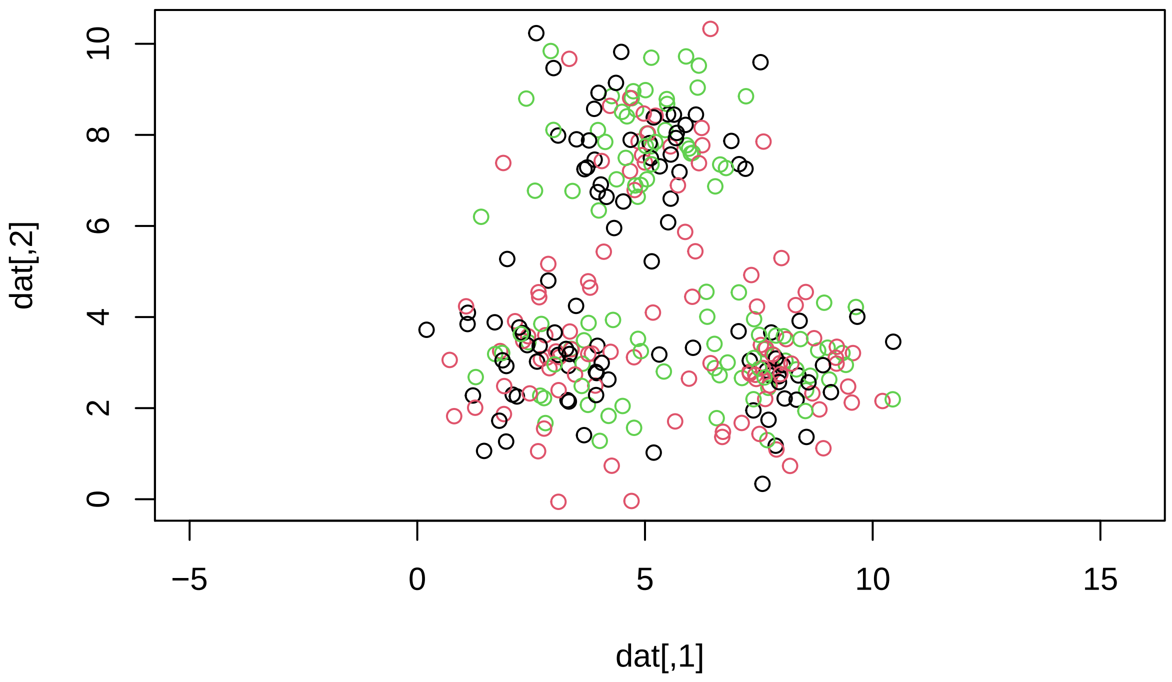
The true group labels are provided in the vector `true_groups`. Of course, you can't use that until the very end where you will perform some verification.

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.

```r
# 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
```



```r
# write your code here
distances <- function(point, means){
  # write code here
  # return(c(A = dist_to_centroid_A, B = dist_to_centroid_B, C = dist_to_centroid_C))
}

converged = FALSE
```

```r
while(!converged){
  # write code here...
  converged <- TRUE
}
```

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.

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:

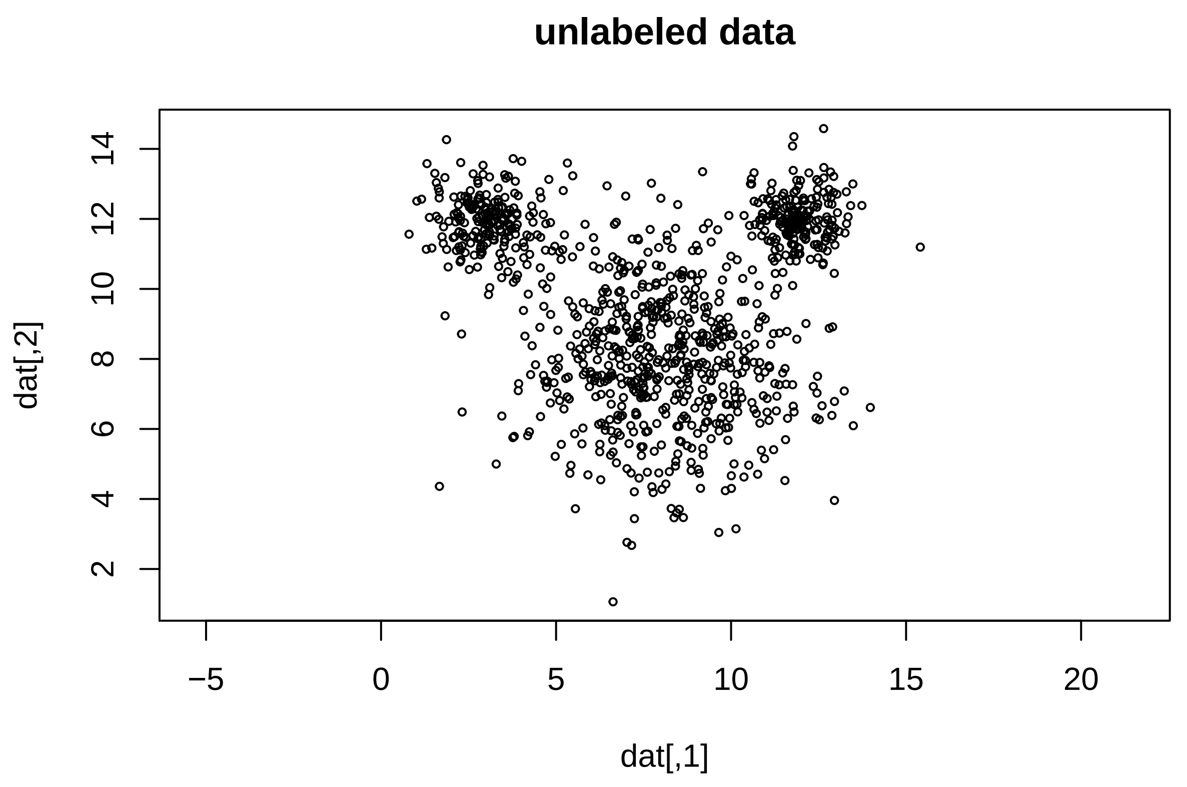
1) X is our observed data
2) Z is the missing data: the cluster to which the observations X belong.
3) X come from a normal distributions defined by the unknown parameters Θ (the mean μ and variance Σ).
4) Z is generated by a categorical distribution based on the unknown class mixing parameters α. ($\sum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

Thus,

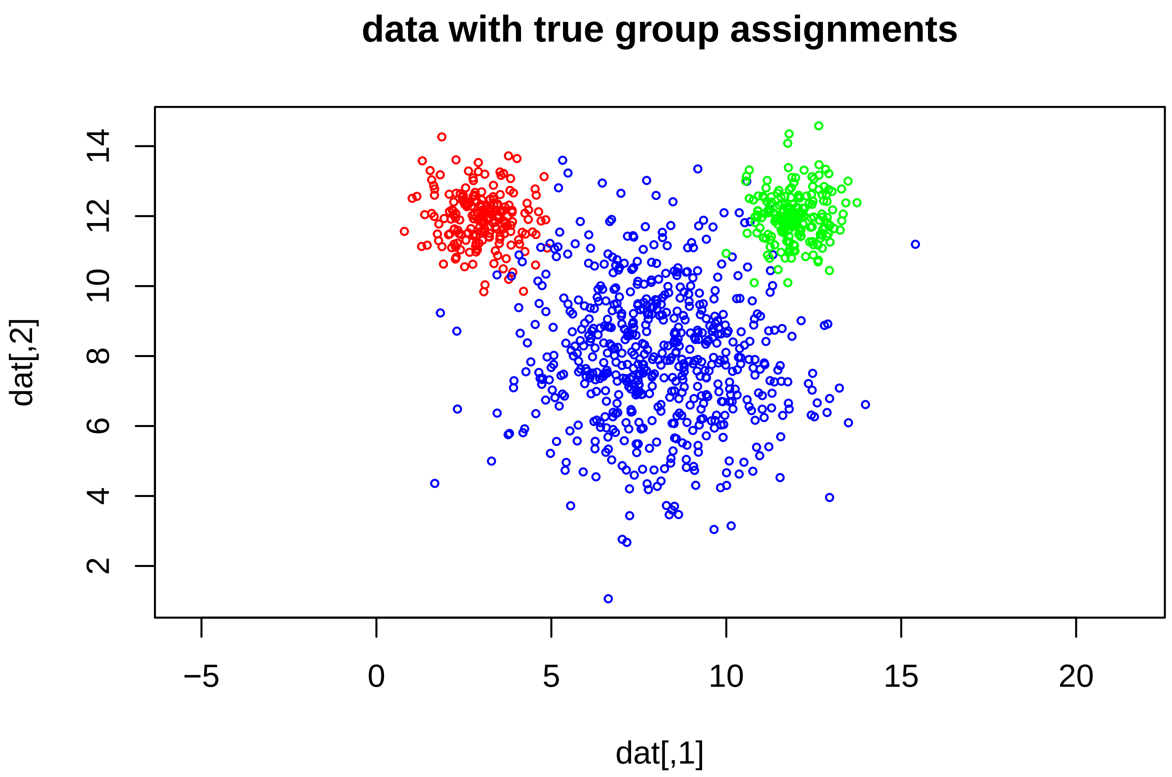
$$P(x|\Theta) = \sum_{k=1}^{K} \alpha_k P(X|Z_k, \theta_k)$$

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

```r
# 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)
```



```r
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 should just use `kmeans()`.)
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
```