Statistical Computing in R - Assignment 3

Valentin Kodderitzsch

2023-12-06

Document Set Up

```
setwd("/Users/valentinkodderitzsch/Coding/r-for-stats/r-course/assignment")

# Set seed to student ID as instructed
set.seed(3895157)
```

Exercise 1

Part 1

```
# Install and load libraries
pkgs <- rownames(installed.packages())
if(!"palmerpenguins" %in% pkgs) install.packages("palmerpenguins")
library(palmerpenguins)
library(dplyr)
library(ggplot2)

# Inspect penguins dataset
?penguins

# Load the dataset into the R environment
data("penguins")

# Convert to dataframe
df = data.frame(penguins)
head(df)
```

```
##   species    island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
## 1  Adelie Torgersen           39.1          18.7               181        3750
## 2  Adelie Torgersen           39.5          17.4               186        3800
## 3  Adelie Torgersen           40.3          18.0               195        3250
## 4  Adelie Torgersen             NA            NA                NA          NA
## 5  Adelie Torgersen           36.7          19.3               193        3450
## 6  Adelie Torgersen           39.3          20.6               190        3650
##      sex year
## 1   male 2007
```

```
## 2 female 2007
## 3 female 2007
## 4   <NA> 2007
## 5 female 2007
## 6   male 2007
```

The penguins dataset has 344 observations with 8 variables. It is formatted as a `tibble` and contains variables denoting physical attributes about different penguins living on different islands.

Variable Name	Type	Description
species	Factor	The type of penguin
Island	Factor	The island on which the penguin lives
bill_length_mm	Numerical	Penguin's bill length in millimeter
bill_depth_mm	Numerical	Penguin's bill depth in millimeter
flipper_length_mm	Numerical	Penguin's flipper length in millimeter
body_mass_g	Numerical	Penguin's weight in grams
sex	Factor	Male or female
year	Numerical	Year of Study

Part 2

The question asks for the joint frequency distribution of penguins by species and island. This can be obtained by grouping by species and then by islands.

```
# Get the unique values in the species and island columns
unique(df$species)
```

```
## [1] Adelie    Gentoo    Chinstrap
## Levels: Adelie Chinstrap Gentoo
```

```
unique(df$island)
```

```
## [1] Torgersen Biscoe    Dream
## Levels: Biscoe Dream Torgersen
```

```
# Display the joint frequency table
table(df$species, df$island)
```

```
##
##             Biscoe Dream Torgersen
##   Adelie        44    56        52
##   Chinstrap      0    68         0
##   Gentoo       124     0         0
```

```
# Total number of penguins
sum(count(df, species, island)$n)
```

```
## [1] 344
```

There exist 3 different penguin species (Adelie, Chinstrap and Gentoo) and 3 different islands (Biscoe, Dream and Torgersen).

Adelie penguins live on all 3 islands. Chinstrap penguins only live on the Dream island and Gentoo penguins live exclusively on the Biscoe island. In total there are 344 penguins in this dataset, i.e. each row represents one penguin.

Part 3

The given null hypothesis in the exercise is that $H_0 : \mu_C \geq \mu_G$, where μ_G is the expected value of `bill_length_mm` for Gentoo penguins, and μ_C is the expected value of `bill_length_mm` for Chinstrap penguins. Thus, the alternative hypothesis must be $H_A : \mu_C < \mu_G$, which can be tested using the `less` alternative of the `t.test` function. The exercise assumes a normal distribution for `bill_length_mm`.

```
# Get the bill length of the Gentoo and the Chinstrap penguin
c_bill = filter(df, species == 'Chinstrap') |> select(bill_length_mm)
g_bill = filter(df, species == 'Gentoo') |> select(bill_length_mm)


# H0: mu_c >= mu_g with Ha: mu_c < mu_g
alpha = 0.05
test_less = t.test(x = c_bill, y = g_bill, alternative = "less", conf.level = 1-alpha)
test_less
```

```
##
##  Welch Two Sample t-test
##
## data:  c_bill and g_bill
## t = 2.706, df = 129.22, p-value = 0.9961
## alternative hypothesis: true difference in means is less than 0
## 95 percent confidence interval:
##      -Inf 2.142598
## sample estimates:
## mean of x mean of y
##  48.83382  47.50488
```

Based of the `p-value = 0.9961` which is greater than the chosen `alpha = 0.05`, we fail to reject the null hypothesis that $\mu_C \geq \mu_G$, i.e we "accept" (even though we technically never accept) the hypothesis that Chinstrap penguins on average have a larger bill length than Gentoo penguins.

If we test for equal means, i.e. $H_0 : \mu_C = \mu_G$, we get a `p-value = 0.0077` which is smaller than `alpha = 0.05`. Therefore, we reject the equal means hypothesis.

Based on both hypothesis tests, we can conclude with more evidence that the first null hypothesis $H_0 : \mu_C \geq \mu_G$ is reasonable, i.e. cannot be rejected.

```
test_equal = t.test(c_bill, g_bill, alternative = "two.sided", conf.level = 1-alpha)
test_equal
```

```
##
##  Welch Two Sample t-test
##
## data:  c_bill and g_bill
```

```
## t = 2.706, df = 129.22, p-value = 0.00773
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.3572698 2.3006212
## sample estimates:
## mean of x mean of y
##  48.83382  47.50488
```

Exercise 2

Part 1

Remove observations with NA values in either `body_mass_g` or `species`. The below code can be re-run to double check that there are no NA values in said columns.

```
# Check for NA in body mass column and return index
if (sum(is.na(df$body_mass_g)) > 0){
  print("Remove the following indexes from body_mass_g")
  which(is.na(df$body_mass_g))}
```

```
## [1] "Remove the following indexes from body_mass_g"
```

```
## [1]   4 272
```

```
# Check for NA in species and return index
if (sum(is.na(df$species)) > 0){
  print("Remove the following indexes from species")
  which(is.na(df$species))}

# Remove rows with NA body mass or NA species
df = filter_at(df, vars(body_mass_g, species), all_vars(!is.na(.)))
```

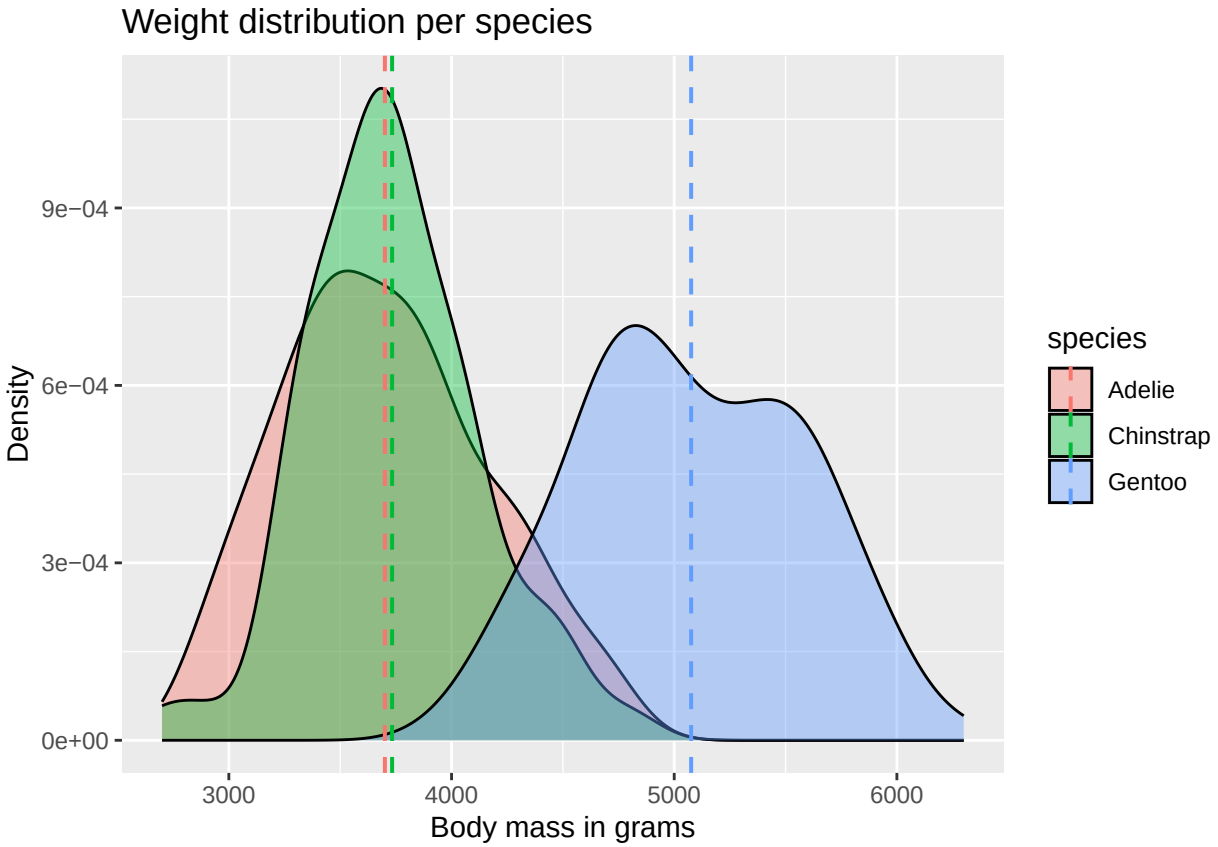
Part 2

Create a density plot that compares the distribution of weight, i.e. `body_mass_g` of the different species.

```
# Calculate the mean for each species
avg_body_mass_g = aggregate(body_mass_g ~ species, data = df, FUN = mean)
avg_body_mass_g
```

```
##     species body_mass_g
## 1    Adelie    3700.662
## 2 Chinstrap    3733.088
## 3    Gentoo    5076.016
```

```
# Overlapping density plot per species, with mean values
# Smoothing parameter has been left at the default of 'adjust = 1'
ggplot(data=df,
       aes(x=body_mass_g, group=species, fill=species)) +
  geom_density(adjust=1, alpha=.4) +
  geom_vline(data = avg_body_mass_g, aes(xintercept = body_mass_g, color = species),
             linetype = "dashed", linewidth = 0.7) +
  labs(title = "Weight distribution per species", x = 'Body mass in grams', y = 'Density')
```



Based on the above density plot we can observe that the mean body mass of Adelie and Chinstrap penguins is very close at around 3700 grams. However, Adelie penguins have a bigger deviation in mass. We can also observe that Gentoo penguins have the largest mean body mass at around 5100 grams.

Part 3

Part 3.1

Convert `body_mass_g` into kilograms and use said column vector (call it `x.mass`) for the rest of the exercises. It is assumed that x (aka `x.mass`) follows a mixture model with 3 normal distributions which can be denoted as follows:

$$f_X(x) = \pi_1 N_1(\mu_1, \sigma_1) + \pi_2 N_2(\mu_2, \sigma_2) + \pi_3 N_3(\mu_3, \sigma_3)$$

This means there exist 3 unknown weights π's, 3 unknown means μ's and and 3 unknown standard deviations σ's.

```
# x is the new body_mass_g vector in kilos
x.mass = df$body_mass_g/1000
```

Part 3.2

Write a function that evaluates the negative log-likelihood of the mixture model. Mathematically speaking, the log-likelihood function is a function of thetas. But in the coding implementation we need to also include

the mixture weights and the clustering probabilities as input arguments as they are assumed to be fixed, i.e they can be thought of as known coefficients in the context of the log-likelihood function. We use the log-likelihood function in the EM algorithm to compute the maximum likelihood estimators for theta.

Reparameterize $\sigma \in [0, \infty)$ to be an unbounded parameter using $\tau = log(\sigma)$, i.e $\tau \in \mathbb{R}$

```
# Log-likelihood function but negated so that it can be minimized using optim().
# Tao is an unbound parameter that can be converted back using sigma = exp(tao)
## @param theta: Vector with parameter values mu1, mu2, mu3, tao1, tao2, tao3
## @param weights: Vector with mixture weights 1 and 2
## @param clt_1.row: Vector with clustering probabilities on
### all observations to be component 1
## @param clt_2.row: Vector with clustering probabilities on
### all observations to be component 2
## @param x: Vector representing the given dataset
neg.loglik = function(theta, weights, clt_1.row, clt_2.row, x) {
  # Extract mu's
  mu1 = theta[1]
  mu2 = theta[2]
  mu3 = theta[3]

  # Extract sigma's by converting from taos
  sigma1 = exp(theta[4])
  sigma2 = exp(theta[5])
  sigma3 = exp(theta[6])

  # Extract mixture weights: \pi_i
  wt_1 = weights[1]
  wt_2 = weights[2]

  # Check that the sum of weights is not greater than 1 (i.e. sum of 1 is allowed)
  if (wt_1+wt_2 > 1) {
    stop("Sum of wt_1 and wt_2 cannot exceed 1")
  }
  wt_3 = 1 - wt_1 - wt_2

  # Check that the sum of probabilistic classification is not greater than 1
  sum = clt_1.row + clt_2.row
  rs = round(sum, 6) # Fix floating-point arithmetic issue
  if ( any(rs > 1) ) {
    stop("Sum of clt_1.row and clt_2.row cannot exceed 1 at any element")
  }
  clt_3.row = 1 - clt_1.row - clt_2.row

  # Density of the mixture model:
  f.x1 = wt_1 * dnorm(x, mu1, sigma1)
  f.x2 = wt_2 * dnorm(x, mu2, sigma2)
  f.x3 = wt_3 * dnorm(x, mu3, sigma3)

  # Negative log-likelihood:
  - sum(clt_1.row * log(f.x1) + clt_2.row * log(f.x2) + clt_3.row * log(f.x3))
}
```

The correctness of the above log-likelihood function has been tested in the Testing section.

Part 4

Implement the Expectation-Maximisation (EM) algorithm and apply it to the vector `x.mass` which represents the body mass in kg. Similarly to the log-likelihood function, I checked the correctness of my implementation of the EM algorithm in the Testing section

First, implement a helper function to initialize the values at iteration 1.

```
# EM init function initializes and return theta, weights clustering probabilities
## @param n.iter: Number of iterations
## @param data: Dataset of the EM algorithm
## @param start_t: Starting values for theta (mu1 to mu3, sig1 to sig3)
## @param start_w: Starting weights w1 and w2
EM_init = function(n.iter = 200, data = x.mass,
                   start_t = c(3.7, 3.73, 5.07, 0.46, 0.38, 0.5),
                   start_w = c(0.3, 0.38)) {
  # Check that the dataset has no NA values
  if (any(is.na(data))) {
    stop("Given dataset in EM_init() contains NA values which is not allowed")
  }
  n = length(data)

  # Estimates for the mixture weights 1 and 2
  wt.hat = matrix(NA, n.iter, 2)

  # Estimates for the clustering probabilities for factor 1 and 2
  clt_1 = clt_2 = matrix(NA, n.iter, n)

  # Theta values are: mu1, mu2, mu3, tao1, tao2, tao3
  theta.hat = matrix(NA, n.iter, 6)

  # Initialize values based on function input
  wt.hat[1, 1] = start_w[1]
  wt.hat[1, 2] = start_w[2]

  # Expectation of the clustering probabilities have to be mixture weight
  ## Set range very small so that the sum of weights is <= 1
  clt_1[1, ] = runif(n, wt.hat[1, 1] - 0.2, wt.hat[1, 1] + 0.2)
  clt_2[1, ] = runif(n, wt.hat[1, 2] - 0.02, wt.hat[1, 2] + 0.02)

  # Reparameterised optimisation (i.e. minimisation of neg.loglik)
  ## Starting values are mu1, mu2, mu3, tao1, tao2, tao3
  ## tao = log(sigma) => sigam = exp(tao)
  opt_res = optim(start_t,
    function(theta) neg.loglik(theta, wt.hat[1,], clt_1[1, ], clt_2[1, ], data))

  # Get the optim parameters
  theta.hat[1, ] = opt_res$par

  output = list('wt.hat' = wt.hat,
                'clt_1' = clt_1,
                'clt_2' = clt_2,
                'theta.hat' = theta.hat)
```

```
  return(output)
}
```

Actually implement the EM algorithm.

```
# EM algorithm return the estimates for theta and its log-likelihood, weights,
## clustering probabilities, and the weights over
## all iteration to check for convergence
## @param n.iter: Number of iterations
## @param data: Dataset for the EM algorithm
## @param start_t: Starting values for theta (mu1, mu2, mu3, tao1, tao2, tao3)
## @param start_w: Starting weights w1 and w2
EM_algo = function(n.iter = 200, data = x.mass,
                           start_t = c(3.7, 3.73, 5.07, 0.46, 0.38, 0.5),
                           start_w = c(0.3, 0.38)) {

  # Create initials values for iteration t = 1
  init_values = EM_init(n.iter, data, start_t, start_w)

  # Extract values
  wt.hat = init_values[["wt.hat"]]
  clt_1 = init_values[["clt_1"]]
  clt_2 = init_values[["clt_2"]]
  theta.hat = init_values[["theta.hat"]]

  # Iteration 2 until end
  for (t in 2:n.iter) {
    # Extract tmp theta and weights from last iteration
    tmp.mu1 = theta.hat[t-1, 1]
    tmp.mu2 = theta.hat[t-1, 2]
    tmp.mu3 = theta.hat[t-1, 3]

    # Convert from taos to sigmas
    tmp.sig1 = exp(theta.hat[t-1, 4])
    tmp.sig2 = exp(theta.hat[t-1, 5])
    tmp.sig3 = exp(theta.hat[t-1, 6])

    tmp.wt_1 = wt.hat[t-1, 1]
    tmp.wt_2 = wt.hat[t-1, 2]

    # E STEP: Updates the clustering probabilities
    p.temp = cbind(
      tmp.wt_1 * dnorm(data, tmp.mu1, tmp.sig1),
      tmp.wt_2 * dnorm(data, tmp.mu2, tmp.sig2),
      (1 - tmp.wt_1 - tmp.wt_2) * dnorm(data, tmp.mu3, tmp.sig3)
    )

    # Clustering probabilities for factor 1
    clt_1[t, ] = p.temp[ , 1]/rowSums(p.temp)

    ## Clustering probabilities for factor 2
    clt_2[t, ] = p.temp[ , 2]/rowSums(p.temp)
```

```
    # M STEP: Updates mixture weights and theta values
    wt_1.hat = mean(clt_1[t, ])
    wt_2.hat = mean(clt_2[t, ])
    wt.hat[t, ] = c(wt_1.hat, wt_2.hat)

    # Reparameterised optimisation (i.e. minimisation of neg.loglik)
    ## Starting values are mu1, mu2, mu3, tao1, tao2, tao3
    ## tao = log(sigma) => sigam = exp(tao)
     opt_result  = optim(theta.hat[t-1, ], function(theta)
      neg.loglik(theta, wt.hat[t, ], clt_1[t, ], clt_2[t, ], data))

     # print(paste("Inside for loop at t =", t))
     # print(opt_result)
     # print(wt.hat[t, ])
     theta.hat[t, ] = opt_result$par
  } # end for loop

  # Log-likelihood value at the last iteration, i.e minus minus is plus
  log.last = - neg.loglik(theta.hat[n.iter, ],
                          wt.hat[n.iter, ],
                          clt_1[n.iter, ],
                          clt_2[n.iter, ], data)

  # Compute wt_3 at final iteration n.iter
  wt_3 = 1 - sum(wt.hat[n.iter, ])

  # Compute clustering probabilities at final iteration n.iter
  clt_total_n.iter = cbind(clt_1[t, ],
                           clt_2[t, ],
                           1 - clt_1[t, ] - clt_2[t, ])

  # Compute the weight iteration for factor 3
  wt_3_iterations = 1 - rowSums(wt.hat)

  # Convert taos to sigmas
  sig_all = exp(theta.hat[n.iter, 4:6])

  output = list('loglik' = log.last,
                'weights' = c(wt.hat[n.iter, ], wt_3 ),
                'theta' = c(theta.hat[n.iter, 1:3], sig_all),
                'weight_iterations' = cbind(wt.hat, wt_3_iterations),
                'clusters' = clt_total_n.iter)

  return(output)
}
```

Part 5

Run the algorithm with different starting values.

To do so, isolate the body mass column to calculate new starting values for theta.

```r
# Isolate the body mass per species and convert to kg
mass_c = as.numeric(filter(df, species == 'Chinstrap') |>
  select(body_mass_g) |> apply(2, function(x) x/1000))

mass_g = as.numeric(filter(df, species == 'Gentoo') |>
  select(body_mass_g) |> apply(2, function(x) x/1000))

mass_a = as.numeric(filter(df, species == 'Adelie') |>
  select(body_mass_g) |> apply(2, function(x) x/1000))

# Set factor 1 = Adelie; factor 2 = Gentoo; factor 3 = Chinstrap
start_t1 = c(mean(mass_a),mean(mass_g), mean(mass_c),
          log(sd(mass_a)), log(sd(mass_g)), log(sd(mass_c)) )
```

Run the EM algorithm. In the first run assume that the starting theta values correspond to the mean body mass per species. Assume that the starting weight `start_w1` is about $\frac{1}{k}$ where $k = 3$. This is a common heuristic for the EM algorithm.

My conclusion (as shown in the Testing section) is that weight 1 corresponds to Adelie, weight 2 to Gentoo and weight 3 to Chinstrap penguins. When examining the given dataset, it becomes clear that the first 150 observations are Adelie penguins only. The following 124 observations are Gentoo penguins only and the final 68 observations are Chinstrap penguins only. Since the observations are clearly ordered by species, the corresponding weights of the mixture model will converge to be the same order (i.e. encoding) as in the dataset.

For each run I set the same seed value so that the runs stay comparable. As I have learned during my testing phase, different seed values and re-runs affect the final result because of the random draws from the uniform distribution in the `EM_int` function.

```r
# Run 1
# Set seed so that runs are comparable
set.seed(3895157)

# Starting values for weights
start_w1 = c(0.3, 0.36)

# Number of iterations
n.iter = 500

# Run the EM algorithm with the above starting values for theta
## and starting weight around 1/3
final_results1 = EM_algo(n.iter = n.iter, data = x.mass,
                         start_t = start_t1, start_w = start_w1)
final_results1[1:3]
```

```
## $loglik
## [1] -464.1998
##
## $weights
## [1] 0.5371473 0.3275975 0.1352552
##
## $theta
## [1] 3.5955807 4.6294628 5.5729352 0.3563990 0.3944398 0.2789946
```

For the second run take the overall mean and standard deviation of the body mass column as starting theta value. Add a small margin to said mean and standard deviation to improve convergence. Choose the same starting weights as in run 1 so that the runs stay comparable.

```
# Run 2
# Set seed so that runs are comparable
set.seed(3895157)

# Starting values for theta
start_t2 = c(mean(x.mass), mean(x.mass) + 0.1, mean(x.mass) - 0.1,
           log(sd(x.mass)), log(sd(x.mass)) + 0.1, log(sd(x.mass)) - 0.1)

# Run the EM algorithm with the above starting values and starting weights around 1/3
final_results2 = EM_algo(n.iter = n.iter, data = x.mass,
                         start_t = start_t2,
                         start_w = start_w1)
final_results2[1:3]
```

```
## $loglik
## [1] -464.2025
##
## $weights
## [1] 0.5371541 0.3276206 0.1352253
##
## $theta
## [1] 3.5956690 4.6295153 5.5729737 0.3564177 0.3944190 0.2789725
```

For the final run set the starting weights to be the same as the distribution of species in the dataset, so about $40 : 20 : 40$. Choose the starting theta values from run 2 as they produced a more favorable result compared to using starting values for theta from run 1.

```
# Run 3
# Set seed so that runs are comparable
set.seed(3895157)

# Set the starting weights to 40:20:40. The last weight is calculated internally.
start_w2 = c(0.4, 0.2)

# Run the EM algorithm with the above starting values form
## run 2 and starting weights like in the dataset
final_results3 = EM_algo(n.iter = n.iter, data = x.mass,
                         start_t = start_t2,
                         start_w = start_w2)
final_results3[1:3]
```

```
## $loglik
## [1] -466.2094
##
## $weights
## [1] 0.5323174 0.1328327 0.3348499
##
## $theta
## [1] 3.5917948 5.5772113 4.6257993 0.3546583 0.2774565 0.4036432
```

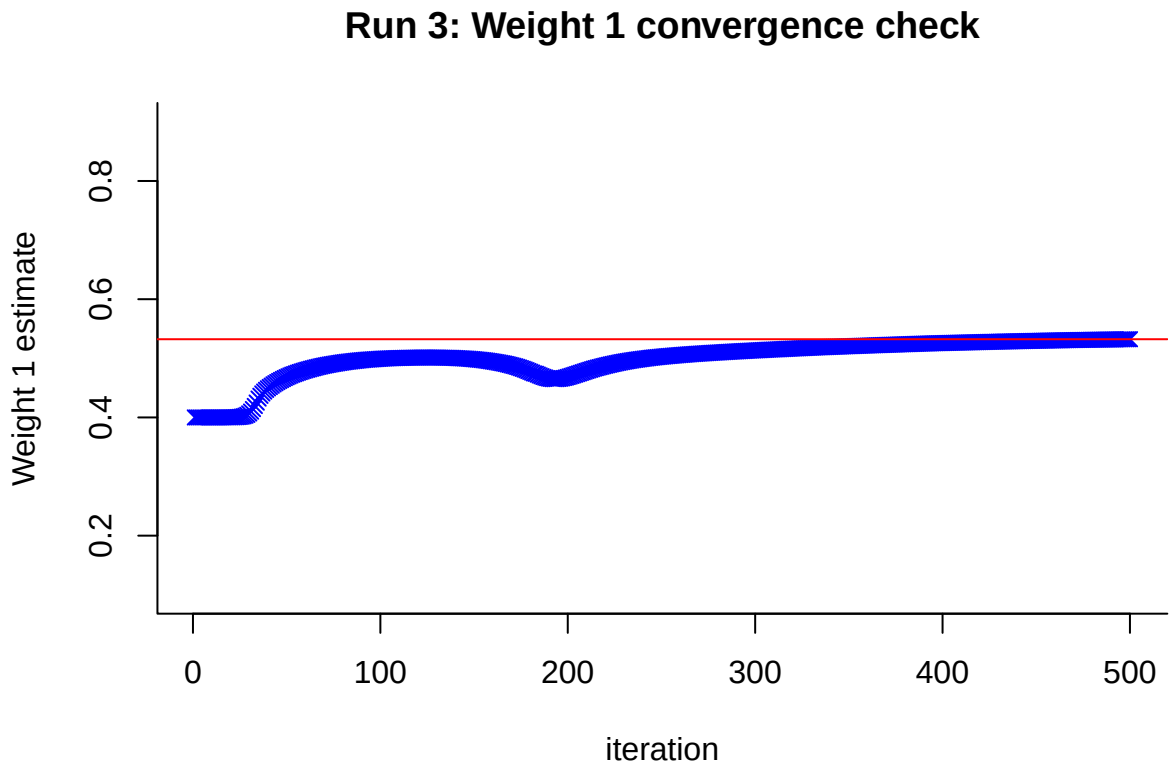
Run 1 should be picked as it has the highest log-likelihood value. However, run 3 produces more favorable results in terms of the order of the theta values. In run 3, $\mu_2 = 5.57$ which is very close to the observed μ of Gentoo penguins. This ordering of the mixutre weights makes the most sense as Gentoo penguins also appear as the second species in the dataset.

Thus, I will choose run 3 for the upcoming sections (followed by run 1 for the sake of curiosity).

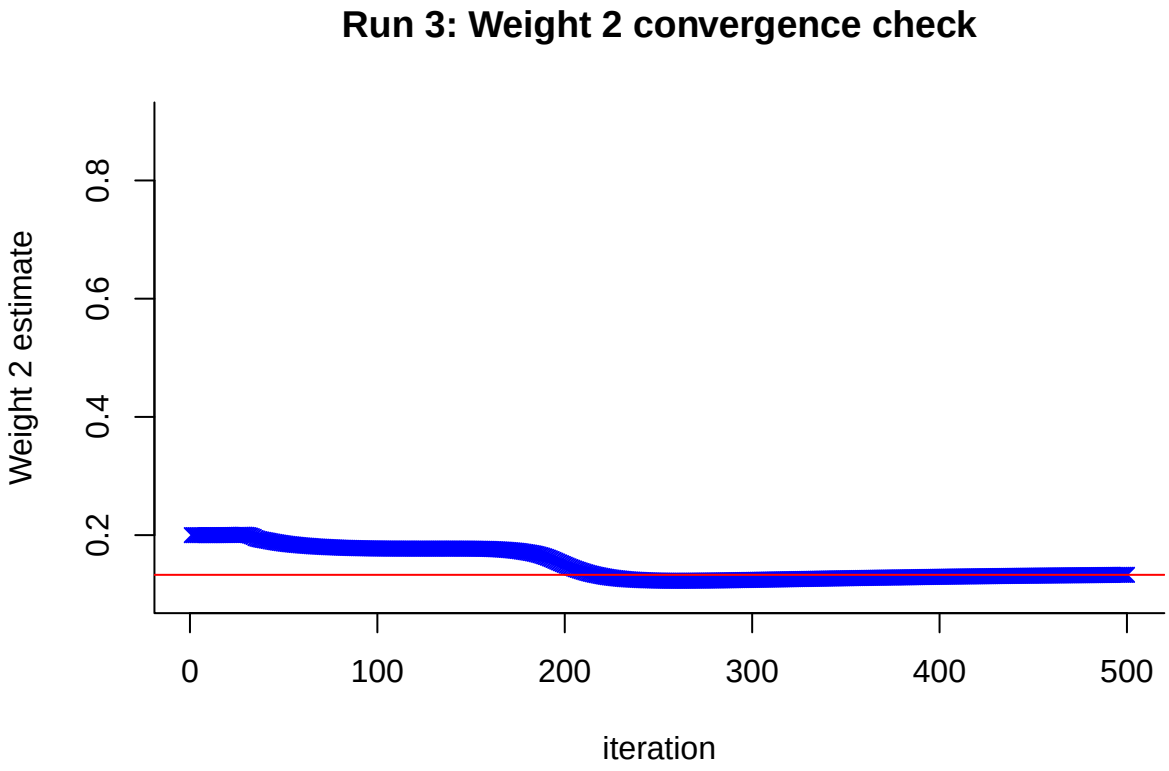
Part 6

Check for convergence by plotting the weights of run 3.

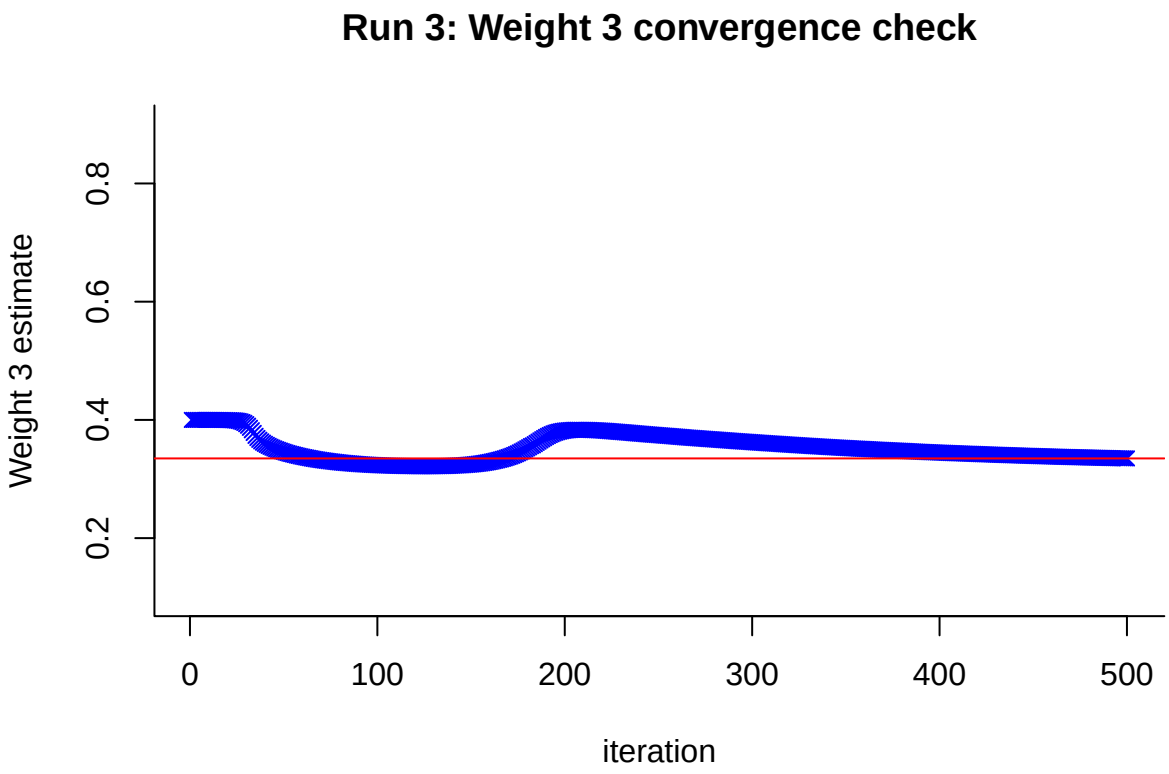
```
#Convergence weight 1
par(bty = 'l')
plot(final_results3[["weight_iterations"]][, 1],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 1 estimate',
     ylim = c(0.1, 0.9), main = 'Run 3: Weight 1 convergence check')
abline(h = final_results3[["weight_iterations"]][n.iter, 1], col = 'red')
```




```
# Convergence weight 2
par(bty = 'l')
plot(final_results3[["weight_iterations"]][, 2],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 2 estimate',
     ylim = c(0.1, 0.9), main = 'Run 3: Weight 2 convergence check')
abline(h = final_results3[["weight_iterations"]][n.iter, 2], col = 'red')
```



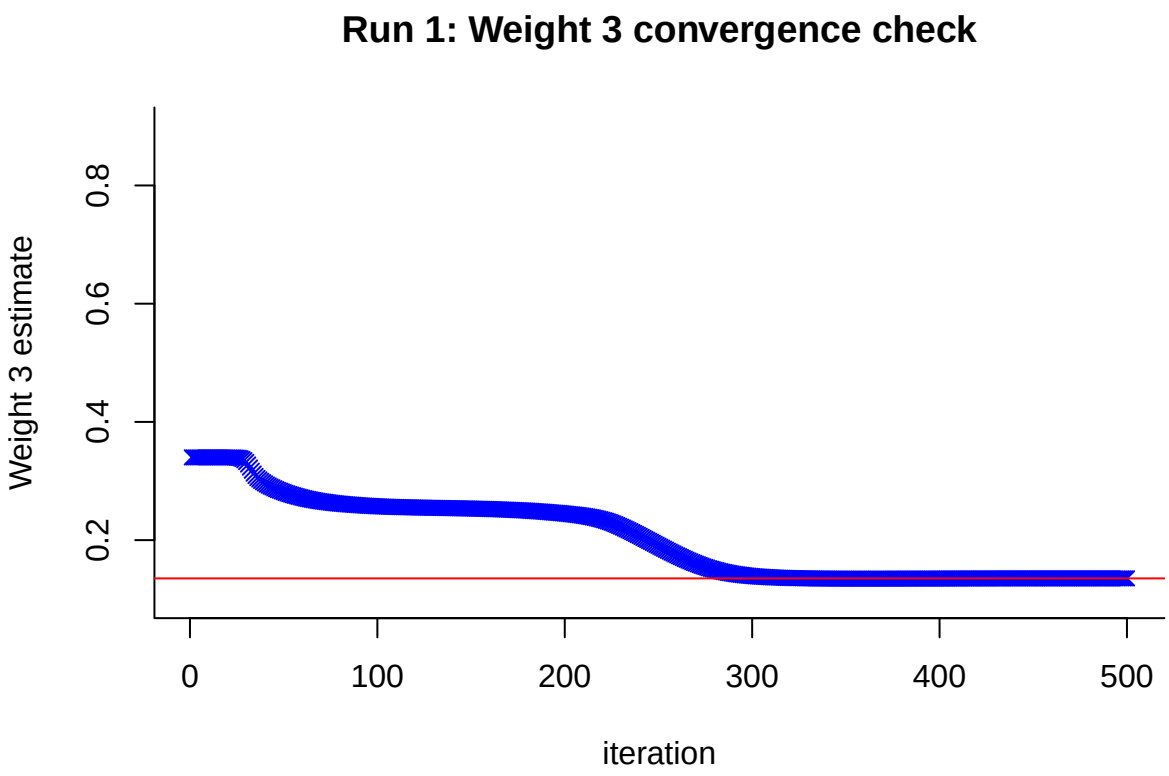

```
# Convergence weight 3
par(bty = 'l')
plot(final_results3[["weight_iterations"]][, 3],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 3 estimate',
     ylim = c(0.1, 0.9), main = 'Run 3: Weight 3 convergence check')
abline(h = final_results3[["weight_iterations"]][n.iter, 3], col = 'red')
```



Based on the above plots we can conclude that the weights have converged for run 3.

For the sake of keeping to a reasonable page limit, only check for convergence of one weight from run 1.

```
# Convergence weight 3, run 1
par(bty = 'l')
plot(final_results1[["weight_iterations"]][, 3],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 3 estimate',
     ylim = c(0.1, 0.9), main = 'Run 1: Weight 3 convergence check')
abline(h = final_results1[["weight_iterations"]][n.iter, 3], col = 'red')
```



Run 1 also seems to have converged. I also manually checked that the other two weights converged as well but omitted said plots from the document to save space.

Part 7

Compare the estimates with the observed mean weight and standard deviation per species. As mentioned in part 5, factor 1 corresponds to Adelie, factor 2 to Gentoo and factor 3 to Chinstrap penguins, i.e AGC ordering.

```
# Create a dataframe
comp = data.frame(estimate = final_results3[['theta']],
                  observed = c(mean(mass_a), mean(mass_g), mean(mass_c),
                               sd(mass_a), sd(mass_g), sd(mass_c)))

# Store estimates in variables
mu_A = final_results3[['theta']][1]
mu_G = final_results3[['theta']][2]
mu_C = final_results3[['theta']][3]

sig_A = final_results3[['theta']][4]
sig_G = final_results3[['theta']][5]
sig_C = final_results3[['theta']][6]

# Name the indexes
rownames(comp) = c("mu_A", 'mu_G', 'mu_C', "sig_A", "sig_G", "sig_C")
```

```r
print(comp)
```

```
##        estimate  observed
## mu_A  3.5917948 3.7006623
## mu_G  5.5772113 5.0760163
## mu_C  4.6257993 3.7330882
## sig_A 0.3546583 0.4585661
## sig_G 0.2774565 0.5041162
## sig_C 0.4036432 0.3843351
```

The mean weight estimates for Adelie and Gentoo close. The Chinstrap mean weight estimate is off. The standard deviation estimates for Adelie and Gentoo are somewhat off, however, the standard deviation estimate for Chinstrap penguins is quite accurate.

Part 8

```r
# Mixture weight estimates
final_results3[['weights']]
```

```
## [1] 0.5323174 0.1328327 0.3348499
```

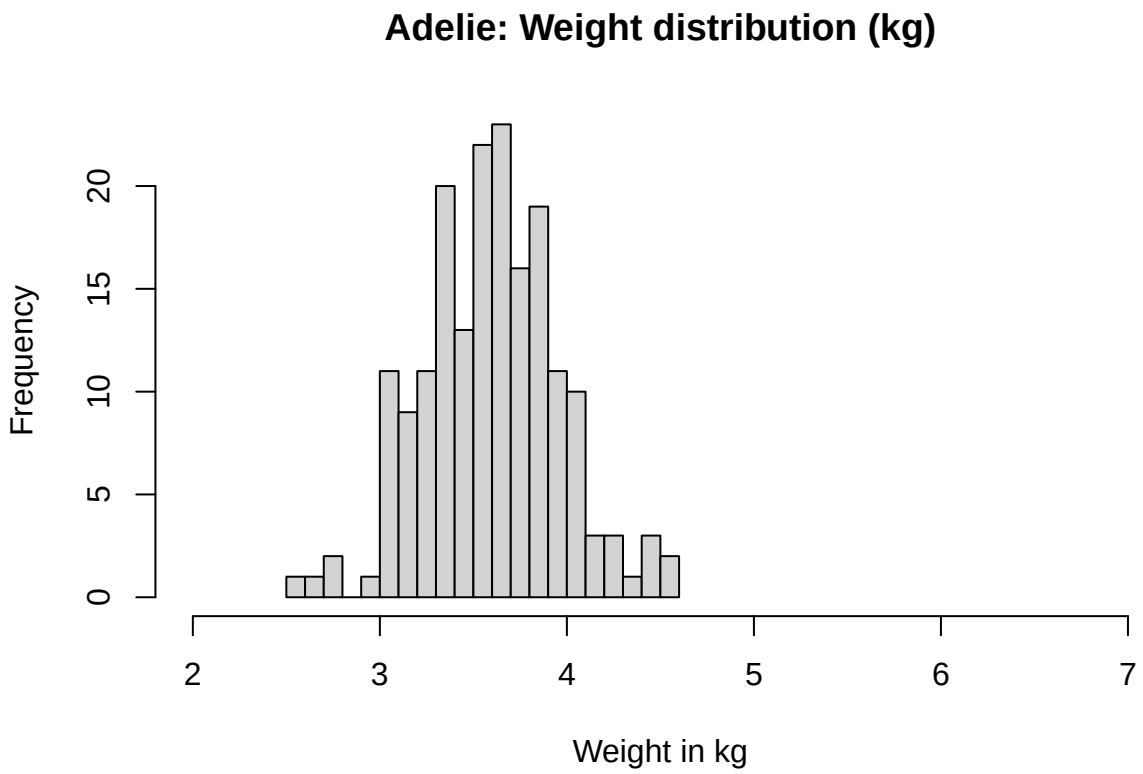
```r
# Store estimates in variables
w_A = final_results3[['weights']][1]
w_G = final_results3[['weights']][2]
w_C = final_results3[['weights']][3]
```

54% of all observations are estimated to be Adelie penguins. 14% are Gentoo and 32% are Chinstrap penguins. This is roughly equivalent to the $40:20:40$ AGC distribution found in the dataset.

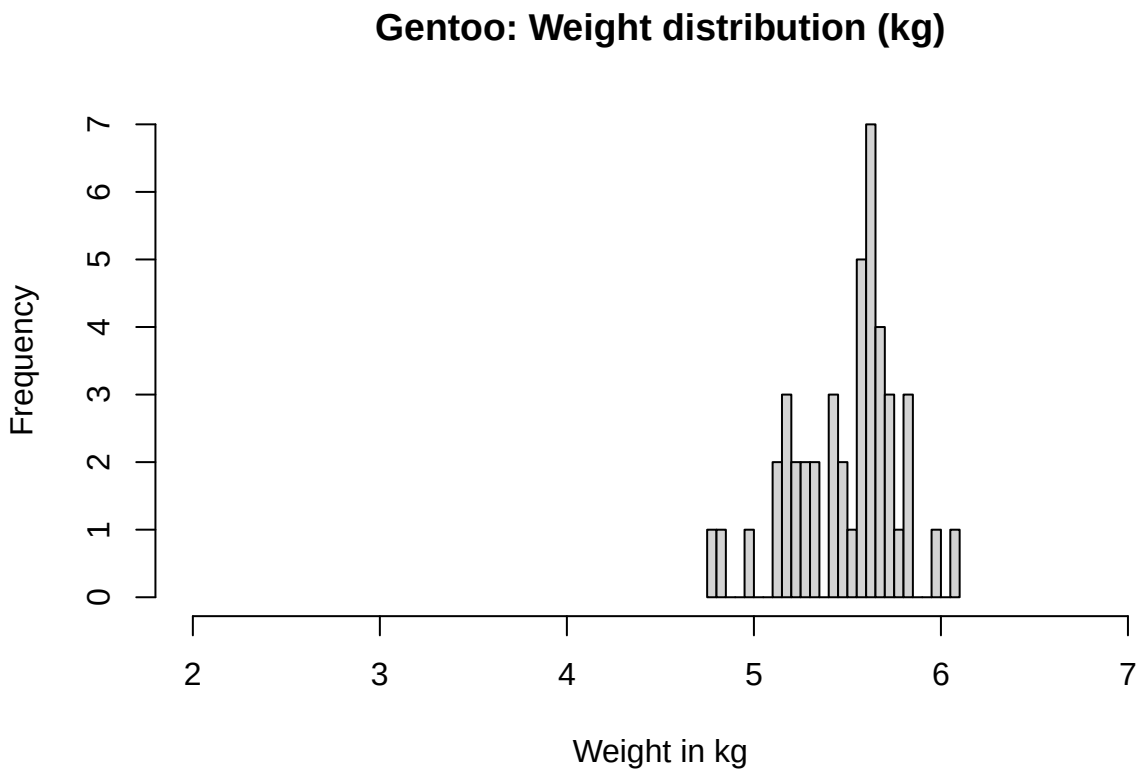
To display the distribution of factor 1, 2, and 3, draw each factor according to the number of penguins classified as factor 1, 2, and 3. Then draw said amount of number from the assumed `rnorm` where μ and σ are the estimated parameters per factor.

Display all histograms on the same x-axis bounds so that they are more comparable.

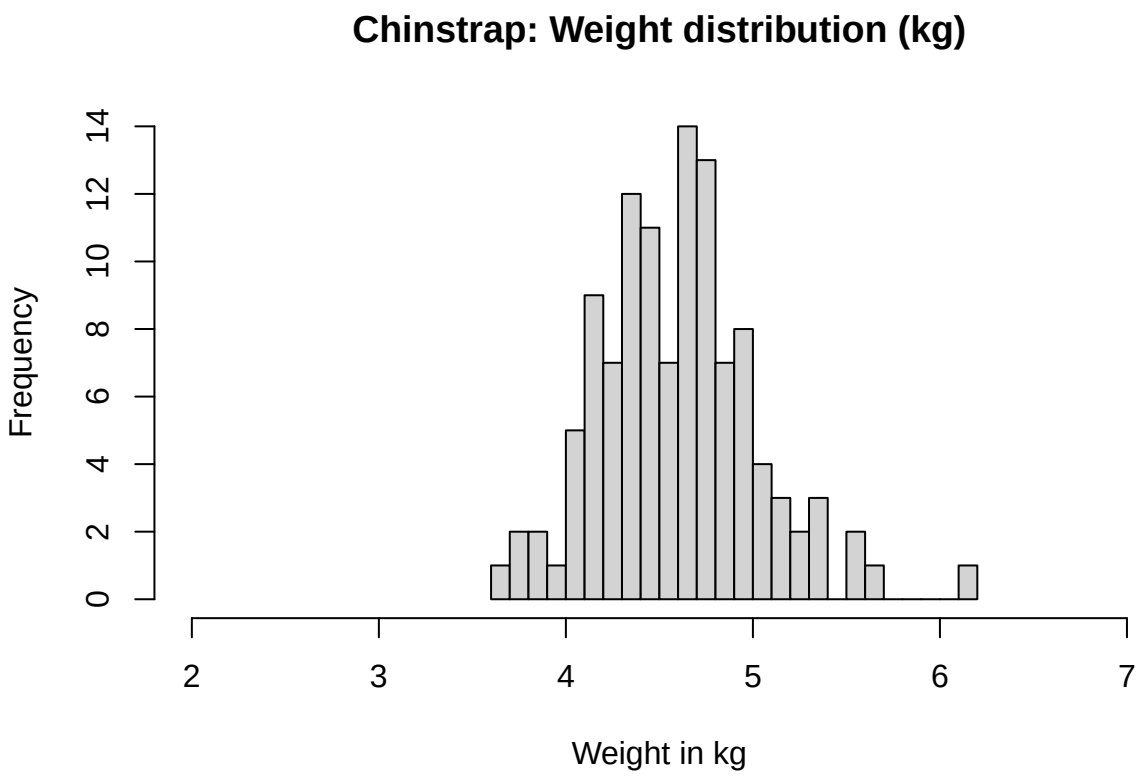
```r
# Length of final dataframe
n = length(x.mass)

# Draw data according to the number of classified penguines per species
# with their estimated mean and sd
dist_A = rnorm(round(w_A*n, 0), mu_A, sig_A)
dist_G = rnorm(round(w_G*n, 0), mu_G, sig_G)
dist_C = rnorm(round(w_C*n, 0), mu_C, sig_C)

# Display the weight distribution Adelie penguines
hist(dist_A, 20,
     main = "Adelie: Weight distribution (kg)",
     xlab = "Weight in kg", xlim = c(2, 7))
```




```
# Gentoo
hist(dist_G, 20,
     main = "Gentoo: Weight distribution (kg)",
     xlab = "Weight in kg", xlim = c(2, 7))
```




```r
# Chinstrap
hist(dist_C, 20,
     main = "Chinstrap: Weight distribution (kg)",
     xlab = "Weight in kg", xlim = c(2, 7))
```



If we were to overlay the 3 histograms they would look similar in shape to the density plot from part 2. The main difference would be that Adelie and Chinstrap penguins do not have the same mean value.

Part 9

Create a variable called `predictedComponent` to represent the weight the classification per observation. For the sake of comparing run 3 with 1, I chose to name the variables `predComp_r3` and `predComp_r1`.

```
# Run 3
# Encode columns as col1 = Ad, col2 = Gent, col3 = Chin
# Classify using the component with the highest probability from run 3 instead
## of using a fixed cut off value
predComp_r3 = apply(final_results3[['clusters']], 1, function(x) which.max(x) )
print(predComp_r3)
```

```
##   [1] 1 1 1 1 1 1 3 1 3 1 1 1 1 3 1 1 3 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 3 1 1 1 3 1 3 1 1 1 1 1 3 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 3 1 1 1 1 1
##  [75] 3 1 1 1 1 1 3 1 3 1 1 1 1 1 1 1 3 1 3 1 3 1 3 1 1 1 3 1 3 1 1 1 1 1 3 1 3
## [112] 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 3 1 1 1 1 1 3 1 1 1 1 1 1 3 1 1
## [149] 1 1 1 3 2 3 2 2 3 3 3 3 3 3 2 3 2 3 2 1 2 3 2 2 3 3 3 3 3 1 2 3 2 3 3 3 2
## [186] 3 2 3 3 3 2 1 2 3 3 2 3 3 2 3 2 3 2 3 3 3 3 3 3 3 2 3 2 3 2 3 2 3 2 3 2 3
## [223] 3 3 3 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 3 3 3 3 3 3 2 3 2 3 2 3 2
## [260] 3 2 3 2 3 2 3 2 3 2 3 3 2 3 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1
## [297] 1 1 1 1 1 1 1 3 1 3 1 1 1 1 1 3 1 3 1 1 1 1 1 3 1 3 1 1 1 1 1 1 1 1 1 1
## [334] 1 1 1 1 1 1 1 1 1
```

```
# Run 1
# Encode columns as col1 = Ad, col2 = Gent, col3 = Chin
# Classify using the component with the highest probability from run 1 instead
## of using a fixed cut off value
predComp_r1 = apply(final_results1[['clusters']], 1, function(x) which.max(x) )
print(predComp_r1)
```

```
##   [1] 1 1 1 1 1 1 2 1 2 1 1 1 1 2 1 1 2 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 2 1 1 1 2 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 2 1 1 1 1 1
##  [75] 2 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 2 1 2 1 2 1 2 1 1 1 2 1 2 1 1 1 1 1 2 1 2
## [112] 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 2 1 1
## [149] 1 1 1 2 3 2 3 3 2 2 2 2 2 2 3 2 3 2 3 1 3 2 3 3 2 2 2 2 2 1 3 2 3 3 2 2 3
## [186] 2 3 2 3 2 3 1 3 2 2 3 2 2 3 2 3 2 3 2 2 2 2 2 2 2 3 2 3 2 3 2 3 2 3 2 3 2
## [223] 2 2 2 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 2 2 2 2 3 2 3 2 3 2 3 2 3
## [260] 2 3 2 3 2 3 2 3 2 3 2 2 3 2 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
## [297] 1 1 1 1 1 1 1 2 1 2 1 1 1 1 1 2 1 2 1 1 1 1 1 2 1 2 1 1 1 1 1 1 1 1 1 1 1
## [334] 1 1 1 1 1 1 1 1 1
```

Part 10 with 11

Compute the joint frequency table of `sex` with `predictedComponent`.

First use run 3.

```
# Run 3
# Joint PDF of Sex vs predicted component
sex = df$sex
table(sex, predComp_r3)
```

```
##         predComp_r3
## sex        1   2   3
##   female 109   0  56
##   male    74  46  48
```

Then compare with run 1.

```
# Run 1
# Joint PDF of Sex vs predicted component
table(sex, predComp_r1)
```

```
##         predComp_r1
## sex        1   2   3
##   female 109  56   0
##   male    74  45  49
```

Encode the actual component with Adelie = 1, Gentoo = 2 and Chinstrap = 3.

```
# Encoding the actual component
actualComponent = ifelse(df$species == 'Adelie', 1, 0) +
  ifelse(df$species == 'Gentoo', 2, 0) +
  ifelse(df$species == 'Chinstrap', 3, 0)

print(actualComponent)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [186] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [223] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [260] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [297] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [334] 3 3 3 3 3 3 3 3 3
```

Now compare both run 1 and 3 with the sex vs actual components joint PDF from the dataset.

```
table(sex, actualComponent)
```

```
##         actualComponent
## sex       1  2  3
##   female 73 58 34
##   male   73 61 34
```

Considering run 3 only, if a penguin has `predictedComponent` $= 1$, then I guess it is a female as $\frac{109}{109+74} > 50\%$ of all penguins in PC1 are female. Using the same logic, guess male for PC2 and female for PC3.

Based on the true sex vs true component distribution the guesses for both PC1 and PC3 would be correct as they have an even split between male and female penguins. The guess for PC2 would be false.

We can also notice that there is not a big difference between run 3 and 1 apart from the fact the PC2 would be guessed female and PC3 would be guessed male for run 1. However, the actual numerical distribution per predicted component is almost identical.

Part 10 with 12

Compute the joint frequency table of `species` with `predictedComponent`. The below table is correct, however, the order of the species is now alphabetic and not AGC as in the dataset because of the `table` function. Thus, we need to use the `actualComponent` variable which has the correct encoding order.

```
# Run 3
# Joint PDF of Species vs predicted component
species = df$species
table(species, predComp_r1)
```

```
##            predComp_r1
## species       1   2   3
##   Adelie    124  27   0
##   Chinstrap  61   7   0
##   Gentoo      3  71  49
```

To compute the classification accuracy for species we need to compare actual vs predicted components. The species order as read from the dataset is 1) Adelie, 2) Gentoo, 3) Chinstrap. Based on the observed sample mean we know that Adelie and Chinstrap penguins are quite similar in weight. Therefore, any misclassifications between Adelie and Chinstrap penguins would not be surprising.

Compute the classification accuracy for run 3.

```r
# Run 3
# Classification accuracy table
accuracy = table(actualComponent, predComp_r3)
accuracy
```

```
##                 predComp_r3
## actualComponent   1   2   3
##               1 124   0  27
##               2   3  46  74
##               3  61   0   7
```

```r
# Numeric classification accuracy
overall_3 = sum(diag(accuracy))/n

print(paste("Overall classification accuracy RUN 3:", 100*overall_3, "%"))
```

```
## [1] "Overall classification accuracy RUN 3: 51.7543859649123 %"
```

```r
print(paste("Classification accuracy for Adelie class:", 100* 124/(124+27), "%") )
```

```
## [1] "Classification accuracy for Adelie class: 82.1192052980132 %"
```

```r
print(paste("Classification accuracy for Gentoo class:", 100* 46/(3+46+74), "%") )
```

```
## [1] "Classification accuracy for Gentoo class: 37.3983739837398 %"
```

```r
print(paste("Classification accuracy for Chinstrap class:", 100* 7/(61+7), "%") )
```

```
## [1] "Classification accuracy for Chinstrap class: 10.2941176470588 %"
```

Compute the classification accuracy for run 1. This is to check if a run with higher log-likelihood values would make more sense than considering the ordering of the theta values.

```r
# Run 1
# Classification accuracy table
accuracy = table(actualComponent, predComp_r1)
accuracy
```

```
##                 predComp_r1
## actualComponent   1   2   3
##               1 124  27   0
##               2   3  71  49
##               3  61   7   0
```

```r
# Numeric classification accuracy
overall_1 = sum(diag(accuracy))/n

print(paste("Overall classification accuracy RUN 1:", 100*overall_1, "%"))
```

```
## [1] "Overall classification accuracy RUN 1: 57.0175438596491 %"
```

```r
print(paste("Classification accuracy for Adelie class:", 100* 124/(124+27), "%") )
```

```
## [1] "Classification accuracy for Adelie class: 82.1192052980132 %"
```

```r
print(paste("Classification accuracy for Gentoo class:", 100* 71/(71+3+49), "%") )
```

```
## [1] "Classification accuracy for Gentoo class: 57.7235772357724 %"
```

```r
print(paste("Classification accuracy for Chinstrap class:", 100* 0/(61+7), "%") )
```

```
## [1] "Classification accuracy for Chinstrap class: 0 %"
```

The overall classification accuracy is 52% for run 3 and 58% for run 1. At first glance, it appears that the classifier that favors a higher log-likelihood instead of correct factor ordering performs better.

However, run 3 classifies 0% of chinstrap penguins correctly. This could be a result of how `predComp_r1` was encoded using the AGC ordering instead of updating the factor ordering to match the order of the EM output for theta.

I believe that it might make more sense to link the theta values of the EM output to the closest observed mean per species from the dataset as factor encoding. Fixing the encoding for run 1 will be left as a task in the future work section due to time constraints. Thus, the classification accuracy of run 1 might be more luck than actual classification prowess.

Regardless, both runs 1 and 3 are pretty bad at classifying the Chinstrap class which was expected as it is almost identical in distribution to the Adelie class. However, **both classifiers perform better than the null model**, which is good news. For this classification problem, a null model has a classification accuracy of $\frac{1}{k}$ with $k = 3$

Testing

The following section tests/shows that my coding implementation works correctly for well localized, simulated data.

Any maximum likelihood estimator (MLE) first assumes a distribution from which the data is sampled, and then computes an estimate for the assumed distribution parameters, i.e the estimands. When given an arbitrary dataset without much context, it is hard to tell what the true underlying distribution is. Consequently, it is difficult to test the correctness of your coding implementation as 1) your estimates might be off because you assumed the wrong distribution to begin with for the give dataset and/or 2) the implementation for the estimator is incorrect.

Therefore, to test for 2) the distribution assumptions made for the MLE must be made for the simulated dataset. In other words, the density function used in the MLE and the simulated dataset have the same distribution.

Log-likelihood function

Check that the negative log-likelihood function is correct using the parametric bootstrap. If my log-likelihood function is correct, then the 6 estimates for the normal distribution parameters should be close to the true parameters (which are known since we simulated the distribution).

Create a simulated dataset where the 3 normal distributions are clearly separated. The idea is that the MLE should be easily discovered.

```
# Create parameters for the simulated data (s.data)
s.n = 500
s.w1 = s.w3 = 0.4
s.w2 = 0.2
s.mu1 = 3
s.mu2 = 10
s.mu3 = 18
s.sig1 = 1.2
s.sig2 = 1.4
s.sig3 = 1.6

# Just like the given dataset, the first 200 entries are factor 1 only,
# the following 100 observations are factor 2 only, same logic for factor 3
s.groups = c(rep(1, 200), rep(2, 100), rep(3, 200))
table(s.groups)
```

```
## s.groups
##   1   2   3
## 200 100 200
```

```
# Create the simulated dataset using the mixture model
s.x = rep(NA, s.n)
s.x[s.groups == 1] = rnorm(sum(s.groups == 1), s.mu1, s.sig1)
s.x[s.groups == 2] = rnorm(sum(s.groups == 2), s.mu2, s.sig2)
s.x[s.groups == 3] = rnorm(sum(s.groups == 3), s.mu3, s.sig3)

# Calculate the coefficient of variation
cv = sd(s.x)/mean(s.x) * 100
```

```r
# Print information about the variability of the data => Needed to put MSE in context
print(paste('mean', mean(s.x)))
```

```
## [1] "mean 10.4472307328154"
```

```r
print(paste('sd', sd(s.x)))
```

```
## [1] "sd 6.83818850359465"
```

```r
print(paste('range', range(s.x)))
```

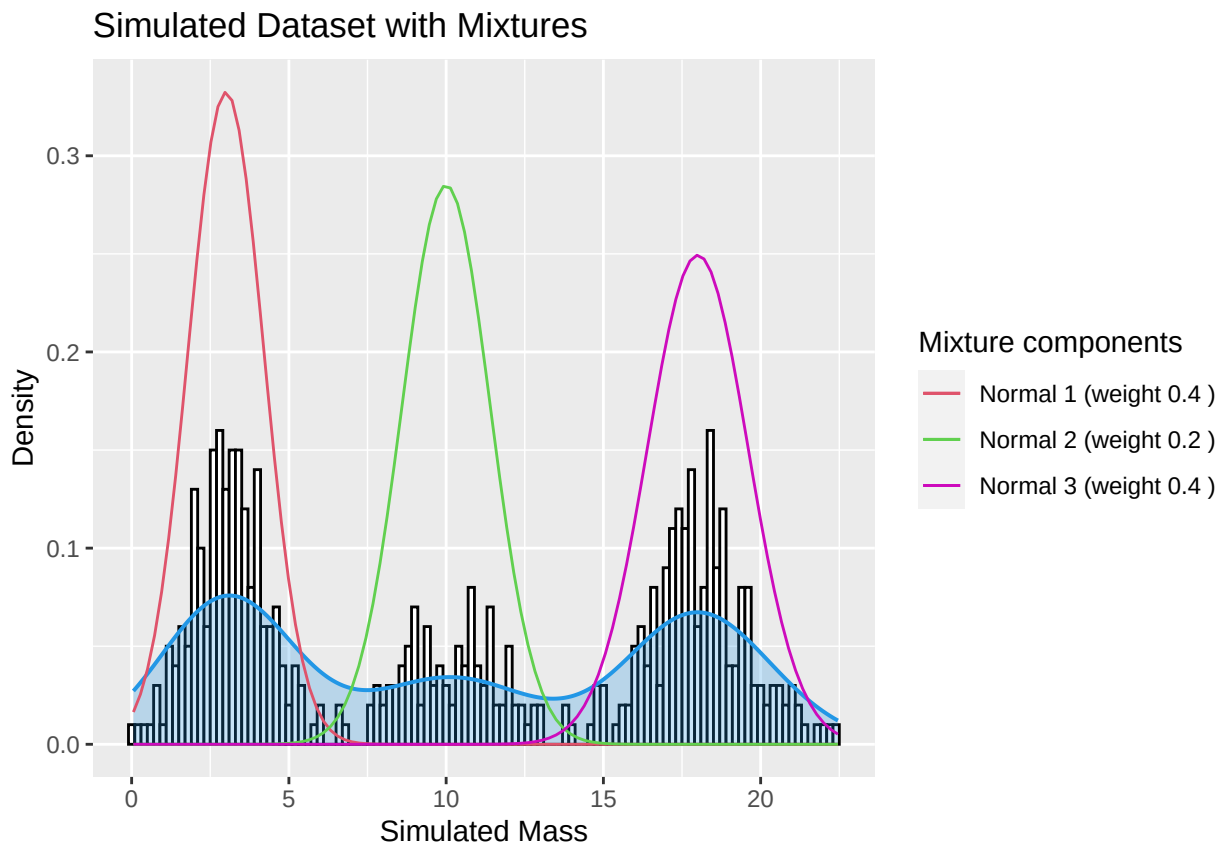
```
## [1] "range 0.0573005842464669" "range 22.4553955106038"
```

```r
print(paste('cv', cv))
```

```
## [1] "cv 65.4545561257252"
```

Visualize the simulated dataset.

```r
# Visualize the simulated dataset
s.df = data.frame(sim_mass = s.x)
hist_plot = ggplot(s.df, aes(x=sim_mass)) +
  geom_histogram(aes(y = after_stat(density)),
                 colour = 1, fill = "white", binwidth = 0.2) +
  geom_density(lwd = 0.7, colour = 4,
               fill = 4, alpha = 0.25)

# Names for the mixture PDF with their respective weight
name1 = paste("Normal 1 (weight", s.w1, ")")
name2 = paste("Normal 2 (weight", s.w2, ")")
name3 = paste("Normal 3 (weight", s.w3, ")")

# Overlay the 3 normal distributions
hist_plot +
  stat_function(fun = dnorm, args = list(mean = s.mu1, sd = s.sig1),
                linewidth = 0.5, aes(colour = name1)) +
  stat_function(fun = dnorm, args = list(mean = s.mu2, sd = s.sig2),
                linewidth = 0.5, aes(colour = name2)) +
  stat_function(fun = dnorm, args = list(mean = s.mu3, sd = s.sig3),
                linewidth = 0.5, aes(colour = name3)) +
  scale_color_manual("Mixture components" ,values = c(10:11, 6)) +
  labs(title = "Simulated Dataset with Mixtures", x = "Simulated Mass", y = "Density") +
  theme(legend.position = "right")
```



Set up the clustering probabilities to be used in the parametric bootstrap.

```
# Set clustering probabilities
b.clt_1 = b.clt_2 = b.clt_3 = rep(NA, s.n)
b.clt_1[s.groups == 1] = 0.9
b.clt_1[s.groups != 1] = 0.05

b.clt_2[s.groups == 2] = 0.9
b.clt_2[s.groups != 2] = 0.05

b.clt_3[s.groups == 3] = 0.9
b.clt_3[s.groups != 3] = 0.05

# Check that the clustering probabilities add up
b.clt_all = cbind(b.clt_1, b.clt_2, b.clt_3)
b.clt_all[1:3, ]
```

```
##      b.clt_1 b.clt_2 b.clt_3
## [1,]     0.9    0.05    0.05
## [2,]     0.9    0.05    0.05
## [3,]     0.9    0.05    0.05
```

```
# Cluster 2 should be 0.9 for observations 201 to 300
b.clt_all[220:222, ]
```

```
##      b.clt_1 b.clt_2 b.clt_3
```

```
## [1,]    0.05     0.9    0.05
## [2,]    0.05     0.9    0.05
## [3,]    0.05     0.9    0.05
```

```
# Cluster 3 should be 0.9 for observations 301 to 500
b.clt_all[420:422, ]
```

```
##      b.clt_1 b.clt_2 b.clt_3
## [1,]    0.05    0.05     0.9
## [2,]    0.05    0.05     0.9
## [3,]    0.05    0.05     0.9
```

Now use the parametric bootstrap to check if the estimates returned by optimizing my log-likelihood function are any good.

```
# True parameters
truth = c(s.mu1, s.mu2, s.mu3,
          s.sig1, s.sig2, s.sig3)

# Number of simulations (left at 10 so that .Rmd knits faster)
## Results seem to stabilise as of 300
## but for actual Law of Large Number use k = 10^4
k = 300

# Bootstrap assuming that the mixture weights are fixed and correct
boot = replicate(k, {
  # Simulate the datset
  x.boot = rep(NA, s.n)
  x.boot[s.groups == 1] = rnorm(sum(s.groups == 1), s.mu1, s.sig1)
  x.boot[s.groups == 2] = rnorm(sum(s.groups == 2), s.mu2, s.sig2)
  x.boot[s.groups == 3] = rnorm(sum(s.groups == 3), s.mu3, s.sig3)

  # Optimise the 6 theta values (mu1, mu2, mu3, tao1, tao2, tao3)
  # and use the truth values as the starting point
  # Use Nelder-Meld as sigma = exp(tao), i.e. sigma has been re-parameterised to be unconstrained
  # Assume that the mixture weights are correctly predicted
  # Assume that the clustering probabilities are correct
  est = optim(truth, function(theta)
    neg.loglik(theta, c(s.w1, s.w2), b.clt_1, b.clt_2, x.boot) )$par

  # Convert taos to sigs
  exp(est[4:5])
  est
})


# Simulated error is each estimation minus the truth
error = apply(boot, 2, function(x) truth - x)

# Overall estimate is the mean of all replications
estimate = apply(boot, 1, mean)

# Bias and mse are the mean and the square of the mean error
```

```r
bias = apply(error, 1, mean)
mse = apply(error^2, 1, mean)

b.summary = data.frame(truth = truth, estimate = estimate, bias = bias, mse = mse)
b.summary
```

```
##   truth  estimate        bias         mse
## 1   3.0  3.946053 -0.94605276 0.901597453
## 2  10.0 10.090910 -0.09091034 0.026206555
## 3  18.0 17.020229  0.97977094 0.973125454
## 4   1.2  1.297315 -0.09731470 0.009625823
## 5   1.4  1.253995  0.14600547 0.021646471
## 6   1.6  1.345613  0.25438689 0.064933665
```

The standard deviation of the simulated dataset is about 6.7. Therefore, an MSE between 0.009 and 0.9 is very low and we can conclude that the code for the log-likelihood function is working correctly.

EM algorithm

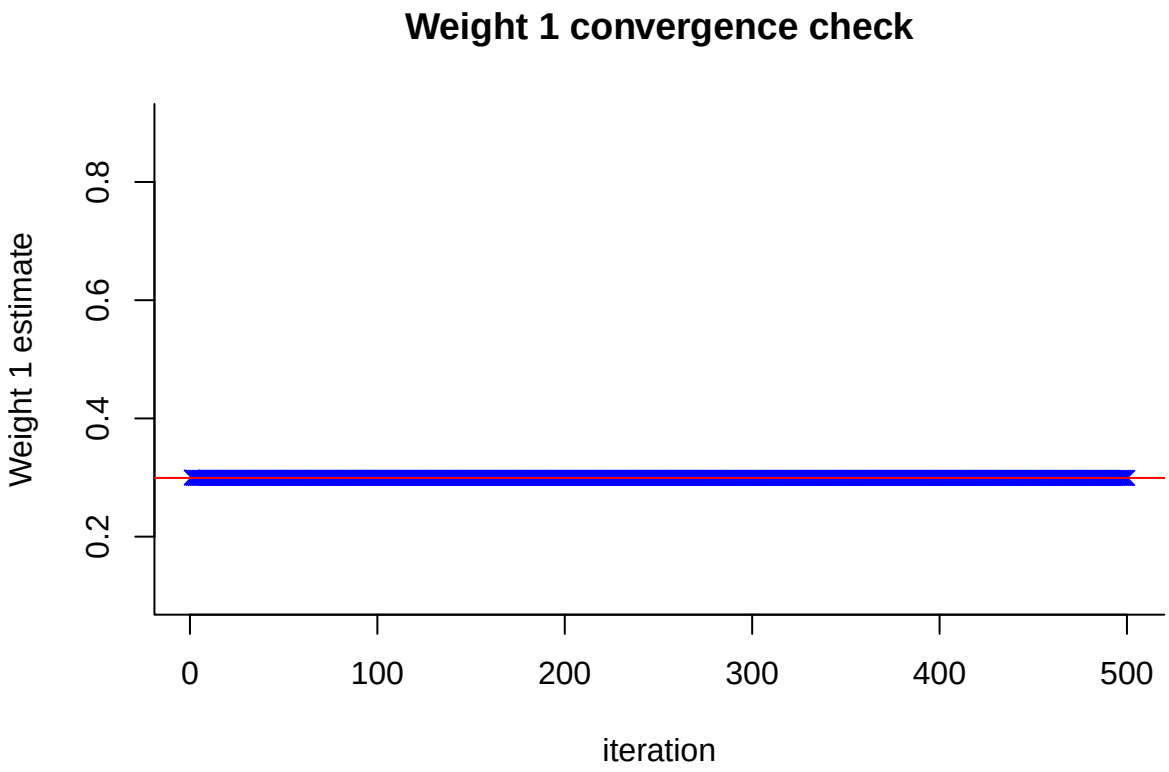
Use the simulated data to check if the EM algorithm runs correctly. Ideally, the classification accuracy is very high and my claim about the order of the factors/weights holds.

Based on my observations, the EM algorithm can fail to converge at random.

```r
# Set seed for reproducibility
set.seed(2)

# Starting values for theta
start_t_sim = c(s.mu1, s.mu2, s.mu3,
           log(sd(s.x)), log(sd(s.x)) + 0.1, log(sd(s.x)) - 0.1)

# Starting values for weights
start_w_sim = c(0.3, 0.36)

# Number of iterations
n.iter = 500

# Run the EM algorithm with the above starting values and starting weight around 1/3
s.final_results = EM_algo(n.iter = n.iter, data = s.x,
                          start_t = start_t_sim, start_w = start_w_sim)
s.final_results[1:3]
```

```
## $loglik
## [1] -2217.959
##
## $weights
## [1] 0.2993415 0.3603531 0.3403054
##
## $theta
## [1] 10.446940 10.612722 10.272534  6.832588  6.837502  6.820296
```

```r
# Check for convergence
#Convergence weight 1
par(bty = 'l')
plot(s.final_results[["weight_iterations"]][, 1],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 1 estimate',
     ylim = c(0.1, 0.9), main = 'Weight 1 convergence check')
abline(h = s.final_results[["weight_iterations"]][n.iter, 1], col = 'red')
```



As we can see from the above plot, the EM algorithm fails to converge. From the EM output we can also read that the algorithm returns the same mean value for all factors as well as the same standard deviation for all factors. However, if the correct starting values are chose at random using the "correct" seed values, the algorithm converges.

I believe that the EM algorithm fails to converge in the `EM_init` function where the clustering probabilities for factor 1 and 2 are drawn from a uniform distribution at random. In the remaining code for the EM algorithm there are no other random draws. However, I cannot exactly explain why random draws might cause convergence issues. Said issue will be left for the future work section.

The below code is exactly the same as the EM code above apart from the seed value.

```r
# Set seed for reproducibility
set.seed(35) # seed 10 converges; seed 35 when knitting the doc

# Starting values for theta
start_t_sim = c(s.mu1, s.mu2, s.mu3,
           log(sd(s.x)), log(sd(s.x)) + 0.1, log(sd(s.x)) - 0.1)
```

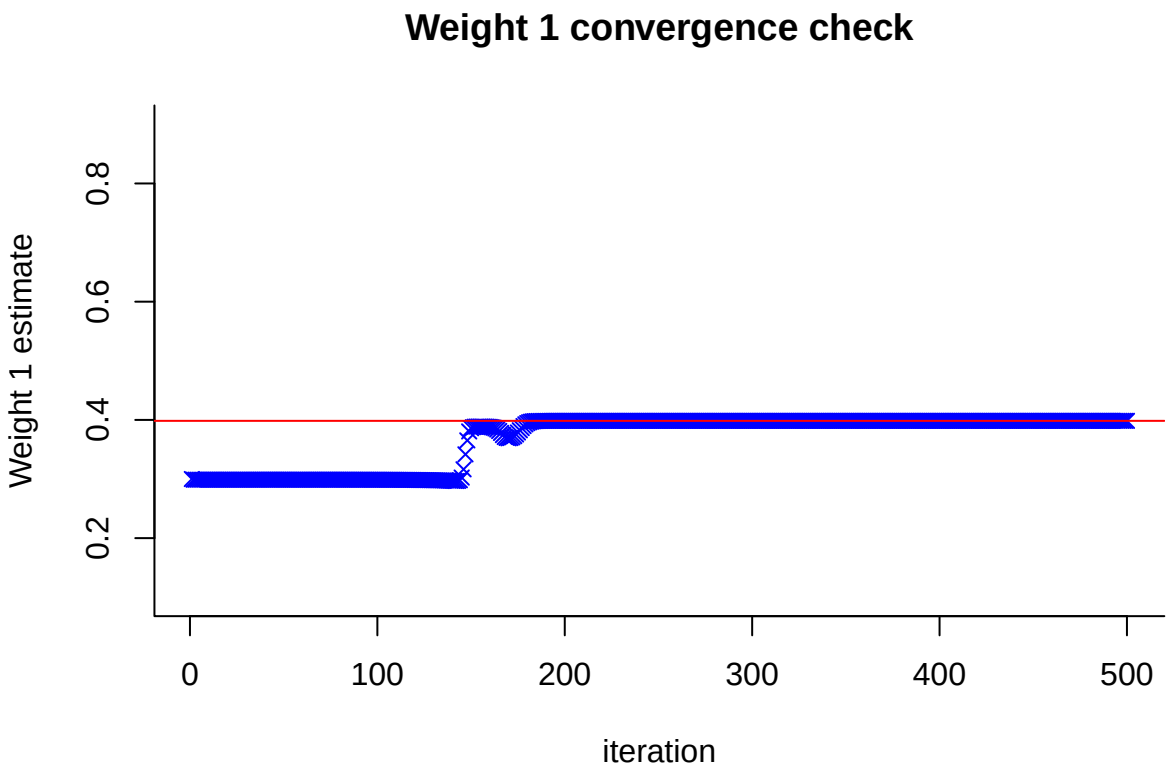
```r
# Starting values for weights
start_w_sim = c(0.3, 0.36)

# Number of iterations
n.iter = 500

# Run the EM algorithm with the above starting values and starting weight around 1/3
s.final_results = EM_algo(n.iter = n.iter, data = s.x,
                          start_t = start_t_sim, start_w = start_w_sim)
s.final_results[1:3]
```

```
## $loglik
## [1] -1408.876
##
## $weights
## [1] 0.3983919 0.2082492 0.3933589
##
## $theta
## [1]  3.074819 10.133488 18.080232  1.145346  1.679901  1.565602
```

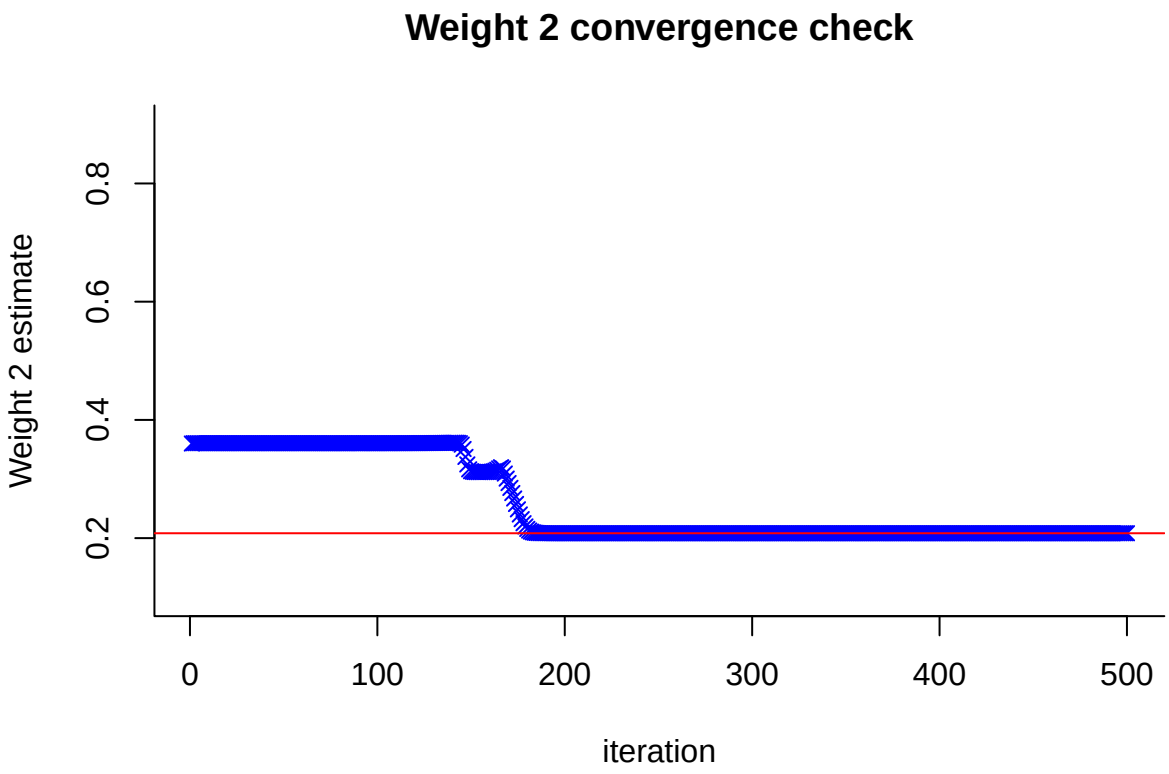
For the correct run, the log-likelihood is much higher, the estimated weights are very close to the true weights, and the estimated theta values are very close to the true theta values.

Now check if the weights converge correctly.

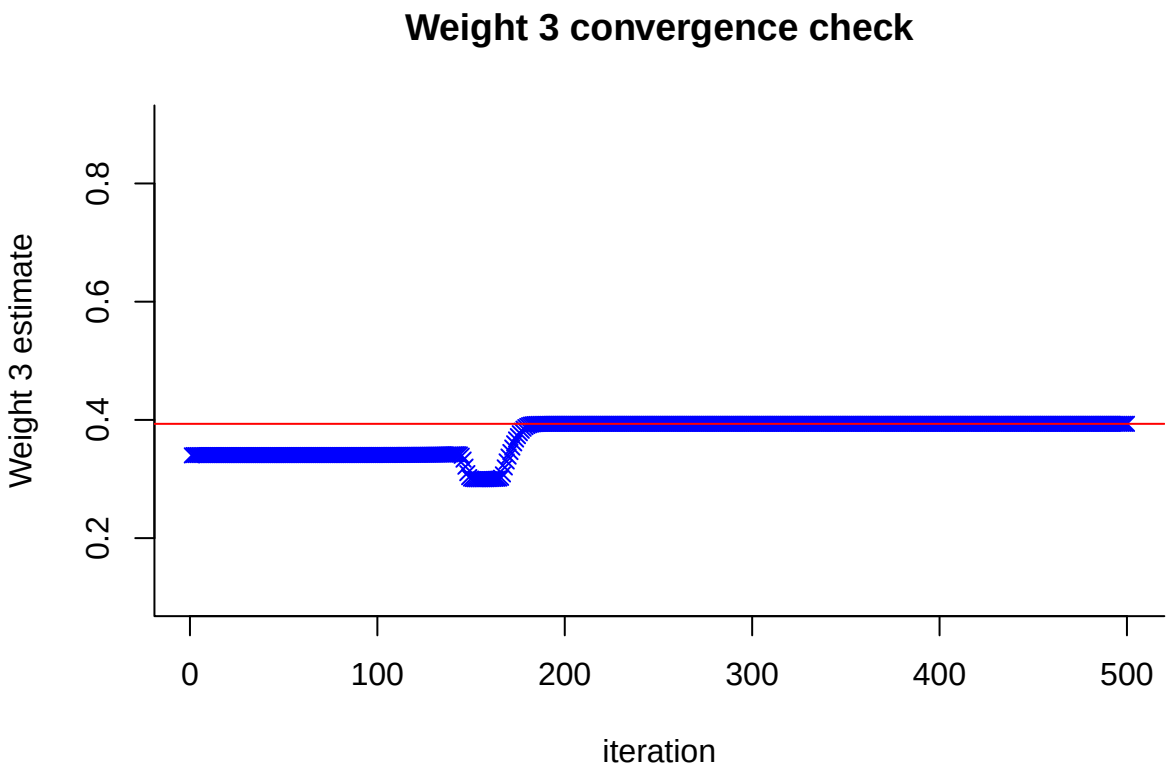
```r
# Check for convergence weight 1
par(bty = 'l')
plot(s.final_results[["weight_iterations"]][, 1],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 1 estimate',
     ylim = c(0.1, 0.9), main = 'Weight 1 convergence check')
abline(h = s.final_results[["weight_iterations"]][n.iter, 1], col = 'red')
```




```
# Check for convergence weight 2
par(bty = 'l')
plot(s.final_results[["weight_iterations"]][, 2],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 2 estimate',
     ylim = c(0.1, 0.9), main = 'Weight 2 convergence check')
abline(h = s.final_results[["weight_iterations"]][n.iter, 2], col = 'red')
```




```
# Check for convergence weight 3
par(bty = 'l')
plot(s.final_results[["weight_iterations"]][, 3],
     xlab = 'iteration', pch = 4, col = 'blue', ylab = 'Weight 3 estimate',
     ylim = c(0.1, 0.9), main = 'Weight 3 convergence check')
abline(h = s.final_results[["weight_iterations"]][n.iter, 3], col = 'red')
```



Compute the classification accuracy on the simulated dataset.

```
# Encode the predicted components
s.max_predComp = apply(s.final_results[['clusters']], 1, function(x) which.max(x) )

# Classification accuracy table
accuracy = table(s.groups, s.max_predComp)
accuracy
```

```
##          s.max_predComp
## s.groups   1   2   3
##        1 200   0   0
##        2   0 100   0
##        3   0   4 196
```

```
# Numeric classification accuracy
sum(diag(accuracy))/s.n
```

```
## [1] 0.992
```

The EM algorithm has a classification accuracy of 99.2% for the simulated dataset. Thus, we can **conclude that the EM algorithm has been correctly implemented** and that we can trust the results produced for the penguins dataset.

Moreover, the estimates for theta are very close to the parameters we chose to simulate the dataset. Thus, we can also trust the theta estimates for the penguins dataset (assuming the underlying distribution matches our assumption). The same holds for the mixture weights which are almost identical to the mixture weights we chose for the simulated datset.

Learnings

1. Used re-parameterized optimisation as I ran into plenty of problems using constrained optimisation (e.g. σ estimated as `1e-5`, leading to `-Inf` in *log* function of log-likelihood when using L-BFGS-B)
2. The debugging tools in R studio are not as great compared to other IDE's like Visual Studio. Used print statements as a debugging tool instead.
3. The factor encoding order corresponds to the order of the factor variable. It is important to NOT re-order the factor variable (e.g. alphabetically) but instead leave it as it first appear in the dataset. E.g for the penguins dataset the species appeared as AGC so the encoding is 1 for A, 2 for G and 3 for C.
4. Limited precision in floating-point arithmetic is real. I encountered the problem in the log-lik function in the clt_row check when:

a = 0.9869627 + 0.01303726 = 0.99999996
b = 0.9994093 + 0.0005906943 = 0.9999999943

When I ran a > 1 would return `false`, but b > 1 would return `true`. This must have happened because of the default tolerance R sets when computing direct comparisons. It definitely caught me off guard when debugging but I was able to fix this issue using the `round` function set to 6 decimals (as problems seem to appear as of the 7th).

Future work

1. Randomly shuffle the group in the simulated dataset and then check if the first element will still be factor 1, i.e. double check learning 3) is correct.
2. Investigate convergence issue of EM algorithm when setting different seed values. Maybe there is a way to increase the probability of convergence at the cost of slightly less precise estimates...