Jared Yu

Algorithms for Data Science

Homework 4

Problem 1

It was mentioned during office hours that the pseudocode portion is not required for this problem. The code for the EM-algorithm will be shown below at the end of the document in **A.1** of the Code Appendix.

Below is a table of iterations for a total of "five" iterations. The table begins with iteration zero, which consists of initializations based on the professor's slides. The initializations exist for only the functions: $m_k^{(i+1)}$, $\sigma_k^{(i+1)}$, and $p_k^{(i+1)}$. The while-loop starts at iteration 1 and utilizes the initializations to help compute first the $g\left(\mathbf{x}_n; \mathbf{m}_k^{(i)}, \boldsymbol{\sigma}_k^{(i)}\right)$ function, before calculating the first $p^{(i)}(k|n)$ function. On the fourth iteration, the results have already converged and so the EM algorithm stops. The algorithm is able to converge relatively quickly, a simple reason is due to the initial data matrix of $\mathbf{x}_n$, where the data can be easily split into two subpopulations and a given center for each subpopulation is not difficult to locate for the EM algorithm. If there were more complicated data points that made it difficult for the algorithm to locate parameters where it could converge then it would require additional iterations.

$Iteration$	$p^{(i)}(k\|n)$				$m_k^{(i+1)}$		$\sigma_k^{(i+1)}$		$p_k^{(i+1)}$	
$initialization$	N/A				2.1766	3.7571	1.1547	1.1547	0.5	0.5
					2.3922	2.9190				
1	0.9302	0.2757	0.8998	0.2041	1.6232	3.6984	0.9303	0.7290	0.5775	0.4225
	0.0698	0.7243	0.1002	0.7959	2.4779	2.5302				
2	0.9986	0.0384	0.9985	0.0355	1.1070	3.9955	0.5289	0.3629	0.5177	0.4823
	0.0014	0.9616	0.0015	0.9645	2.4993	2.5008				
3	1	0	1	0	4	0	0.3536	0.3536	0.5	0.5
	0	1	0	1	2.5	2.5				
4	1	0	1	0	1	4	0.3536	0.3536	0.5	0.5
	0	1	0	1	2.5	2.5				

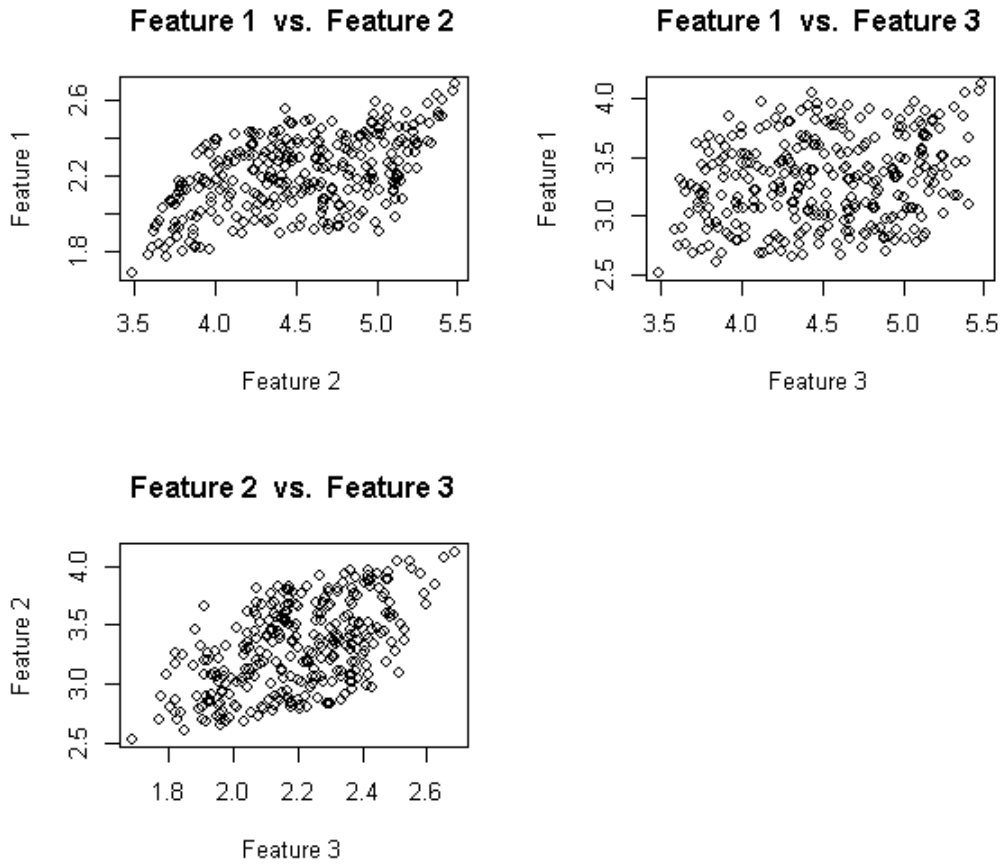
Problem 2

Below is a table of using the EM algorithm from problem 1 on the iris dataset to find the unknown parameters, μ_k, σ_k, and p_k. The table uses a similar format as the table in problem 1. The values for μ_k are found in $\mathbf{m}_k^{(i+1)}$, σ_k in $\sigma_k^{(i+1)}$, and p_k in $p_k^{(i+1)}$. The table shows that convergence took place after 33 iterations.

$Iteration$	$\mathbf{m}_k^{(i+1)}$				$\sigma_k^{(i+1)}$				$p_k^{(i+1)}$		
$initialization$	5.8568	6.0750	6.7465	7.3839	0.9479	0.9479	0.9479	0.9479	$0.\overline{3}$	$0.\overline{3}$	$0.\overline{3}$
	3.0644	3.1793	3.5327	3.8683							
	3.7867	4.2519	5.6834	7.0423							
	1.2117	1.4126	2.0307	2.6175							
33	5.006	5.9052	6.8464		0.2752	0.4041	0.4036		0.3333	0.4139	0.2527
	3.428	2.7489	3.0737								
	1.462	4.4026	5.7305								
	0.246	1.4326	2.0746								

Problem 3

The 300 observations were generated using the same synthesized data technique from the previous homework assignment. The problem has already provided initial values of $\boldsymbol{\mu}$, Σ, min, and max for each of the three features. Below is a plot for each of the three combinations of features:

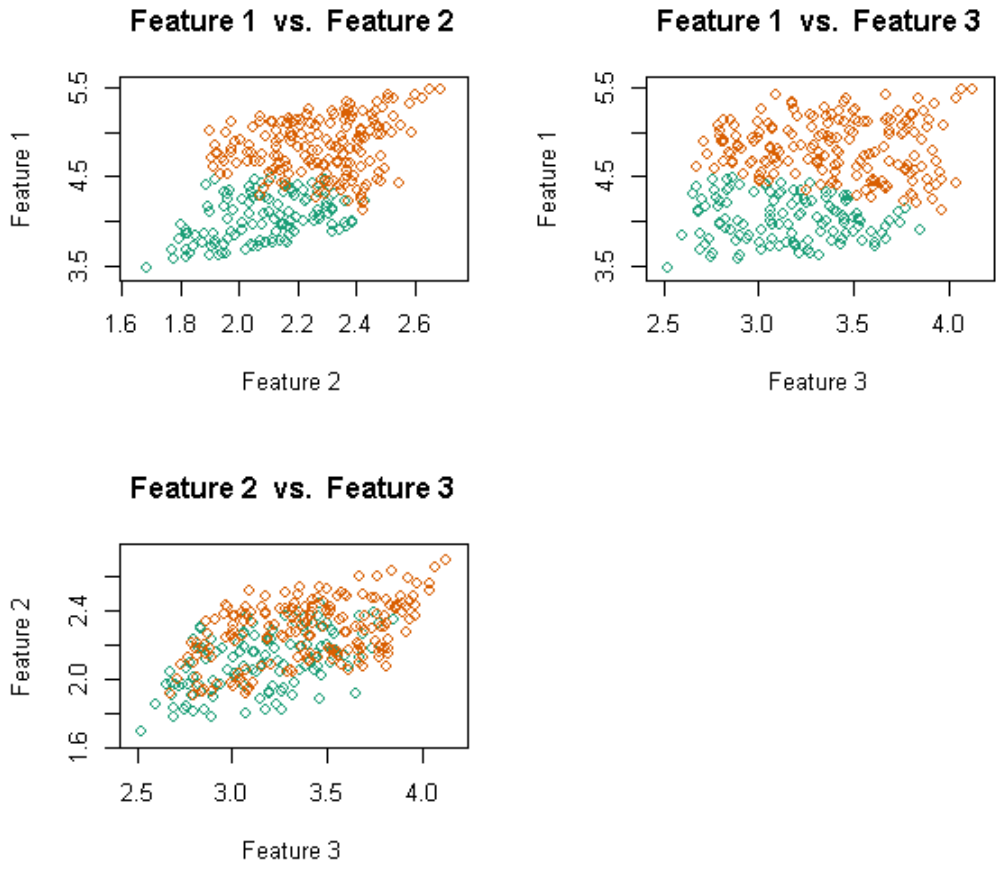


In order to use the EM-algorithm, it is necessary for the user to select an a priori value for K, the number of subpopulations. This is the hidden latent variable that is trying to be estimated. Looking at the data, it is difficult to ascertain with certainty the number of subpopulations that can be determined from the randomized data. It is necessary to choose a $K > 1$, otherwise the EM-algorithm will have no use and have a 100% accuracy on the prediction without fail. The values of $K = 2$ and 3 have been tested.

Below are the results for using $K = 2$, guessing that there is a total of two subpopulations within the data. The estimates of μ_k, σ_k, and p_k can be found below in the table from $\mathbf{m}_k^{(i+1)}$, $\sigma_k^{(i+1)}$, and $p_k^{(i+1)}$ respectively. The algorithm was able to converge after 85 iterations.

Iteration	$\mathbf{m}_k^{(i+1)}$		$\sigma_k^{(i+1)}$		$p_k^{(i+1)}$	
85	4.048	4.811	0.2613	0.2971	0.4076	0.5924
	2.0937	2.2731				
	3.1621	3.3949				

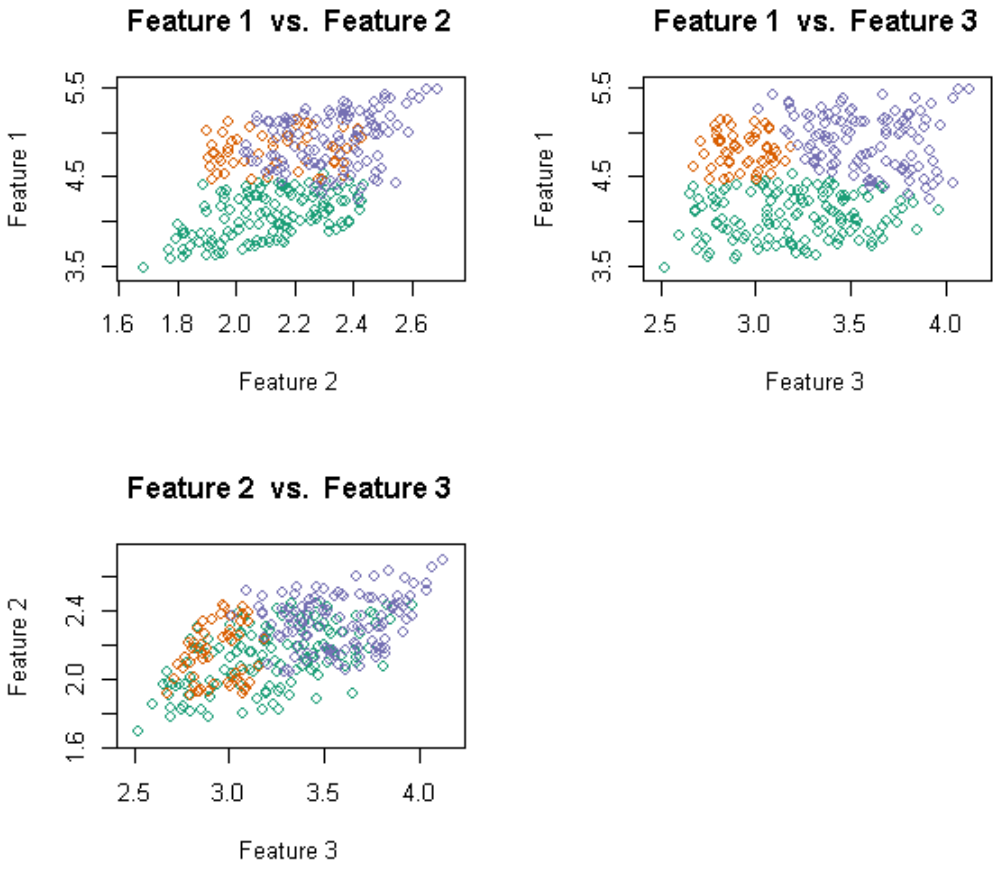
The classifier has predicted that of the 300 observations, 122 observations belong to subpopulation 1 and 178 belong to subpopulation 2.



Below are the results for using $K = 3$, guessing that there is a total of three subpopulations within the data. The estimates of μ_k, σ_k, and p_k can be found below in the table from $\mathbf{m}_k^{(i+1)}$, $\sigma_k^{(i+1)}$, and $p_k^{(i+1)}$ respectively. The algorithm was able to converge after 81 iterations.

Iteration	$\mathbf{m}_k^{(i+1)}$			$\sigma_k^{(i+1)}$			$p_k^{(i+1)}$		
81	4.0677	4.7896	4.8893	0.2641	0.1822	0.2551	0.4519	0.1805	0.3675
	2.115	2.1435	2.3323						
	3.2078	2.9674	3.5767						

The classifier has predicted that of the 300 observations, 135 observations belong to subpopulation 1 and 53 belong to subpopulation 2 and 112 belong to subpopulation 3.



Problem 4

The Bayes classifier in problem 4 is derived from page 5 of the MachineLearning file uploaded to Blackboard. The formula is as follows:

$$P(C_j|\mathbf{x}_0) = \frac{P(C_j)\frac{1}{\sqrt{(2\pi)^n|\Sigma_j|}}\exp\left[-\frac{1}{2}(\mathbf{x}_0-\mu_j)^T\Sigma_j^{-1}(\mathbf{x}_0-\mu_j)\right]}{\sum_{i=1}^{C}P(C_i)\frac{1}{\sqrt{(2\pi)^n|\Sigma_i|}}\exp\left[-\frac{1}{2}(\mathbf{x}_0-\mu_i)^T\Sigma_i^{-1}(\mathbf{x}_0-\mu_i)\right]}.$$

The different classes are determined previously in problem 2. From these predicted labels, the new parameters for mean and variance are calculated and input into the above Bayesian classifier.

The following confusion matrix was given as a result of the classification process:

	True Label			
Classifier Results		1	2	3
	1	50	0	0
	2	0	50	13
	3	0	0	37

The results indicate that accuracy is $91.\bar{3}\%$, slightly better than the result from using just the EM-algorithm of $89.\bar{3}\%$. There was certainly a slight improvement of using the Bayes classifier in combination with the EM-algorithm. The EM-algorithm has already calculated a similar probability, but this provided a set of labels for the dataset. Using the Bayes classifier allowed for the similar conditional probability of $P(C_j|\mathbf{x}_0)$ to be calculated, where a new set of prior and posterior values are generated. It is difficult to understand exactly why the Bayes classifier provides a better result than the similar process of calculating $p(k|n)$. It is also not certain if it will always provide better results, it seems possible that there are cases where the results won't be as good.

In comparison, from the professors' code in Matlab, it was shown that far better results were obtained using more complicated multivariate normal Gaussian distributions that were able to achieve an accuracy of over 96%. There is clearly a large difference in the performance between this simpler model and the other Gaussian models that better utilize the EM algorithm.

In terms of classifiers, it shows that in order to obtain desirable results, it is necessary to study thoroughly both the models and the situations where it is being utilized. Looking at all possible results and examining which model provides the best accuracy is one of the most certain ways to determine what type of model would fit best in a specific dataset.

It seems that there are numerous possible models that can be utilized, and the results may vary depending on the data that has been selected. In this case, it seems that due to the overlapping situations of two of the subpopulations, a more complicated Gaussian model is required in order to obtain superior results. However, it seems that classification may have other situations to consider, such as when deep learning has been used where results are often quite good, but the

interpretation is not quite as bound to these mathematical or statistical formulas. Or there can be cases such as the Parzen window where there is a machine learning type classifier that can truly deliver 100% accuracy results.

Code Appendix

A.1

```r
# Libraries
library(RColorBrewer)
# Jared Yu
# Algorithms for Data Science
# Homework 4

### Problem 1
# The g_function function is used to calculate the kth output of the
# g function in the EM-algorithm (based on lecture slides).
# The function determines the multivariate normal distribution
# for the data matrix when given a set of means and standard deviations.
g_function <- function(X_matrix,
                       column_means_k,
                       sigma_k,
                       number_of_columns) {
  # Calculate the g() for a certain value of k
  return((1 / ((sqrt(2 * pi) * sigma_k))^number_of_columns) *
    exp((-0.5)*(colSums((t(X_matrix - column_means_k))^2) / sigma_k^2)))
}

# The conditional_probablity_kn function is used to calculate the conditional
# probability for some k, where k = 1,...,K, given the dataset. This function
# is looking for the probabilities of the latent variables that are determined
# a priori by the user. The latent variable in this case is K, the number of
# subpopulations in the dataset. The results are known as the responsibilities.
conditional_probability_kn <- function(mixing_probability,
                                       g_output,
                                       total_K) {
  # Calculate the numerator
  numerator <- t(sapply(seq_along(1:total_K), function(index_k) {
    mixing_probability[index_k] * g_output[index_k, ]
  }))

  # Calculate the denominator
  denominator <- matrix(rep(colSums(numerator), each = total_K), nrow = total_K)

  # Return the responsibility that component k takes for explaining
  # the observations of the dataset.
  return(numerator / denominator)
}

# The mixing_probabilities function determines the mixing coefficients for
# each of the subpopulations. The rule is that the mixing coefficients
# should sum to one. The mixing coefficients help to better understand
# the superposition of the K Gaussian desnsities.
mixing_probabilities <- function(number_observations_N,
                                 responsibilities) {
  # Return the mixing coefficients
  return((1 / number_observations_N) * rowSums(responsibilities))
```

```
}

# The means_function function calculates the column means of the dataset
# for each possible value of k, k = 1,...,K. It does so using the responsibilities
# that are found using the conditional_probability_kn function.
means_function <- function(X_matrix,
                           responsibilities,
                           total_K) {
  # Return the matrix of means
  return(sapply(seq_along(1:total_K), function(index_k) {
    t(responsibilities[index_k,] %*% as.matrix(X_matrix)) /
      sum(responsibilities[index_k,])
  }))
}

# The standard_deviation_function function is used to calculate the sigma_k
# function for each iteration of the EM algorithm. It calculates the standard
# deviation for each column of the k subpopulations.
standard_deviations_function <- function(X_matrix,
                                         responsibilities,
                                         weights,
                                         total_K,
                                         total_N,
                                         number_of_dimensions) {
  return(sapply(1:total_K, function(index_k) {
    sqrt(sum(responsibilities[index_k,] %*% ((X_matrix - matrix(rep(weights[, index_k],
           each = total_N), ncol = number_of_dimensions))^2)) /
           (number_of_dimensions * sum(responsibilities[index_k,])))
  }))
}

EM_algorithm <- function(data_matrix,
                         number_of_subpopulations_K,
                         max_iterations = 100,
                         convergence_cutoff = 0.001,
                         sample_data = FALSE) {
  # Helper variables
  data_matrix <- as.matrix(data_matrix)
  dimensions_D <- ncol(data_matrix)  # Number of dimensions in the dataset
  number_observations <- nrow(data_matrix)  # Number of observations in the dataset
  one_vector <- matrix(rep(1, dimensions_D), ncol = 1)  # Vector of ones
  convergence <- FALSE  # Boolean variable for convergence
  iteration <- 0  # Counter for number of iterations
  set.seed(685)  # Set seed and randomizing vector for means
  new_list <- list("conditional_probability" = NULL,  # A list to save output
                   "weighted_means" = NULL, "sigma_k" = NULL,
                   "mixing_coefficient" = NULL)

  # Adjust for sample code
  if (sample_data == TRUE) {
    random_vector <- matrix(c(-0.18671, 0.72579), ncol = 2)
  } else {
    random_vector <- matrix(rnorm(dimensions_D), ncol = dimensions_D)
  }

  # Initialize parameters
  sample_column_mean_vector <- colMeans(data_matrix)  # xbar_col
  sample_std_dev_vector <- sqrt(diag(var(data_matrix)))  # sigma_col
  weighted_means_iteration_i <- round(sample_column_mean_vector %*%
              t(one_vector) + sample_std_dev_vector %*%  random_vector, 4)  # m_k
  sigma_k_iteration_iteration_i <- round(mean(sample_std_dev_vector) %*%
```

```
                                        t(one_vector), 4)  # sigma_k
mixing_coefficient_iteration_i <- t(rep(1, number_of_subpopulations_K)) /
  number_of_subpopulations_K  # p_k

output <- list()  # Helper list to save output
# Go through iterations until convergence or limit
while ((convergence == FALSE) & (iteration < max_iterations)) {
  old_list <- new_list  # Update old_list
  # E-step
  # Calculate the (i + 1)th g function
  g_function_iteration_i <- t(sapply(seq_along(1:number_of_subpopulations_K),
      function(index_k) {
        g_function(X_matrix = data_matrix,
        column_means_k = matrix(rep(weighted_means_iteration_i[, index_k],
          each = number_observations), ncol = dimensions_D),
        sigma_k = sigma_k_iteration_iteration_i[index_k],
        number_of_columns = dimensions_D)
      }))

  # Calculate the responsibilities of each component k
  conditional_probability_iteration_i <- conditional_probability_kn(
    mixing_probability = mixing_coefficient_iteration_i,
    g_output = g_function_iteration_i,
    total_K = number_of_subpopulations_K)
  new_list$conditional_probability <- round(conditional_probability_iteration_i, 4)

  # M-step
  # Calculate the (i + 1)th value for the means function
  weighted_means_iteration_i <- means_function(
    X_matrix = data_matrix,
    responsibilities = conditional_probability_iteration_i,
    total_K = number_of_subpopulations_K)
  new_list$weighted_means <- round(weighted_means_iteration_i, 4)

  # Calculate the column standard deviation of each k for the (i + 1)th iteration
  sigma_k_iteration_iteration_i <- standard_deviations_function(
    X_matrix = data_matrix,
    responsibilities = conditional_probability_iteration_i,
    weights = weighted_means_iteration_i,
    total_K = number_of_subpopulations_K,
    total_N = number_observations,
    number_of_dimensions = dimensions_D)
  new_list$sigma_k <- round(sigma_k_iteration_iteration_i, 4)

  # Calculate the mixing coefficient for each k group for the (i + 1th) iteration
  mixing_coefficient_iteration_i <- mixing_probabilities(
    number_observations_N = number_observations,
    responsibilities = conditional_probability_iteration_i)
  new_list$mixing_coefficient <- round(mixing_coefficient_iteration_i, 4)

  # Update counter for number of iterations
  iteration <- iteration + 1

  # Update list with new list of elements
  output <- c(output, list(new_list))

  # Check for convergence
  if (iteration > 1) {
    list_difference <- Map("-", old_list, new_list)
    convergence_score <- sum(sapply(list_difference,
                function(list_index) sum(abs(list_index))))
```

```
      if (convergence_score < convergence_cutoff)
        convergence <- TRUE
      print(c(iteration, convergence_score))
    }
  }

  # Name each iteration
  names(output) <- paste("Iteration", 1:length(output), sep = '')

  # Return output list
  return(output)
}

# References: https://stackoverflow.com/questions/27594541/export-a-list-into-a-csv-or-txt-file-in-r
# Create the sample matrix from the slides
sample_matrix <- matrix(c(1, 4, 1, 4, 2, 2, 3, 3), ncol = 2)

# Use EM algorithm on sample matrix from problem 1
sample_output <- EM_algorithm(data_matrix = sample_matrix,
                              number_of_subpopulations_K = 2,
                              max_iterations = 6,
                              sample_data = TRUE)

# Output results to table
lapply(sample_output, function(list_index)
  write.table(data.frame(list_index), 'sample_data.txt', append = TRUE, sep = ','))
```

A.2

```
### Problem 2
# Use EM algorithm on iris data
iris_data <- iris[,1:4]
iris_output <- EM_algorithm(data_matrix = iris_data,
                            number_of_subpopulations_K = 3,
                            max_iterations = 100)

# Output results to table
lapply(iris_output$iteration33, function(list_index)
  write.table(data.frame(list_index), 'em_iris.txt', append = TRUE, sep =',' ))

### Problem 3
mean_vector <- c(4.5, 2.2, 3.3)
variance_covariance_matrix <- matrix(c(0.5, 0.1, 0.05,
                                       0.1, 0.25, 0.1,
                                       0.05, 0.1, 0.4), ncol = 3)
column_minimum <- c(3.5, 1.7, 2.5)
column_maximum <- c(5.5, 2.7, 4.1)

set.seed(685)  # Set seed so that random numbers are constant
# The synthesize_iris_data function is a function to be used on each class of the
# iris data. It will generate 100 new observations for each class that are random.
# It will rotate them using a covariance matrix and normalize them with the assistance
# of a helper function, normalize_column().
synthesize_iris_data <- function(covariance_data = variance_covariance_matrix,
                                 minimum_data = column_minimum,
                                 maximum_data = column_maximum,
                                 number_of_generated_data) {
  # Number of features
```

```r
  number_of_features <- length(minimum_data)

  # Random matrix of 300 observations
  random_matrix <- matrix(rep(runif(n = number_of_generated_data *
                                    number_of_features)), ncol = number_of_features)

  # Rotate the random matrix according to the covariance matrix
  rotated_data <- as.data.frame(random_matrix %*% covariance_data)

  # Normalize each column of the rotated data
  normalized_data <- sapply(seq_along(rotated_data), function(index)
    normalize_column(column_index = index,
                     random_data = rotated_data,
                     minimums = minimum_data,
                     maximums = maximum_data))

  # Return the normalized data
  return(normalized_data)
}

# The normalize_column function is a helper function that is used within
# the synthesize_iris_data() function. It is repeated for each column
# of a given class, where it will normalize the random data.
normalize_column <- function(random_data,
                             column_index,
                             minimums,
                             maximums) {
  # Select the current column from the rotated data
  rotated_data_column <- random_data[, column_index]

  # Create a Pmin and Pmax for the min and max of the rotated data
  random_min <- min(rotated_data_column)
  random_max <- max(rotated_data_column)

  # Create a min and max from the actual iris data
  actual_min <- minimums[column_index]
  actual_max <- maximums[column_index]

  # Normalize the rotated data
  rotated_data_column <- ((rotated_data_column - random_min) /
    (random_max - random_min)) *
    (actual_max - actual_min) +
    actual_min

  # Return the normalized data
  return(rotated_data_column)
}

# Generate 300 observations per class, rotated and normalized
synthetic_data <- synthesize_iris_data(number_of_generated_data = 300)

# Mean shift the data
synthetic_data_mean <- colMeans(synthetic_data)
synthetic_mean_shift <- synthetic_data_mean - mean_vector
synthetic_mean_vector <- t(as.matrix(synthetic_mean_shift) %*% rep(1, 300))

# Final synthesized dataset
synthetic_data_final <- synthetic_data - synthetic_mean_vector

# Estimate the parameters of the synthesized data (K = 2, 3)
synthetic_EM_2 <- EM_algorithm(data_matrix = synthetic_data_final,
```

```
            max_iterations = 100,
            number_of_subpopulations_K = 2,
            convergence_cutoff = 0.001)

synthetic_EM_3 <- EM_algorithm(data_matrix = synthetic_data_final,
            max_iterations = 100,
            number_of_subpopulations_K = 3,
            convergence_cutoff = 0.001)

# Write data to text document
sapply(2:4, function(index) {
  write.table(synthetic_EM_2$Iteration85[[index]], 'em_synth_2.txt', append = TRUE, sep = ',')
})

sapply(2:4, function(index) {
  write.table(synthetic_EM_3$Iteration81[[index]], 'em_synth_3.txt', append = TRUE, sep = ',')
})

# Create scatterplot for all combinations
feature_names <- c("Feature 1", "Feature 2", "Feature 3")
plot_combinations <- t(combn(x = 1:3, m = 2))
par(mfrow = c(2,2))
apply(plot_combinations, 1, function(index) {
  plot(synthetic_data_final[,index[1]], synthetic_data_final[,index[2]],
       main = paste(feature_names[index[1]], " vs. ", feature_names[index[2]]),
       xlab = feature_names[index[2]], ylab = feature_names[index[1]])
})
dev.off()

# Plot limits for x,y-lims
feature_1_limits <- c(min(synthetic_data_final[,1]), max(synthetic_data_final[,1]))
feature_2_limits <- c(min(synthetic_data_final[,2]), max(synthetic_data_final[,2]))
feature_3_limits <- c(min(synthetic_data_final[,3]), max(synthetic_data_final[,3]))
plot_limits <- matrix(c(feature_1_limits, feature_2_limits, feature_3_limits), ncol = 3)

# Plot the classifier's subpopulations (K=2)
random_classifier_2 <- synthetic_EM_2$Iteration85$conditional_probability
random_classifier_results_2 <- apply(random_classifier_2, 2,
                                     function(index_i) c(1:3)[which.max(index_i)])

synthetic_subpopulations_2 <- as.data.frame(synthetic_data_final)
synthetic_subpopulations_2['labels'] <- random_classifier_results_2

# Plot the classifier's subpopulations (K=3)
random_classifier_3 <- synthetic_EM_3$Iteration81$conditional_probability
random_classifier_results_3 <- apply(random_classifier_3, 2,
                                     function(index_i) c(1:3)[which.max(index_i)])

synthetic_subpopulations_3 <- as.data.frame(synthetic_data_final)
synthetic_subpopulations_3['labels'] <- random_classifier_results_3

# Reference: https://stackoverflow.com/questions/4785657/how-to-draw-an-empty-plot
# The subpopulation_plots function is used to help plot the different 2-d plots
# of the different classes. The function will plot all the different combinations
# of the features in a separate plot.
subpopulation_plots <- function(plot_data,
                                number_of_subpopulations_K,
                                xy_limits,
                                column_names) {
  plot_colors <- brewer.pal(3, "Dark2")  # Possible color combinations
  plot_combinations <- t(combn(x = 1:3, m = 2))  # Possible plot combinations
```

```
  apply(plot_combinations, 1, function(index_i) {  # Plot the boundaries and titles
    plot(1,
       type = "n",
       xlim = c(xy_limits[1, index_i[2]] - 0.05, xy_limits[2, index_i[2]] + 0.05),
       ylim = c(xy_limits[1, index_i[1]] - 0.05, xy_limits[2, index_i[1]] + 0.05),
       main = paste(column_names[index_i[1]], " vs. ", column_names[index_i[2]]),
       xlab = column_names[index_i[2]],
       ylab = column_names[index_i[1]])

    sapply(1:number_of_subpopulations_K, function(kth_subpopulation) {  # Plot the points
      subpopulation_points(index_j = index_i,
                           index_k = kth_subpopulation,
                           point_data = plot_data,
                           color_set = plot_colors)
    })
  })
}

# The subpopulation_points function is used as a helper
# function to plot the different points per class in a specific
# plot from the subpopulation_plots() function.
subpopulation_points <- function(index_j,
                                 index_k,
                                 point_data,
                                 color_set) {
  # index_j refers to the set of possible features, feature 1, 2, and 3
  # index_k refers to the subpopulation label k, k = 1,2 or 1,2,3
  points(point_data[point_data$labels == index_k, index_j[2]],
         point_data[point_data$labels == index_k, index_j[1]],
         col = color_set[index_k])
  # kde_contour <- kde2d(point_data[point_data$labels == index_k, index_j[2]],
        # point_data[point_data$labels == index_k, index_j[1]])
  # contour(kde_contour, add = T, col = color_set[index_k])
}

# Subpopulation plots (K=2)
par(mfrow = c(2,2))
subpopulation_plots(plot_data = synthetic_subpopulations_2,
                    number_of_subpopulations_K = 2,
                    xy_limits = plot_limits,
                    column_names = feature_names)
dev.off()
# Subpopulation plots (K=3)
par(mfrow = c(2,2))
subpopulation_plots(plot_data = synthetic_subpopulations_3,
                    number_of_subpopulations_K = 3,
                    xy_limits = plot_limits,
                    column_names = feature_names)
dev.off()

### Problem 4
# Select the final results of using EM algorithm on the iris data
final_classifier <- iris_output$Iteration33$conditional_probability

# Determine which classes are predicted using the classifier
classifier_results <- apply(final_classifier, 2,
                            function(index_i) c(1:3)[which.max(index_i)])

bayes_data <- iris_data  # Create list data
bayes_data['label'] <- classifier_results
classifier_list <- split(bayes_data[,1:4], bayes_data$label)
```

```r
probability_class_j <- table(classifier_results) /  # prior probabilities
  length(classifier_results)


# The multivariate_normal_pdf function calculates the multivariate
# normal pdf for as the posterior probability. It requires the
# parameters of variance and mu for a given class. It utilizes
# a separate mahalnobis_distance() function to help complete
# the calculation.
multivariate_normal_pdf <- function(variance_covariance_i,
                                    mean_i,
                                    data_set) {
  number_of_observations_n <- nrow(data_set)  # Save the value for n

  return(  # Return the multivariate normal pdf
    ((((2 * pi)^number_of_observations_n) *
    det(variance_covariance_i))^(-0.5)) *
    exp((-0.5) *
    mahalanobis_distance(X_matrix = data_set,
                         mean_vector = mean_i,
                         variance_covariance_matrix = variance_covariance_i))
  )
}

# The mahalanobis_distance function is a helper function used
# with the multivariate_normal_pdf() function to help calculate
# the mahalanobis distance portion of the calculation.
mahalanobis_distance <- function(X_matrix,
                                 mean_vector,
                                 variance_covariance_matrix) {
  # Helper variables
  one_vector <- rep(1, nrow(X_matrix))
  mean_matrix <- as.matrix(one_vector) %*% t(as.matrix(mean_vector))
  mean_subtracted_matrix <- as.matrix(X_matrix - mean_matrix)

  # Calculate the values for the squared mahalanobis distance
  mahalanobis_values <- apply(mean_subtracted_matrix, 1, function(index) {
    t(index) %*%
      solve(variance_covariance_matrix) %*%
      index
  })

  # Return the squared Mahalanobis distance
  return(mahalanobis_values)
}

# The bayes_classifier function is used to calculate the probability
# that an observation belongs to a certain class. It uses classes that
# have been determined using the EM_algorithm to derive new parameters
# for the classifier.
bayes_classifier <- function(list_data,
                             class_probability) {
  # Retrieve X matrix
  X_matrix <- do.call(rbind, list_data)
  rownames(X_matrix) <- NULL

  # Calculate variance-covariance matrices
  variance_covariance_list <- lapply(list_data, var)

  # Calculate mean vectors
```

```r
  mean_list <- lapply(list_data, function(list_index) {
    apply(list_index, 2, function(index_i) colMeans(as.data.frame(index_i)))
  })

  # Calculate multivariate normal pdfs
  multivariante_normal_list <- lapply(seq_along(list_data), function(index_i) {
    multivariate_normal_pdf(data_set = X_matrix,
                            variance_covariance_i = variance_covariance_list[[index_i]],
                            mean_i = mean_list[[index_i]])
  })

  # Calculate the product of the prior with each corresponding posterior
  prior_posterior_list <- lapply(seq_along(list_data), function(list_index) {
    class_probability[list_index] * multivariante_normal_list[[list_index]]
  })

  # Create the denominator for the conditional probability
  bayes_denominator <- Reduce("+", prior_posterior_list)

  # Calculate the responsibilities per class
  responsibility_list <- lapply(seq_along(list_data), function(list_index) {
    prior_posterior_list[[list_index]] /
      bayes_denominator
  })

  # Change the list to a dataframe of responsibilities
  responsibility_data_frame <- as.data.frame(do.call(cbind, responsibility_list))

  # Output the labels which have the highest probabilities per observation
  apply(responsibility_data_frame, 1, function(index) c(1:3)[which.max(index)])
}

# Create confusion matrix of results and determine the accuracy
bayes_result <- bayes_classifier(list_data = classifier_list,
                                 class_probability = probability_class_j)
true_label <- rep(c(1, 2, 3), each = 50)
table(bayes_result, true_label)
sum(diag(table(bayes_result, true_label))) / 150
sum(diag(table(classifier_results, true_label))) / 150
```