Assesed Coursework 2

Henry Bourne

2023-01-10

EM algorithm

In this second assesed coursework for SC1 I will be investigating the EM algorithm and how it can be applied to find the Gaussian Mixture Model (GMM) that maximizes the likelihood. The GMM is a model comprised of the sum of several Gaussians, if we want to have a GMM comprising of K Gaussians then we have that $p(x) = \sum_{i=1,...,k} w_i \cdot N(x|\mu_i, \sigma_i^2)$, ie. the probability of an observation x is the weighted sum of Gaussian probability density functions. Each Gaussian, labelled k, has its own mean, μ_k, and its own variance, σ_i^2, where these are a vector and matrix in the multivariate case. The w_i's are the weights and represent the probability that a point comes from its corresponding Gaussian. The GMM has latent variables and although this isn't immediately obvious we can show that it does. Let z_j be the latent variable corresponding to observation x_j, where z_j is one-hot encoded with the i-th element of z_j, $z_j^{(i)}$ equal to one if the point x_j belongs to the i-th Gaussian. We have that:

$$p(z_j) = \prod_i (w_i)^{z_j^{(i)}} \tag{1}$$

$$p(x_j|z_j) = \prod_i N(x_j|\mu_i, \sigma_i)^{z_j^{(i)}} \tag{2}$$

By the above we can then right that:

$$p(x_j) = \sum_i p(z_j^{(i)} = 1) p(x_j|z_j^{(i)} = 1) \tag{3}$$

$$= \sum_i w_i \cdot N(x_j|\mu_i, \sigma_i) \tag{4}$$

So from the above we can see that there is in fact a latent variable in the GMM and that it is simply integrated/summed out. The parameters of the GMM therefore comprise of the means, $\mu_1, ..., \mu_k$, variances, $\sigma_1^2, ..., \sigma_k^2$, and weights, $w_1, ..., w_k$. Let θ denote a vector and let it contain all the parameters we just mentioned, we would like to find the maximum likelihood estimate of θ ie. $\text{argmax}_\theta p(D|\theta)$ where D is our dataset consisting of observations x_j.

As the GMM contains latent variables it makes it difficult to find an analytical solution and we must therefore rely on numerical methods. There exist many numerical methods we could use to try and find the maximum likelihood estimate including Newtons method, gradient descent and stochastic gradient descent among many others. The advantages of the EM algorithm are that at every iteration it is guaranteed that the maximum likelihood will not decrease, it does not require any derivatives and is computationally less expensive (although can be very slow). It also will result in estimates of the parameters that are feasible without the need for constraints during optimization, which is important here as we must have that the weights sum to one, $\sum_{i=1}^k w_i = 1$. We would like to do this from scratch and start by applying it to an easy dataset.

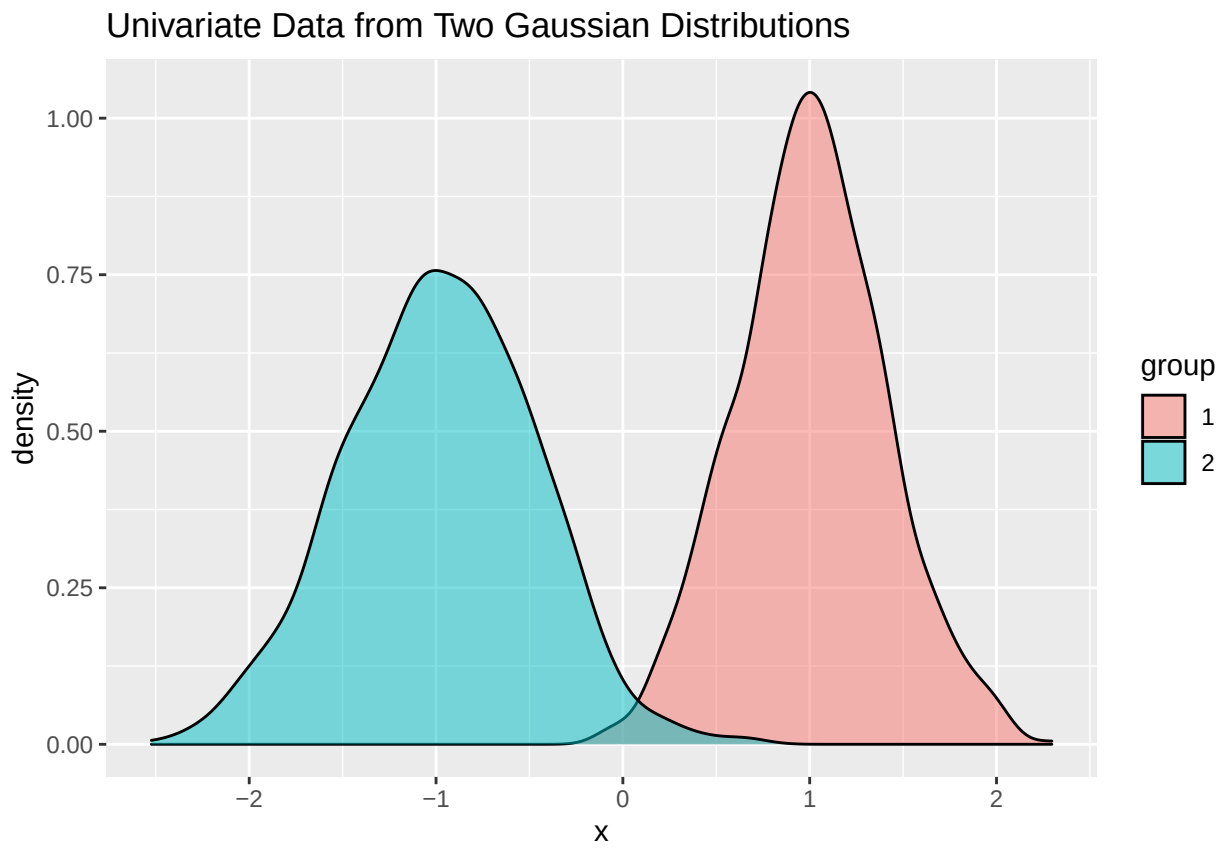
Note that throughout this R Markdown file we use techniques from the SC1 module. I will now point out a few of the many examples of topics from the module that have come in use during creating this document. For example the work is done in a R Markdown file so that text can be integrated throughout and a very readable

document produced which is an example of literate programming. Also note that all code is commented to give extra description as to what is going on at each step. The code has also been uploaded to my Github for easy access "github.com/h-aze" (Perhaps I should have saved and pushed to Github more frequently however as I lost a bunch of my code from not saving! a lesson for next time). Throughout the code I also use both vectorization and matrices to speed up computation, during creation I also had to do extensive amounts of debugging I also use profiling which I mention later on. Finally I also use *ggplot2* from the tidyverse. Now onto the EM algorithm!

A toy dataset

First we will create some data to test our functions piece by piece as we build them. We will create a binary classification dataset that is univariate and is generated using two Gaussian distributions, we will use the help of *ggplot2* from the *tidyverse* to plot.

```
# Set the seed for reproducibility
set.seed(123)

# Generate data from 2 different univariate Gaussian
# distributions
n <- 1000
Gauss_1 <- rnorm(n, 1, 0.4)
Gauss_2 <- rnorm(n, -1, 0.5)

# Combine the data into a single dataset and convert
# into a dataframe
Gauss <- c(Gauss_1, Gauss_2)
Gauss <- data.frame(x = Gauss, group = as.factor(c(rep(1,
    n), rep(2, n))))

# Plot the data using ggplot2
library(ggplot2)
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    ggtitle("Univariate Data from Two Gaussian Distributions")
```



Initialization

The first thing to do if we would like to use the EM algorithm to fit a GMM to this data is to have some initial parameters from which to start. We need a mean, standard deviation, and weight for each Gaussian. One way in which we can initialize the parameters is by randomly subsetting the data and calculating the means and standard deviations of the subsets and setting the weights equal to each other, we write a function that does exactly this here:

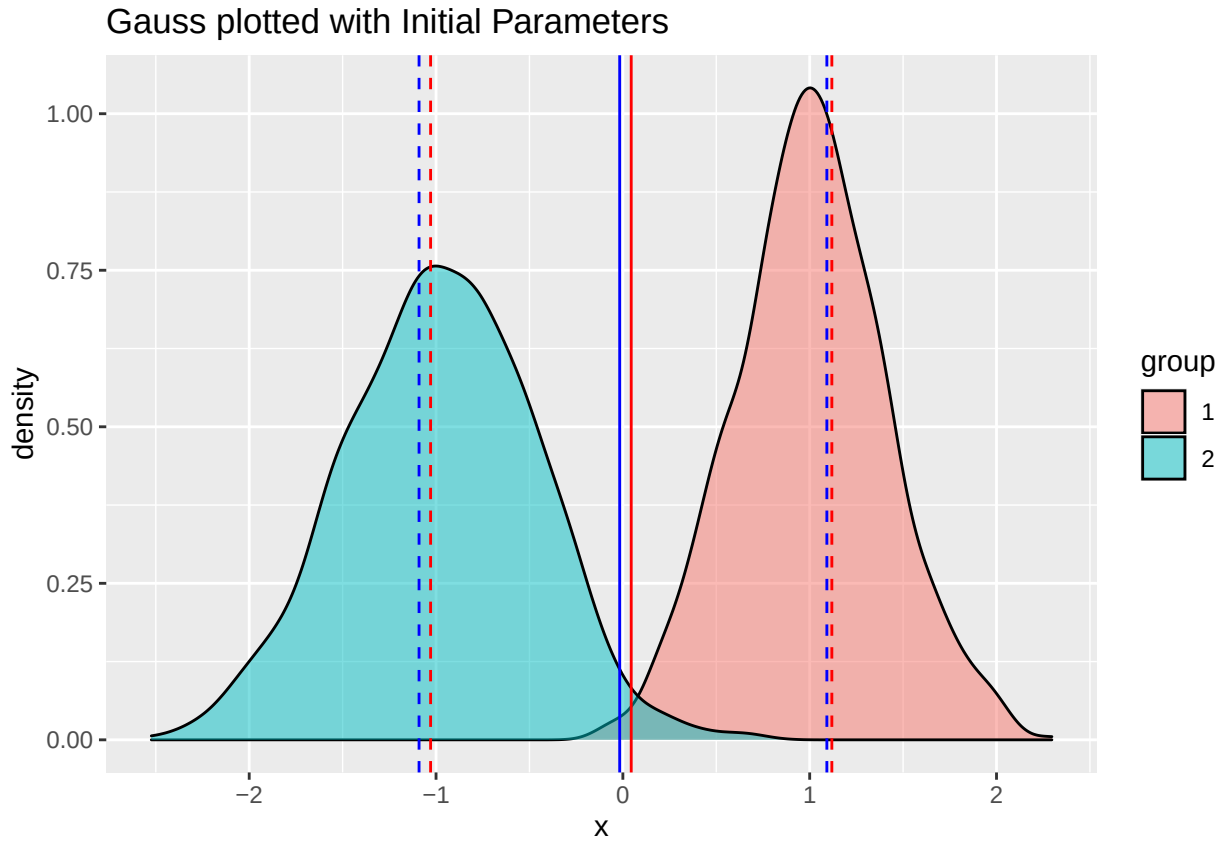
```
init_params_subset <- function(data, k) {
    data <- data[sample(length(data))]  # We shuffle the data

    # Initialize the mean
    mu <- aggregate(data, list(rep_len(1:k, length(data))),
        mean)[, -1]

    # Initialize the standard deviation
    sigma <- aggregate(data, list(rep_len(1:k, length(data))),
        sd)[, -1]

    # Initialize the weights
    weight <- rep(1/k, k)

    list(mu = mu, sigma = sigma, weight = weight)
}
```

Let's now plot the same graph we had before but this time with the initial parameters laid on top to see how accurate they are:

```
par <- init_params_subset(Gauss[, -2], 2)
par
```

```
## $mu
## [1]  0.04470246 -0.01701869
##
## $sigma
## [1] 1.073838 1.108917
##
## $weight
## [1] 0.5 0.5
```

```
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[1], color = "red", linetype = "solid") +
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = par$mu[2], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[1], color = "blue", linetype = "dashed") +
    ggtitle("Gauss plotted with Initial Parameters")
```



The solid red line indicates the initial mean we have generated and the red dashed lines one standard deviation (the one we generated) from this mean for Gaussian 1, similarly the blue lines correspond to the initialization for Gaussian 2. We see that the means and standard deviations do accurately reflect the center and spread of the entire dataset, however, aren't particularly representative of either of the Gaussians.

There are many ways we could initialize our parameters, here we define a few more ways we could do so and at the bottom provide a function for initializing the parameters with any method simply by specifying the method name:

```
init_params_kmeans <- function(data, k) {
    # We perform k means
    clusters <- kmeans(data, k)

    # Initialize the mean with the centers found with
    # kmeans
    mu <- clusters$centers

    # Initialize the standard deviation by finding the
    # standard deviation of each cluster
    sigma <- aggregate(data, list(clusters$cluster), sd)[,
        -1]

    # Initialize the weights
    weight <- rep(0, k)
    for (i in 1:k) {
        weight[i] <- sum(clusters$cluster == i)/length(data)
    }

    list(mu = mu, sigma = sigma, weight = weight)
}

init_params <- function(data, k, method = "subset") {
    if (method == "subset") {
        params <- init_params_subset(data, k)
    } else if (method == "kmeans") {
        params <- init_params_kmeans(data, k)
    } else {
        stop("Didn't give valid method for initializing parameters")
    }
    params
}
```

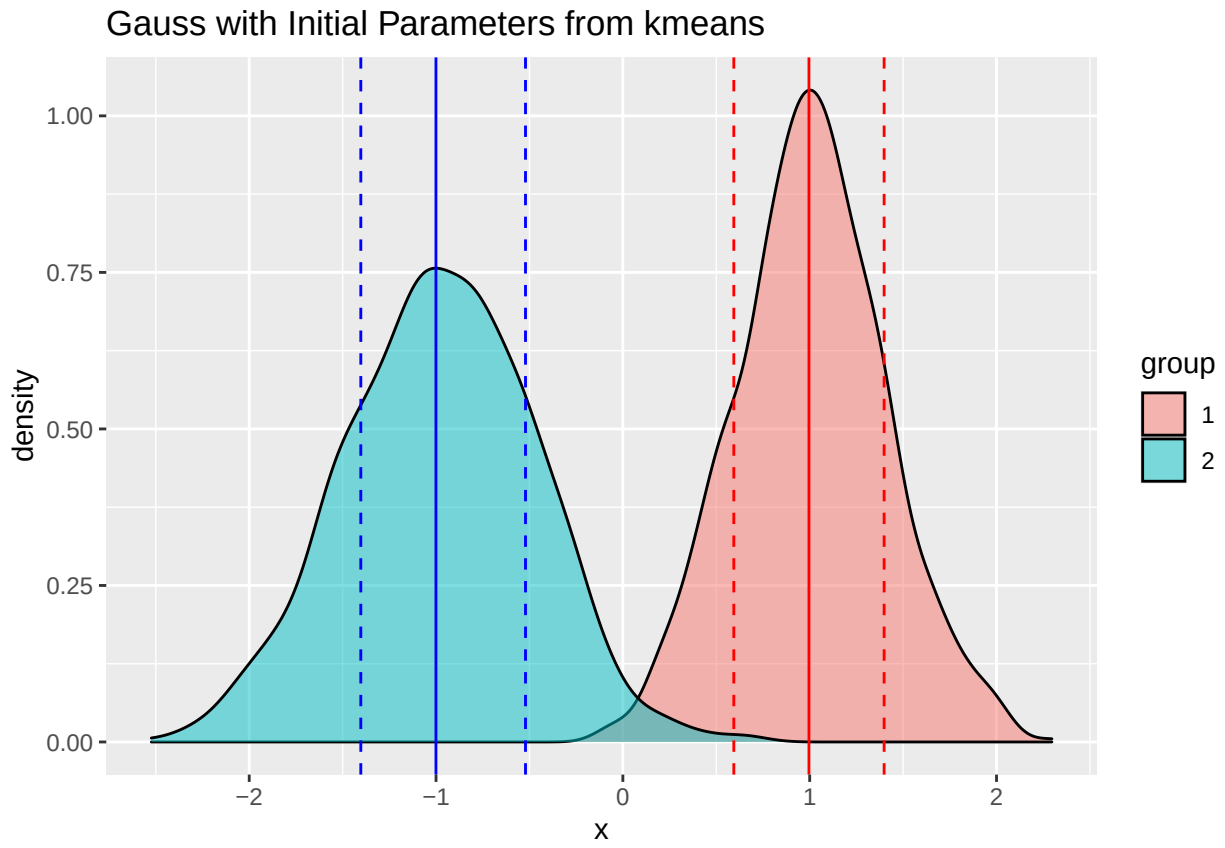
So for example if we wanted to use the kmeans method of initialization, which is where we use the results of k means to inform our initialization we can simply write the following:

```
par <- init_params(Gauss[, -2], 2, method = "kmeans")
par
```

```
## $mu
##        [,1]
## 1  0.996107
## 2 -1.000367
##
## $sigma
## [1] 0.4021855 0.4791955
##
## $weight
## [1] 0.508 0.492
```

```
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[1], color = "red", linetype = "solid") +
```

```
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = par$mu[2], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[1], color = "blue", linetype = "dashed") +
    ggtitle("Gauss with Initial Parameters from kmeans")
```



Here we see that we are given very good initial parameters for the EM algorithm to start with, in fact so good that we basically have what the EM algorithm would come up with after being run. So for harder problems it may be beneficial to use kmeans as a starting point for the EM algorithm. As a side note one might wonder why we need the EM algorithm if we could just use kmeans? the quick answer is that the EM algorithm also models variance and is a probabilistic model.

E-step

Now we have our initial parameters we can move onto the EM algorithm itself, the first step of the EM algorithm being the E-step. The E-step is the expectation step and in the general sense what we are trying to do at this step is update our lower bound function on the maximum likelihood. We achieve this when we minimize the KL divergence between the latent variable distribution and its posterior which happens when the latent variable distribution is its posterior. When looking specifically at the problem of fitting a GMM we find that the expected value of the latent variable under the posterior distribution is given by:

$$p_{j,i} = \frac{w_i N(x_j|\mu_i, \sigma_i)}{\sum_{r=1}^{k} w_r N(x_j|\mu_r, \sigma_r)} \tag{5}$$

Hence to perform the E-step all we must do is calculate the matrix $P = [p_{j,i}]$. Now we know what the E-step is, let's implement it:

```
E_step <- function(data, mu, sigma, weight) {
    k <- length(mu)
    n <- length(data)
    p <- matrix(NA, nrow = n, ncol = k)

    # Loop for each Gaussian
    for (i in 1:k) {
        # Compute the probability of each data-point
        # coming from Gaussian i given current
        # parameters for Gaussian i
        p[, i] <- weight[i] * dnorm(data, mu[i], sigma[i])
    }
    # For each data point sum probabilities of
    # belonging to each Gaussian
    s <- rowSums(p)

    # We return the posterior
    p/s
}
```

And let's test it:

```
post <- E_step(Gauss[, -2], par$mu, par$sigma, par$weight)
head(post)
```

```
##           [,1]         [,2]
## [1,] 0.9990196 9.804183e-04
## [2,] 0.9997003 2.996840e-04
## [3,] 0.9999992 8.472679e-07
## [4,] 0.9998953 1.046949e-04
## [5,] 0.9999145 8.550153e-05
## [6,] 0.9999995 5.303436e-07
```

Printing just the first few rows we see that we obtain a matrix with two columns where for each datapoint the first column tells us the (posterior) probability that it came from Gaussian 1 and column 2 the probability it came from Gaussian 2, note these probabilities sum to 1.

M-step

Now we have completed the E-step we need to perform the M-step, or the maximization step. Given the posterior probabilities we have now obtained in the E-step we now need to compute the values of the parameters that maximize the likelihood according to the current posterior distribution. Again here we are trying to maximize the lower bound function, but here we fix the latent distribution and now try to maximize it wrt. our parameters (mu, sigma and weight). We can work out that the lower bound function is maximized when:

$$w_i = \frac{N_i}{n} \tag{6}$$

$$\mu_i = \frac{1}{N_i} \sum_{j=1}^{n} p_{j,i} x_j \tag{7}$$

$$\sigma_i^2 = \frac{1}{N_i} \sum_{j=1}^{n} p_{j,i} (x_j - \mu_i)^2 \tag{8}$$

Where n is the number of observations and $N_i = \sum_{j=1}^{n} p_{j,i}$. Now we can implement the M-step:

```
M_step <- function(data, posteriors) {
    k <- ncol(posteriors)
    n <- length(data)

    mu <- rep(NA, k)
    sigma <- rep(NA, k)
    weight <- rep(NA, k)

    N <- colSums(posteriors)
    for (i in 1:k) {
        # Update mu
        mu[i] <- sum(posteriors[, i] * data)/N[i]

        # Update sigma
        sigma[i] <- sqrt(sum(posteriors[, i] * (data - mu[i])^2)/N[i])

        # Update weights
        weight[i] <- N[i]/n
    }

    list(mu = mu, sigma = sigma, weight = weight)
}
```

Let's now test this function:

```
par <- M_step(Gauss[, -2], post)
par
```

```
## $mu
## [1]  0.9997246 -0.9883821
##
## $sigma
## [1] 0.4031437 0.4943125
##
## $weight
## [1] 0.5041098 0.4958902
```

We see that by running the M-step we have updated our parameters and their values now vary slightly to our initial values.

Convergence

Now we have most of the pieces necessary wo build the EM algorithm: we have values to start from, we have the E-step, we have the M-step. Now all that remains is determining when we should finish iterating between the E and the M step. We will set a maximum number of iterations that the EM algorithm will perform, but what if it converges a long way before this maximum number of iterations? preferably we would like to stop as soon as it converges. There are multiple ways one can measure convergence for the EM algorithm, we will discuss using the likelihood as a convergence measure. As the EM algorithm is used to find the maximum likelihood estimate of the parameters a good way to measure convergence is to check how the log likelihood is changing. Specifically we will compute the log likelihood after each iteration and measure the difference, once the change in the log likelihood is below some threshold ϵ we will take this to mean that it is has converged and stop iterating. First we need a way of computing the log likelihood:

```r
neg_log_L <- function(data, mu, sigma, weight) {
    k <- length(mu)
    lL <- 0
    for (i in 1:k) {
        lL <- lL + (weight[i] * dnorm(data, mu[i], sigma[i]))
    }
    -sum(log(lL))
}
```

Let's test this function works by evaluating it on the data and current parameters:

```r
neg_log_L(Gauss[, -2], par$mu, par$sigma, par$weight)
```

```
## [1] 2544.906
```

Now all we have to do is keep track of the negative log likelihood and see if the change in it gets bellow some threshold, ϵ. Note that here we are in fact computing the negative log likelihood so we will expect this to decrease with each iteration (as we are expecting the log likelihood to increase). There are also other ways we could test convergence such as measuring the change in the parameters, in particular it has been argued that the best way to test for convergence is to measure the change in the variance parameters as these are the last to stabalize.

Putting it all together

Now we will put everything we have talked about together to create the EM algorithm:

```r
EM_iter <- function(data, params, eps = 1e-05, max_iter = 1000) {
    iter <- 0
    prev_lL <- Inf
    curr_lL <- neg_log_L(data, params$mu, params$sigma,
        params$weight)
    neg_log_liks <- rep(NA, max_iter)

    while ((prev_lL - curr_lL > eps) & (iter < max_iter)) {
        posteriors <- E_step(data, params$mu, params$sigma,
            params$weight)
        params <- M_step(data, posteriors)

        prev_lL <- curr_lL
        curr_lL <- neg_log_L(data, params$mu, params$sigma,
            params$weight)
        iter <- iter + 1
        neg_log_liks[iter] <- prev_lL
    }

    neg_log_liks[iter + 1] <- curr_lL
    neg_log_liks <- neg_log_liks[!is.na(neg_log_liks)]
    min_neg_log_lik <- curr_lL
    if (min(neg_log_liks) != min_neg_log_lik) {
        warning("The minimum log likelihood achieved during optimization is not the last value of the n
    }

    list(params = params, neg_log_liks = neg_log_liks, min_neg_log_lik = min_neg_log_lik)
}
```

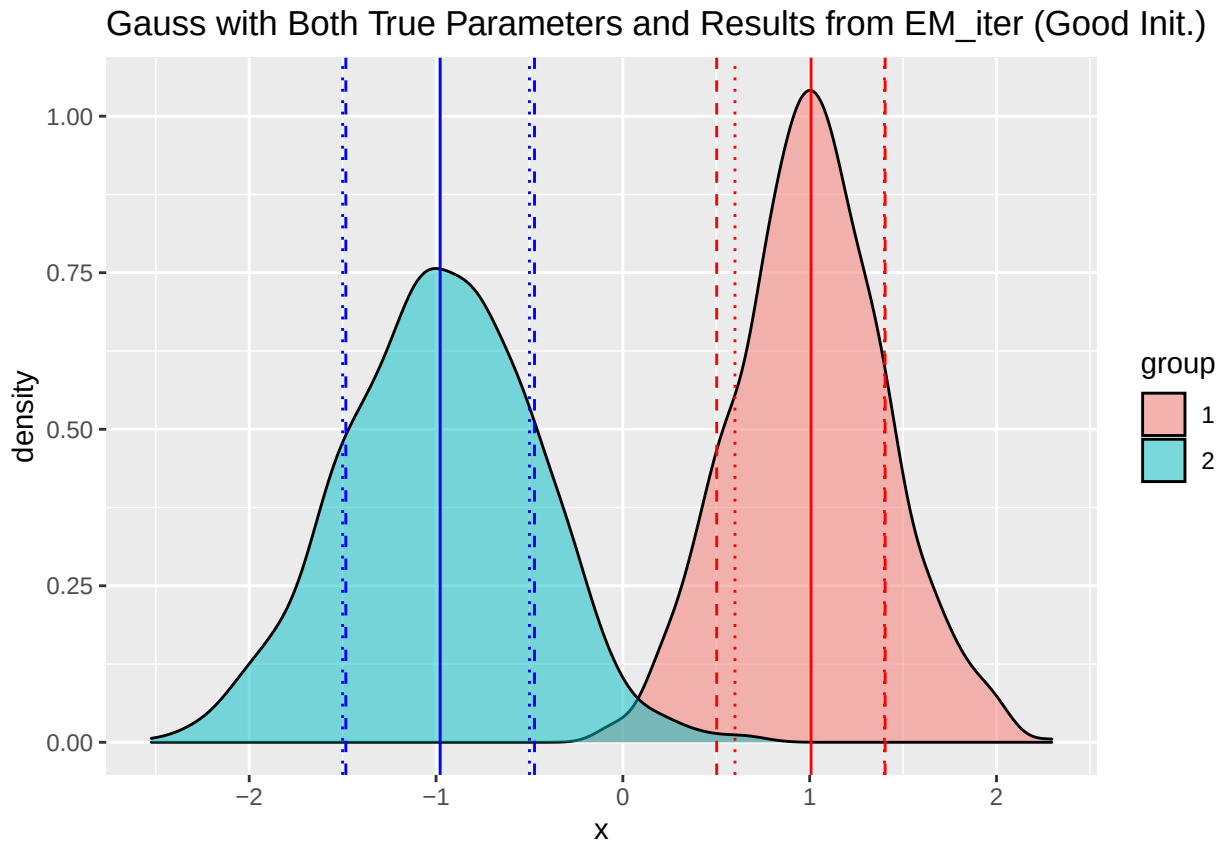
Now let's initialize some parameters and test our EM algorithm on our data:

```
set.seed(123)
par <- init_params(Gauss[, -2], 2, method = "kmeans")
out <- EM_iter(Gauss[, -2], par)
par <- out$params
par
```

```
## $mu
## [1] -0.9778135  1.0070871
##
## $sigma
## [1] 0.5049785 0.3966770
##
## $weight
## [1] 0.5004005 0.4995995
```

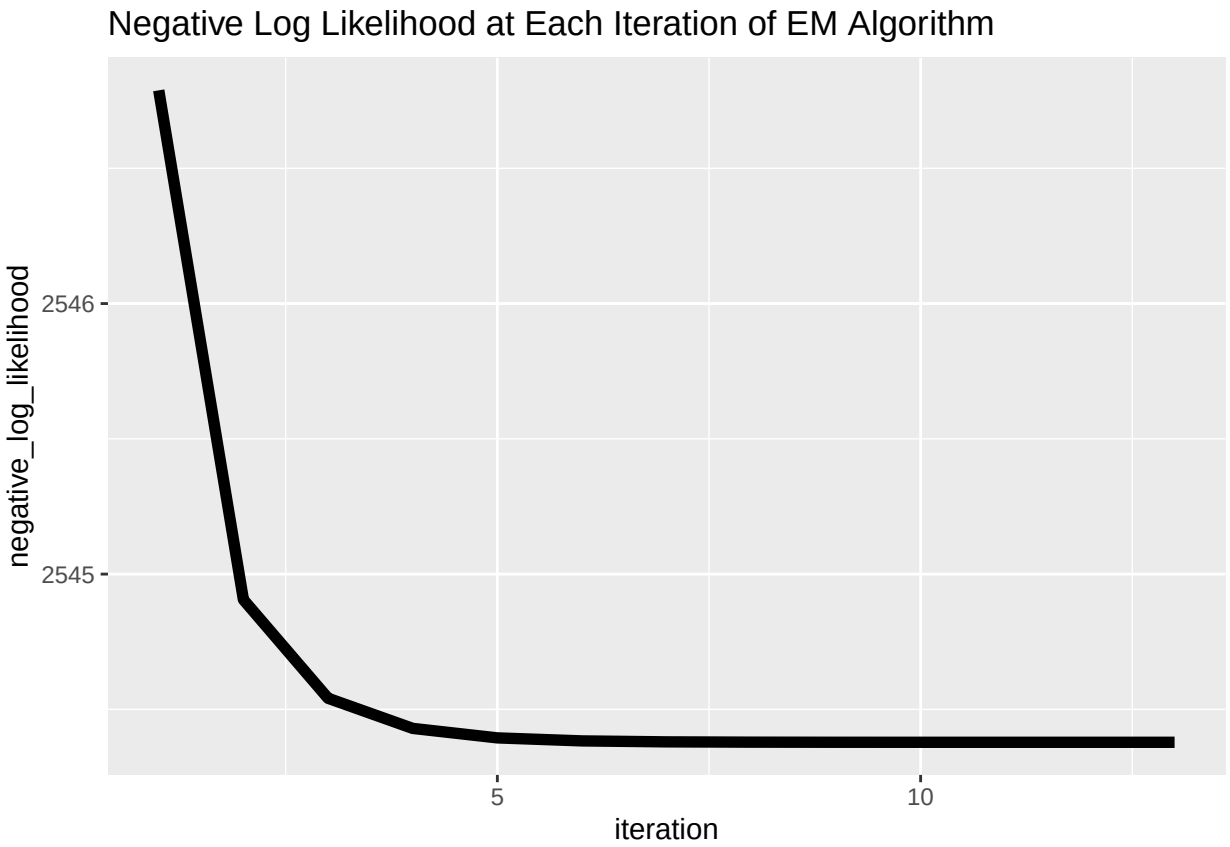
First let's draw attention to the values we have obtained for the weights, we notice that they are very close in value which reflects the true structure of the data, ie. that half the data points come from one Gaussian and half from the other. Now let's plot the our data along with the means and standard deviations we have obtained from running the EM algorithm:

```
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[1], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "blue", linetype = "dashed") +
    geom_vline(xintercept = -1 - 0.5, color = "blue", linetype = "dotted") +
    geom_vline(xintercept = -1 + 0.5, color = "blue", linetype = "dotted") +

geom_vline(xintercept = par$mu[2], color = "red", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = 1 - 0.4, color = "red", linetype = "dotted") +
    geom_vline(xintercept = 1 + 0.4, color = "red", linetype = "dotted") +

ggtitle("Gauss with Both True Parameters and Results from EM_iter (Good Init.)")
```



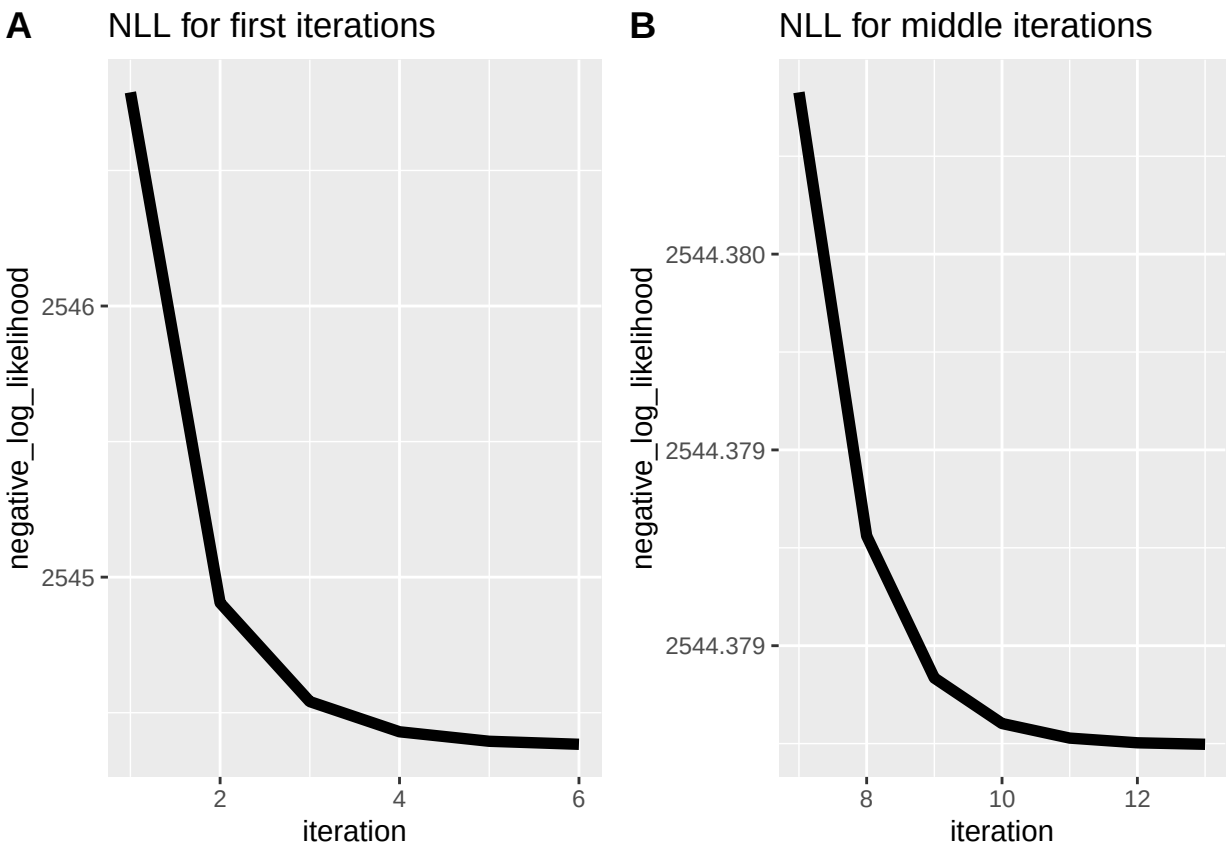
We see that each Gaussian in our mixture model has centered itself on one of the Gaussians that underlie the dataset and it also has produced standard deviations that much up well, the dashed lines again represent one standard deviation from the mean for our estimated GMM and the dotted lines represent the true standard deviation from the true mean. To make sure everything is going smoothly during each iteration in the EM algorithm let us also plot the negative log likelihood of each iteration:

```
nll_over_iters <- data.frame(negative_log_likelihood = out$neg_log_liks,
    iteration = 1:length(out$neg_log_liks))
ggplot(nll_over_iters, aes(x = iteration, y = negative_log_likelihood)) +
    geom_line(color = "black", size = 2) + ggtitle("Negative Log Likelihood at Each Iteration of EM Alg
```



We see that the negative log likelihood appears to be decreasing throughout, however, it is not very clear due to a initial rapid decrease. Let's divide the iterations up and plot again to make it more clear:

```
library(cowplot)

first <- ggplot(nll_over_iters[1:6, ], aes(x = iteration,
    y = negative_log_likelihood)) + geom_line(color = "black",
    size = 2) + ggtitle("NLL for first iterations")

second <- ggplot(nll_over_iters[7:nrow(nll_over_iters),
    ], aes(x = iteration, y = negative_log_likelihood)) +
    geom_line(color = "black", size = 2) + ggtitle("NLL for middle iterations")

plot_grid(first, second, labels = "AUTO")
```



Here we can see very clearly that in each of these plots the negative log likelihood is continually decreasing, meaning that the EM algorithm at each iteration is producing a larger maximum likelihood which is what we expect from the EM algorithm. So it appears we have a fully functioning EM algorithm!

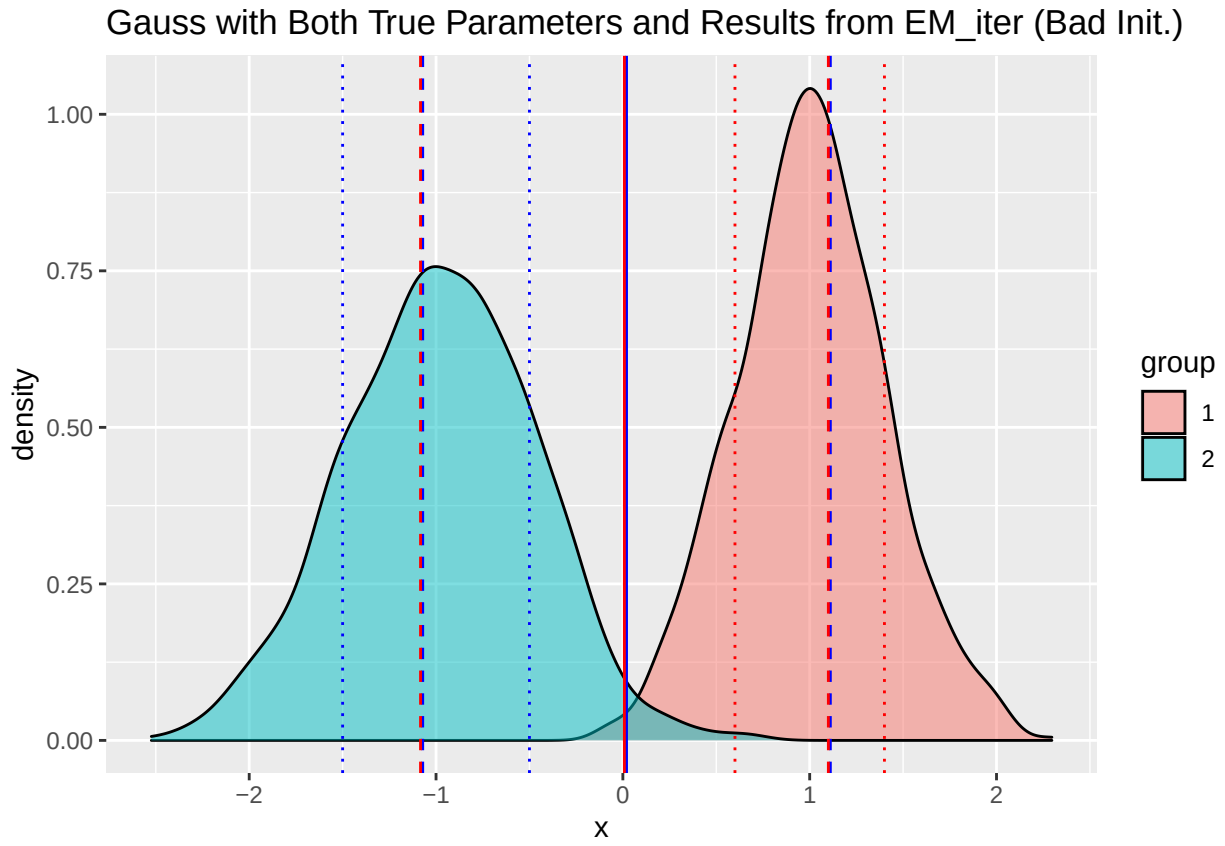
However, we did get lucky here, let's look at a different case where we aren't so lucky:

```
set.seed(84792)
par <- init_params(Gauss[, -2], 2)
out <- EM_iter(Gauss[, -2], par)
par <- out$params
par
```

```
## $mu
## [1] 0.019598923 0.008084766
##
## $sigma
## [1] 1.090919 1.091869
##
## $weight
## [1] 0.5000036 0.4999964
```

```
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[1], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "blue", linetype = "dashed") +
    geom_vline(xintercept = -1 - 0.5, color = "blue", linetype = "dotted") +
    geom_vline(xintercept = -1 + 0.5, color = "blue", linetype = "dotted") +
```

```
geom_vline(xintercept = par$mu[2], color = "red", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = 1 - 0.4, color = "red", linetype = "dotted") +
    geom_vline(xintercept = 1 + 0.4, color = "red", linetype = "dotted") +

ggtitle("Gauss with Both True Parameters and Results from EM_iter (Bad Init.)")
```



Here we set the random seed to give us a "bad initialization", we see that this "bad initialization" leads to a bad result, ie. a GMM that doesn't model the data very well at all. This is a common problem encountered when working with the EM algorithm, to remedy it we rerun the EM algorithm multiple times and select the parameters that give the smallest negative log likelihood. Hence our reason for naming the function *EM_iter*, we now create a function (namely *EM*) that will run *EM_iter* multiple times:

```
EM <- function(data, k, iters = 50, init_method = "subset",
    eps = 1e-05, max_iter = 1000) {
    best_lL <- Inf
    best_out <- NULL
    for (i in 1:iters) {
        params <- init_params(data, k, method = init_method)
        out <- EM_iter(data, params, eps = eps, max_iter = max_iter)
        if (out$min_neg_log_lik < best_lL) {
            best_lL <- out$min_neg_log_lik
            best_out <- out
        }
```

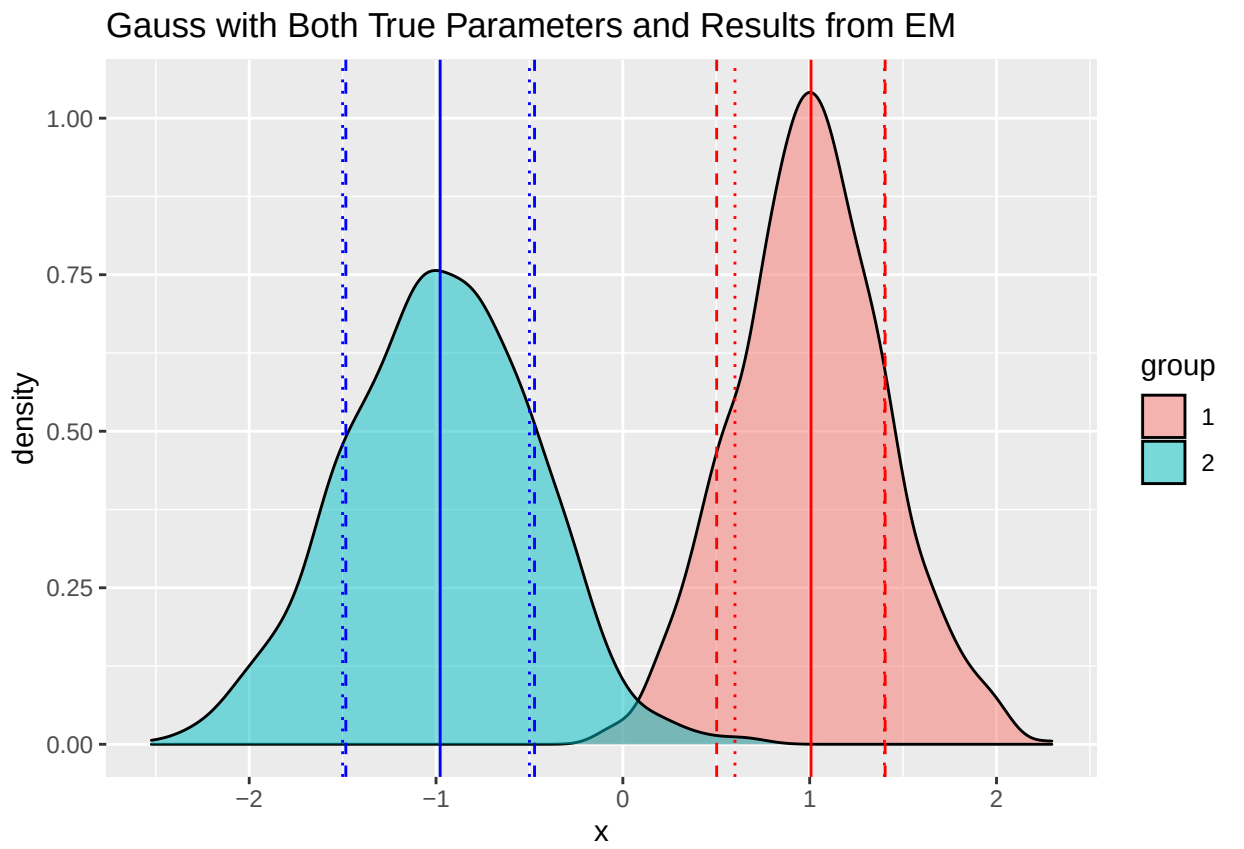
```
    }
    best_out
}
```

Let's put this function to the test:

```
set.seed(123)  # So we are talking about the same thing!

out <- EM(Gauss[, -2], 2)
par <- out$params
par
```

```
## $mu
## [1] -0.9777755  1.0071143
##
## $sigma
## [1] 0.5050159 0.3966523
##
## $weight
## [1] 0.5004169 0.4995831
```

```
ggplot(Gauss, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[1], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "blue", linetype = "dashed") +
    geom_vline(xintercept = -1 - 0.5, color = "blue", linetype = "dotted") +
    geom_vline(xintercept = -1 + 0.5, color = "blue", linetype = "dotted") +

geom_vline(xintercept = par$mu[2], color = "red", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = 1 - 0.4, color = "red", linetype = "dotted") +
    geom_vline(xintercept = 1 + 0.4, color = "red", linetype = "dotted") +

ggtitle("Gauss with Both True Parameters and Results from EM")
```



Now we are running the EM algorithm multiple times and then outputting the results of the run that gave us the minimum negative log likelihood (maximum log likelihood), this should avoid us from getting bad models from converging on local optima. As we see the result we get is good and matches the underlying distribution very well.

Compute

In order to makes sure the above functions are efficient I took time to profile the EM algorithm (with the help of profvis) to check its efficiency. On profiling the EM algorithm there were a few takeaways and improvements made to the above. First of all *apply* functions were replaced with for loops as they were faster to compute, second of all I noticed that it spent alot of time in the *dnorm* and in the *colSums* and *rowSums* functions.

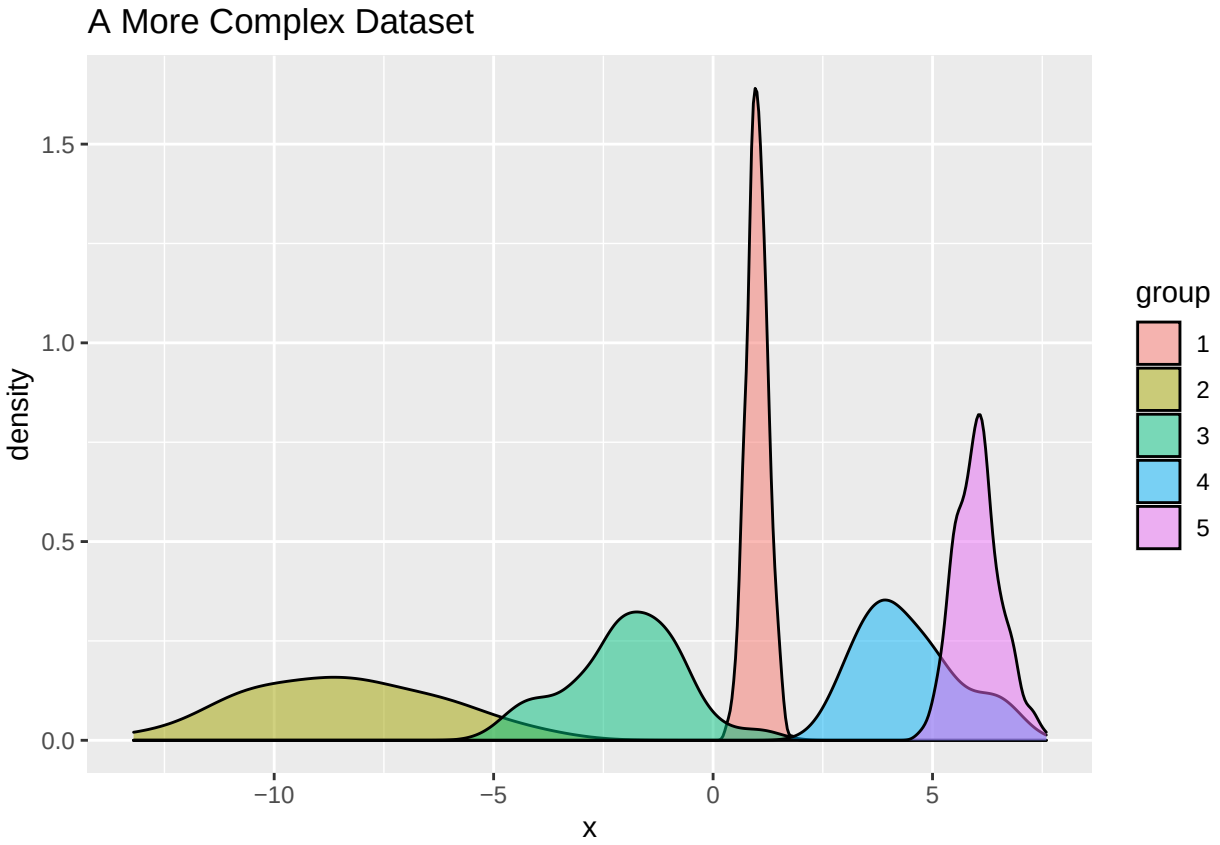
We note that *rowSums* causes $2n^2$ data load from RAM to cache and *colSums* only $n^2 + n$ so *colSums* is more memory efficient. However, we use both equally, once in the E-step and once in the M-step. So it wouldn't make sense to transpose our matrix, or create the matrix differently, given the extra computation incurred by doing so. Hence, there is not much we could do here except write our own version in C which may be quicker (there exist implementations of *colSums* and *rowSums* that are more efficient). Now, turning our attention to *dnorm* we note that *dnorm* is implemented in C code and very efficient, so there is not much we could do here to speed this up.

So from profiling we identified that *apply* functions should be replaced with for loops, which has been done, but apart from this it doesn't appear much else can be done for the sake of increasing efficiency without massively increasing complexity. Now that we've analysed efficiency we can move onto a slightly harder dataset.

More Complex Example

Finally to test the algorithm on something slightly more complex we will create a dataset that has more Gaussians with more varying parameters, more overlap and an uneven number of datapoints per Gaussian:

```
# Set the seed for reproducibility
set.seed(123)

# Generate data from 5 different univariate Gaussian
# distributions
n1 <- 600
n2 <- 40
n3 <- 200
n4 <- 15
n5 <- 300
Gauss_1 <- rnorm(n1, 1, 0.25)
Gauss_2 <- rnorm(n2, -8, 2)
Gauss_3 <- rnorm(n3, -2, 1.3)
Gauss_4 <- rnorm(n4, 4, 1)
Gauss_5 <- rnorm(n5, 6, 0.5)

# Combine the data into a single dataset and convert
# into a dataframe
Gauss2 <- c(Gauss_1, Gauss_2, Gauss_3, Gauss_4, Gauss_5)
Gauss2 <- data.frame(x = Gauss2, group = as.factor(c(rep(1,
    n1), rep(2, n2), rep(3, n3), rep(4, n4), rep(5, n5))))

# Plot the data using ggplot2
library(ggplot2)
ggplot(Gauss2, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    ggtitle("A More Complex Dataset")
```



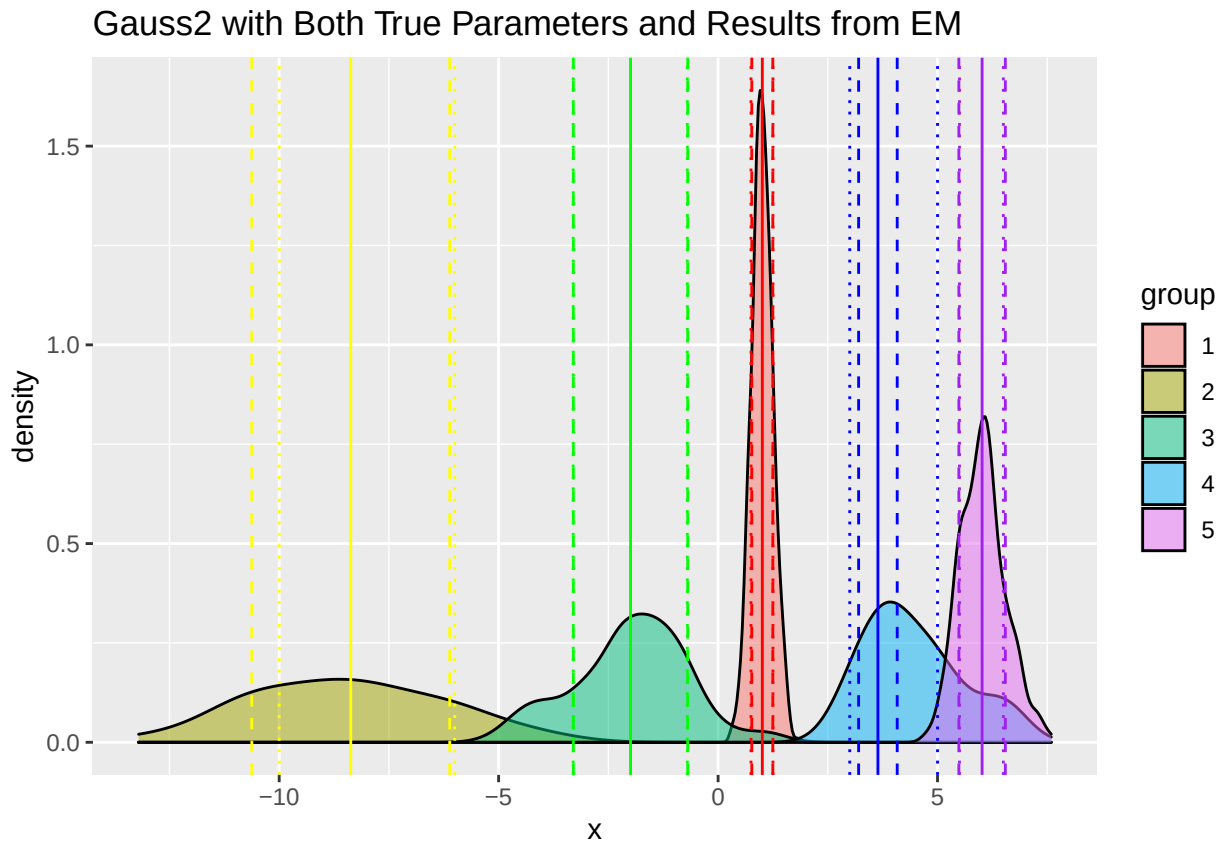
This new data set contains more Gaussians (5 instead of 2), more variability in standard deviation and have very different means. They also vary widely in the number of samples we have for each one from just 15 datapoints for Gaussian 4 and 600 for Gaussian 1. This should present more of a challenge to our EM algorithm, let's test it now:

```
set.seed(123)  # So we are talking about the same thing!

out <- EM(Gauss2[, -2], 5)
par <- out$params
par
```

```
## $mu
## [1]  1.007589 -8.368623 -1.994526  3.641798  6.015471
##
## $sigma
## [1] 0.2395872 2.2546328 1.3029869 0.4397508 0.5294799
##
## $weight
## [1] 0.519119173 0.035206026 0.172990980 0.006997982 0.265685839
```

```
ggplot(Gauss2, aes(x, fill = group)) + geom_density(alpha = 0.5) +
    geom_vline(xintercept = par$mu[4], color = "blue", linetype = "solid") +
    geom_vline(xintercept = par$mu[4] + par$sigma[4], color = "blue",
        linetype = "dashed") + geom_vline(xintercept = par$mu[4] -
    par$sigma[4], color = "blue", linetype = "dashed") +
    geom_vline(xintercept = 4 - 1, color = "blue", linetype = "dotted") +
    geom_vline(xintercept = 4 + 1, color = "blue", linetype = "dotted") +
```

```
geom_vline(xintercept = par$mu[1], color = "red", linetype = "solid") +
    geom_vline(xintercept = par$mu[1] + par$sigma[1], color = "red",
        linetype = "dashed") + geom_vline(xintercept = par$mu[1] -
    par$sigma[1], color = "red", linetype = "dashed") +
    geom_vline(xintercept = 1 - 0.25, color = "red", linetype = "dotted") +
    geom_vline(xintercept = 1 + 0.25, color = "red", linetype = "dotted") +

geom_vline(xintercept = par$mu[5], color = "purple", linetype = "solid") +
    geom_vline(xintercept = par$mu[5] + par$sigma[5], color = "purple",
        linetype = "dashed") + geom_vline(xintercept = par$mu[5] -
    par$sigma[5], color = "purple", linetype = "dashed") +
    geom_vline(xintercept = 6 - 0.5, color = "purple", linetype = "dotted") +
    geom_vline(xintercept = 6 + 0.5, color = "purple", linetype = "dotted") +

geom_vline(xintercept = par$mu[3], color = "green", linetype = "solid") +
    geom_vline(xintercept = par$mu[3] + par$sigma[3], color = "green",
        linetype = "dashed") + geom_vline(xintercept = par$mu[3] -
    par$sigma[3], color = "green", linetype = "dashed") +
    geom_vline(xintercept = -2 - 1.3, color = "green", linetype = "dotted") +
    geom_vline(xintercept = -2 + 1.3, color = "green", linetype = "dotted") +

geom_vline(xintercept = par$mu[2], color = "yellow", linetype = "solid") +
    geom_vline(xintercept = par$mu[2] + par$sigma[2], color = "yellow",
        linetype = "dashed") + geom_vline(xintercept = par$mu[2] -
    par$sigma[2], color = "yellow", linetype = "dashed") +
    geom_vline(xintercept = -8 - 2, color = "yellow", linetype = "dotted") +
    geom_vline(xintercept = -8 + 2, color = "yellow", linetype = "dotted") +

ggtitle("Gauss2 with Both True Parameters and Results from EM")
```



We see from the plot that our means match up pretty well with the peaks of the distributions (and also lie roughly half way inbetween the true standard deviation lines), except for Gaussian 4 where it accurately gets the peak of the data however the true mean does not lie at the peak. The standard deviations also match up well with the true standard deviations for the most part, with the standard deviations for Guassian 4 being too small and skewed due to the inaccurate mean. We also note that for group 2 the mean we have estimated is slightly too much to the left and the standard deviation slighty too big, however, by not a large amount.

Now moving onto the weights we note that its done a pretty good job at estimating the probability for which Gaussian a point comes from assigning small weights to Gaussians 2 and 4 which has the smallest number of datapoints and large weights to Gaussians 1 and 5 which had the largest number of datapoints. So, overall our EM algorithm has produced a GMM that accurately models the data.