EM ALGORITHM

Daniel Losada and Gabriel Pons

2025-03-10

1. Introduction

In this project, we explore the application of the Expectation–Maximization (EM) algorithm for Gaussian mixtures, motivated by its central role in unsupervised learning and its versatility in clustering and density estimation. We coded our approach to both generate synthetic data and segment real photographs, allowing us to analyze the convergence behavior and practical utility of the algorithm. Our goal is to understand how the iterative procedure refines parameter estimates and improves cluster assignments over time.

It is highly recommended to follow this report with the HTML version as it contains some interactive plots.

2. Auxiliar Functions

In this section we explain all the functions that we have created in order to carry the project (excluding the photographic section, which will be covered afterwards).

The first function that we defined is `initialize_values`.

```
initialize_values <- function(data, n, d, K) {
  set.seed(123)  # Ensure reproducibility

  # Initialize mixing proportions (equal for all Gaussians)
  Pis <- rep(1/K, K)

  # Initialize means randomly chosen from the dataset
  Mus <- data[sample(1:n, K), , drop=FALSE]  # Select K random points

  # Initialize covariance matrices with the overall covariance of the data
  Sigmas <- lapply(1:K, function(i) cov(data))  # Copy initial covariance for all clusters

  return(list(Pis = Pis, Mus = Mus, Sigmas = Sigmas))
}
```

We coded the function `initialize_values` to set up the initial parameters for our Gaussian mixture model. First, we invoke `set.seed(123)` to ensure reproducibility. Then, we define the mixing proportions π_k to be equal for all K components by using `rep(1/K, K)`. Next, we randomly select K points from the dataset as the initial means μ_k. Lastly, we compute the overall covariance matrix of the data with `cov(data)` and replicate it K times to obtain Σ_k. The function then returns a list containing `Pis`, `Mus`, and `Sigmas`.

Below is our explanation for the `convert_to_rgb` function. In this function, we transform a matrix of segmentation labels into a colored image. We coded it as follows:

```r
convert_to_rgb <- function(K, segmented_img) {
  # Generate K distinct colors using the rainbow palette
  my_colors <- rainbow(K)

  # Get image dimensions (height and width) from the segmentation matrix
  height <- nrow(segmented_img)
  width  <- ncol(segmented_img)

  # Create an empty 3D array for the output RGB image (height x width x 3 channels)
  segmented_rgb <- array(0, dim = c(height, width, 3))

  # Loop over each cluster index from 1 to K
  for (k in seq_len(K)) {
    # Convert the k-th color to an RGB vector scaled to [0, 1]
    rgb_vals <- col2rgb(my_colors[k]) / 255

    # Create a logical mask where the segmentation equals k
    mask <- (segmented_img == k)

    # Assign the RGB values to the corresponding channels for pixels in cluster k
    segmented_rgb[,,1][mask] <- rgb_vals[1]  # Red channel
    segmented_rgb[,,2][mask] <- rgb_vals[2]  # Green channel
    segmented_rgb[,,3][mask] <- rgb_vals[3]  # Blue channel
  }

  return(segmented_rgb)
}
```

We coded this function to work seamlessly within our project. First, we use `rainbow(K)` to generate a palette of K distinct colors, one for each Gaussian component. The segmentation result, stored in `segmented_img`, is assumed to be a matrix where each element is an integer label corresponding to the cluster assigned to that pixel.

Next, we obtain the image dimensions with `nrow(segmented_img)` and `ncol(segmented_img)`, and create an empty three-dimensional array `segmented_rgb` with dimensions $height \times width \times 3$. This array will store the final image in RGB format.

Within the loop, for each cluster k, we extract the kth color from our palette and convert it to an RGB vector scaled to the range $[0, 1]$ using `col2rgb(my_colors[k]) / 255`. Then, by creating a boolean mask that identifies all pixels where the segmentation equals k, we assign the corresponding RGB values to the red, green, and blue channels in `segmented_rgb`.

Finally, the function returns the `segmented_rgb` array, which is now a color image representing our segmentation.

Below is our explanation for the function `generate_gaussian_mixture_data`. In this function, we generate synthetic data from a mixture of Gaussians in d dimensions with n observations and K clusters. We coded it in R as follows:

```r
generate_gaussian_mixture_data <- function(n, d, K, seed=42) {
  set.seed(seed)

  # Equal proportion
  Pis <- rep(1/K, K)

  # Generate random means and covariance matrices for each Gaussian
  Mu <- lapply(1:K, function(i) runif(d, min = 0, max = 10))  # Random means in [0,10]
  Sigma <- lapply(1:K, function(i) {
    A <- matrix(runif(d*d, min=0.5, max=3), d, d)  # Random matrix
    return(t(A) %*% A)  # Ensures positive semi-definiteness
  })

  # Initialize data
  data <- matrix(0, n, d)
  z <- sample(1:K, n, replace=TRUE, prob=Pis)  # Assign clusters

  for (i in 1:n) {
    data[i, ] <- rmvnorm(1, Mu[[z[i]]], Sigma[[z[i]]])
  }

  return(list(data = data, z = z, Mu = Mu, Sigma = Sigma, Pis = Pis))
}
```

We coded this function as part of our project to create a synthetic dataset that follows a Gaussian mixture model. First, we set the random seed with `set.seed(seed)` to ensure reproducibility.

Then, we initialize the mixing proportions using

$$\pi_k = \frac{1}{K} \quad \text{for } k = 1, \ldots, K,$$

implemented as `Pis <- rep(1/K, K)`.

Next, we generate random mean vectors for each Gaussian component. We use `runif` to generate d random numbers in the interval $[0, 10]$, and we store these in the list `Mu`. In parallel, we generate a random covariance matrix for each component. For each cluster, we create a random matrix `A` with elements sampled uniformly from $[0.5, 3]$, and then compute

$$\Sigma_k = A^T A,$$

using `t(A) %*% A`. This guarantees that each covariance matrix is positive semi-definite. These matrices are stored in the list `Sigma`.

After generating the parameters, we initialize an empty data matrix `data` of dimensions $n \times d$. We then assign each observation to a cluster by sampling from
$1, 2, \ldots, K$

with equal probability (`Pis`). For each observation i, we generate a data point from the multivariate normal distribution

$$x_i \sim N(\mu_{z[i]}, \Sigma_{z[i]}),$$

using `rmvnorm`. The true cluster assignments are stored in the vector `z`.

Finally, the function returns a list containing the generated data matrix `data`, the true labels `z`, the list of mean vectors `Mu`, the list of covariance matrices `Sigma`, and the mixing proportions `Pis`.

This function forms the foundation of our project by providing a realistic synthetic dataset that we later use to test and evaluate our EM algorithm implementation.

The EM algorithm in our project relies in the following functions: `compute_gamma` and `compute_l`

We coded the function `compute_gamma` to compute the responsibilities (posterior probabilities) for each data point belonging to each cluster. In our implementation, for each observation x_i and each cluster k, we calculate

$$\gamma_{ik} = \frac{\pi_k \, \phi_k(x_i)}{\sum_{j=1}^{K} \pi_j \, \phi_j(x_i)},$$

where `pi` represents the mixing proportions and `phi` holds the density values computed by the multivariate normal probability density function. The code multiplies `pi` by `phi` and then normalizes each row so that the probabilities sum to 1.

```
# Compute the responsibilities (gamma values) for each component.
# These represent the posterior probabilities that each data point belongs to each cluster.
compute_gamma <- function(pi, phi) {
  # Multiply the mixing proportions `pi` by the component densities `phi`
  gamma <- phi * pi
  # Normalize each row so that the sum over components equals 1
  gamma <- gamma / rowSums(gamma)
  return(gamma)
}
```

We coded the function `compute_l` to compute the overall log-likelihood of the data given the current model parameters. For each observation, the mixture density is

$$\sum_{k=1}^{K} \pi_k \, \phi_k(x_i),$$

and the log-likelihood is computed as

$$l = \sum_{i=1}^{n} \log \Big(\sum_{k=1}^{K} \pi_k \, \phi_k(x_i) \Big).$$

This function simply returns the sum of the log of the row sums of `phi * pi`.

```
# Compute the log-likelihood of the data given the current parameters.
# This is done by summing the log of the overall mixture density for each observation.
compute_l <- function(pi, gamma, phi) {
  return(sum(log(rowSums(phi * pi))))
}
```

3. EM Algorithm Generalization

```
# It computes the responsibilities, updates the parameters, and stores the evolution
# history for the first 8 iterations.


# The EM_algorithm function takes as input:
# - `data`: an n x d matrix of observations.
# - `pi`: initial mixing proportions.
# - `mu`: initial means (one for each component).
# - `sigma`: initial covariance matrices (one for each component).
# - `tol`: the convergence tolerance for changes in the log-likelihood.
# - `verbose`: if TRUE, prints intermediate log-likelihood values.

EM_algorithm <- function(data, pi, mu, sigma, tol = 1e-2, verbose = FALSE) {
  # Get the number of components (K), number of data points (n) and dimensions (d)
  K <- length(pi)
  n <- nrow(data)
  d <- ncol(data)

  # Compute initial density values for each Gaussian component using `dmvnorm`
  phi <- sapply(1:K, function(k) dmvnorm(data, mu[k,], sigma[[k]]))
  # Compute the initial responsibilities using the current parameters
  gamma <- compute_gamma(pi, phi)

  # Compute the initial log-likelihood
  L_temp <- compute_l(pi, gamma, phi)
  if(verbose){ cat("Initial log-likelihood: ", L_temp, "\n") }

  L <- 0
  iteration_counter <- 0

  # Store histories for the first 8 iterations for visualization later
  gamma_history <- list()
  mu_history <- list()
  sigma_history <- list()

  # Iterate until the absolute change in log-likelihood is less than the tolerance
  while (abs(L - L_temp) >= tol) {
    L <- L_temp

    ## Expectation step: Recompute responsibilities with current parameters.
    gamma <- compute_gamma(pi, phi)
    # Store the current gamma, mu, and sigma if within the first 8 iterations.
    if(iteration_counter < 8) {
      gamma_history[[iteration_counter + 1]] <- gamma
      mu_history[[iteration_counter + 1]] <- mu
      sigma_history[[iteration_counter + 1]] <- sigma
    }

    ## Maximization step:
    # Compute the effective number of points assigned to each component:
    Nk <- colSums(gamma)

    # Update the mixing proportions
```

```
    pi <- Nk / n

    # Update the means as the weighted average of the data points
    mu <- t(gamma) %*% data / Nk

    # Update the covariance matrices as the weighted covariance
    sigma <- lapply(1:K, function(k) {
      X_centered <- sweep(data, 2, mu[k,])
      sigma_k <- (t(X_centered) %*% (X_centered * gamma[, k])) / Nk[k]
      # Regularize the covariance matrix to avoid singularities.
      epsilon <- 1e-6
      sigma_k <- sigma_k + diag(epsilon, ncol(data))
      return(sigma_k)
    })

    # Recompute the density values with updated parameters
    phi <- sapply(1:K, function(k) dmvnorm(data, mu[k,], sigma[[k]]))

    # Compute new log-likelihood
    L_temp <- compute_l(pi, gamma, phi)
    iteration_counter <- iteration_counter + 1
    if(verbose){
      cat("Iteration: ", iteration_counter, " | Current log-likelihood: ", L_temp, "\n")
    }
  }

  # Return final parameters and the history for further visualization.
  return(list(n_iterations = iteration_counter, pi = pi, mu = mu, sigma = sigma,
              gamma = gamma, gamma_history = gamma_history, mu_history = mu_history,
              sigma_history = sigma_history))
}
```

The `EM_algorithm` function implements the iterative procedure of the EM algorithm for a Gaussian mixture model. Initially, we extract the number of clusters K, the number of observations n, and the number of dimensions d from the input data. We then compute the initial density values for each cluster using the function `dmvnorm` and determine the initial responsibilities by calling `compute_gamma`. At this stage, the initial log-likelihood is computed using `compute_l`, and if the `verbose` flag is set to `TRUE`, we print this value to track our progress. This initialization provides the starting point for the subsequent iterative updates.

To facilitate later visualizations and a better understanding of the convergence behavior, we also store the evolution of key parameters during the first eight iterations. Specifically, we create lists— `gamma_history`, `mu_history`, and `sigma_history` (to record the responsibilities, means, and covariance matrices at each iteration). These stored histories allow us to later visualize how the model parameters evolve as the algorithm converges.

During the iterative update process, the algorithm recalculates the responsibilities γ using `compute_gamma` at each iteration. If the iteration counter is less than 8, we save the current values of γ, the means `mu`, and the covariance matrices `sigma` in their respective history lists. Next, we compute the effective number of points for each cluster as

$$N_k = \sum_{i=1}^{n} \gamma_{ik},$$

which represents the weighted count of points assigned to each cluster. The mixing proportions are updated according to

$$\pi_k = \frac{N_k}{n},$$

and the means are updated as the weighted average

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik}\, x_i.$$

Similarly, the covariance matrices are updated to reflect the weighted covariance of the data:

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{n} \gamma_{ik}\, (x_i - \mu_k)(x_i - \mu_k)^T.$$

To avoid singular matrices, we add a small constant $\epsilon = 10^{-6}$ to the diagonal of each covariance matrix. We added this part after testing with some images. In a specific image that had large areas of the same exact color, at some point, the algorithm was giving a log-likelihood of $-\inf$ and crashing. We concluded that this was happening because as the algorithm converged, the pixels on the clusters were almost equal, giving covariance matrix close to 0.

After these updates, we recompute the density values using the updated parameters and calculate the new log-likelihood. The algorithm repeats this process until the absolute change in the log-likelihood falls below the specified tolerance tol, incrementing the iteration counter at each step and printing progress if `verbose` is enabled.

Once convergence is achieved, the function returns a list containing the total number of iterations, the final mixing proportions, means, and covariance matrices, as well as the final responsibilities. In addition, it returns the histories of γ, `mu` , and `sigma` from the first eight iterations, which are essential for later visualization of how the algorithm converged. This implementation allows us to monitor both the point-wise responsibilities and the evolution of the model parameters over the first few iterations, providing valuable insights into the behavior and convergence of our EM algorithm.

4. Visualizations

In this section we will expose different graphs to show how our algorithm behaves. To do so, we will visualize the data before, during and after the processing.

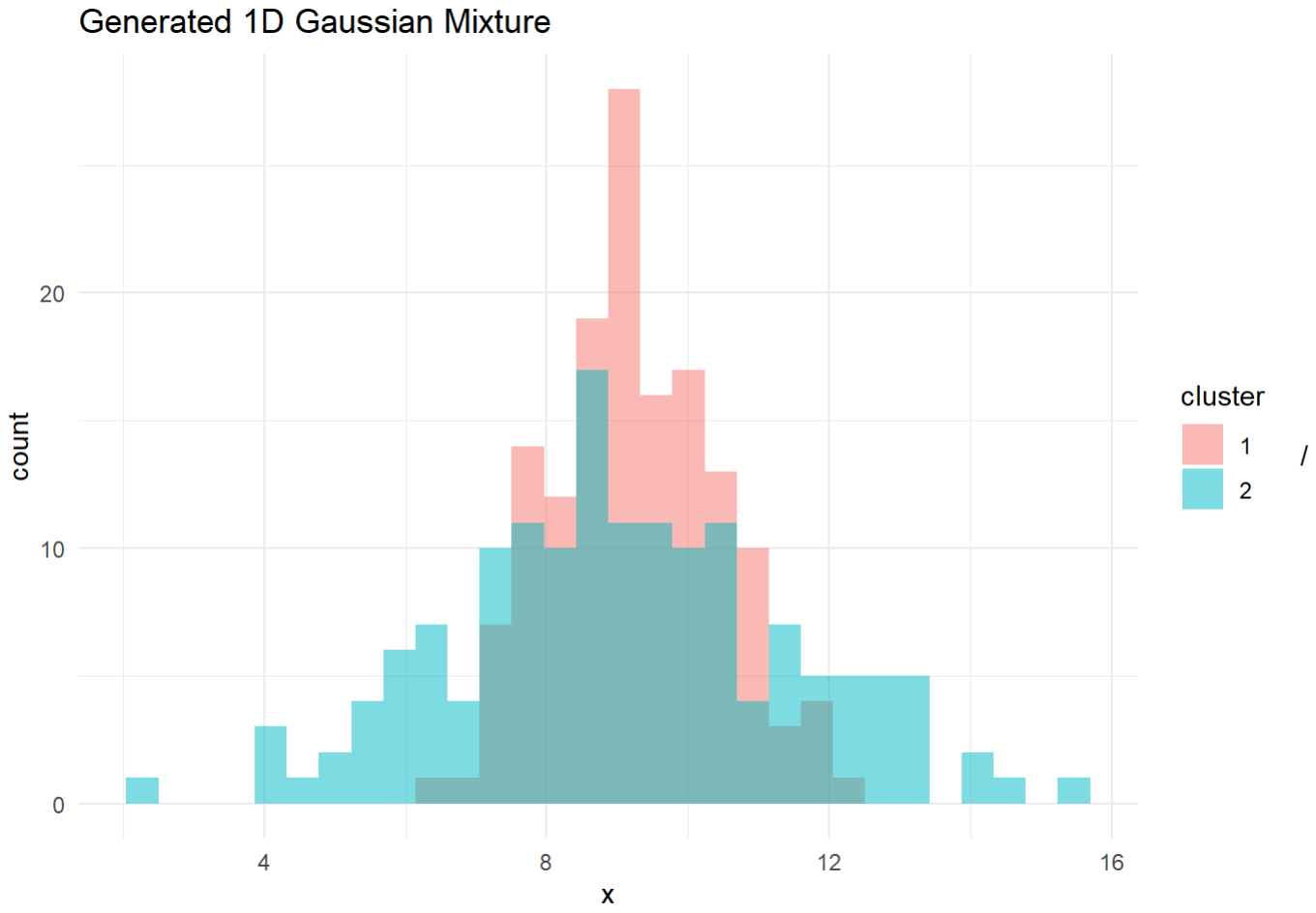
4.1. 1 Dimension Visualizations

We generated a 1D Gaussian mixture with two clusters, stored the data and true labels in a data frame, and then used `ggplot` to plot a histogram colored by the cluster assignments.

```r
# 1) Generate 1D data (K=2, n=300, etc.)
d <- 1
K <- 2
n <- 300
set.seed(563)
data_info_1d <- generate_gaussian_mixture_data(n, d, K)
data_1d <- data_info_1d$data  # shape: (n x 1)
true_labels_1d <- data_info_1d$z

df_1d <- data.frame(
  x       = data_1d[,1],
  cluster = factor(true_labels_1d)
)

# Plot the histogram of the generated data (colored by the true cluster)
ggplot(df_1d, aes(x, fill=cluster)) +
  geom_histogram(alpha=0.5, position="identity", bins=30) +
  theme_minimal() +
  ggtitle("Generated 1D Gaussian Mixture")
```



The histogram displays the distribution of the generated 1D data, with each bar shaded according to which of the two true clusters each observation belongs.

```r
# 2) Initialize parameters for EM
init_1d <- initialize_values(data_1d, nrow(data_1d), d, K)

# 3) Run EM algorithm
res_1d <- EM_algorithm(
  data   = data_1d,
  pi     = init_1d$Pis,
  mu     = init_1d$Mus,
  sigma  = init_1d$Sigmas,
  tol    = 1e-4
)

cat("1D EM converged in", res_1d$n_iterations, "iterations.\n")
```

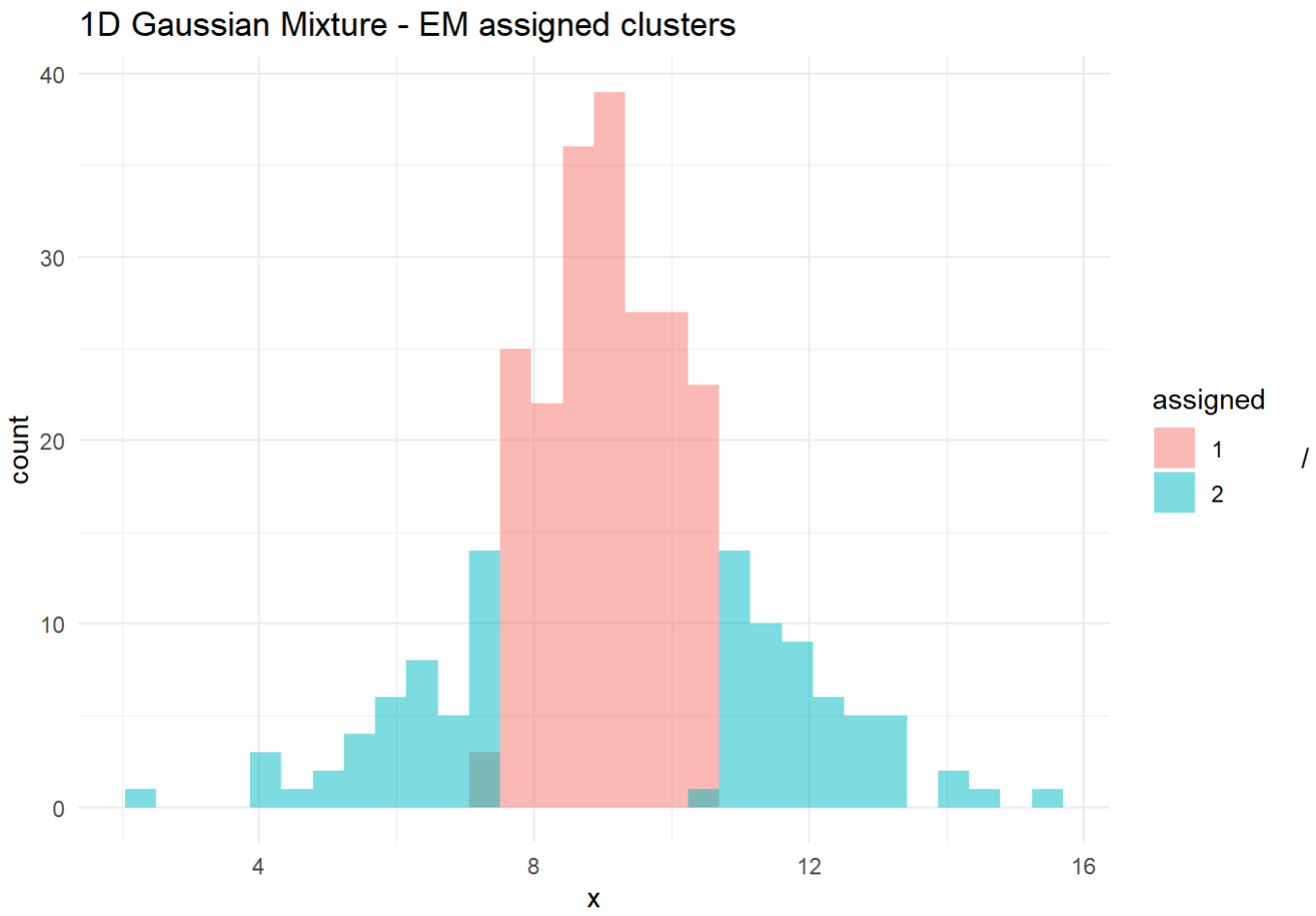
```
## 1D EM converged in 55 iterations.
```

We coded this part to determine each data point's final cluster assignment from the EM algorithm and optionally correct label switching. First, we retrieve the responsibilities in `gamma_1d` and assign each point to the cluster with the highest probability via `which.max`. Next, we create a confusion matrix `cm` by comparing the true labels and the assigned labels. If the diagonal sum in `cm` (i.e., correctly matched labels) is less than the off-diagonal sum, we swap the labels to ensure consistency with our original cluster labeling. We then store these possibly corrected labels in `df_1d$assigned`.

```r
# 5) Assign each data point to the most probable cluster
gamma_1d <- res_1d$gamma
assigned_1d <- apply(gamma_1d, 1, which.max)

# create a confusion matrix to assignate the correct label to each class
cm <- table(true_labels_1d, assigned_1d)
if (cm[1,1] + cm[2,2] < cm[1,2] + cm[2,1]) {
  assigned_1d <- ifelse(assigned_1d == 1, 2, 1)
}
df_1d$assigned <- factor(assigned_1d)

# 6) Visualize final clusters
ggplot(df_1d, aes(x, fill=assigned)) +
  geom_histogram(alpha=0.5, position="identity", bins=30) +
  theme_minimal() +
  ggtitle("1D Gaussian Mixture - EM assigned clusters")
```



Compared to the previous histogram, which showed the true cluster labels, the new plot displays how the EM algorithm actually clustered the data points. This final histogram may look similar overall, but not every observation was well classified (the red tails weren't captured by the algorithm). this is due to working in one dimension, causing the overlapping of the Gaussian one way or another.

Visualizing the iterative progress in 1D adds little insight because all points lie on a single axis, making it hard to distinguish intermediate steps clearly.

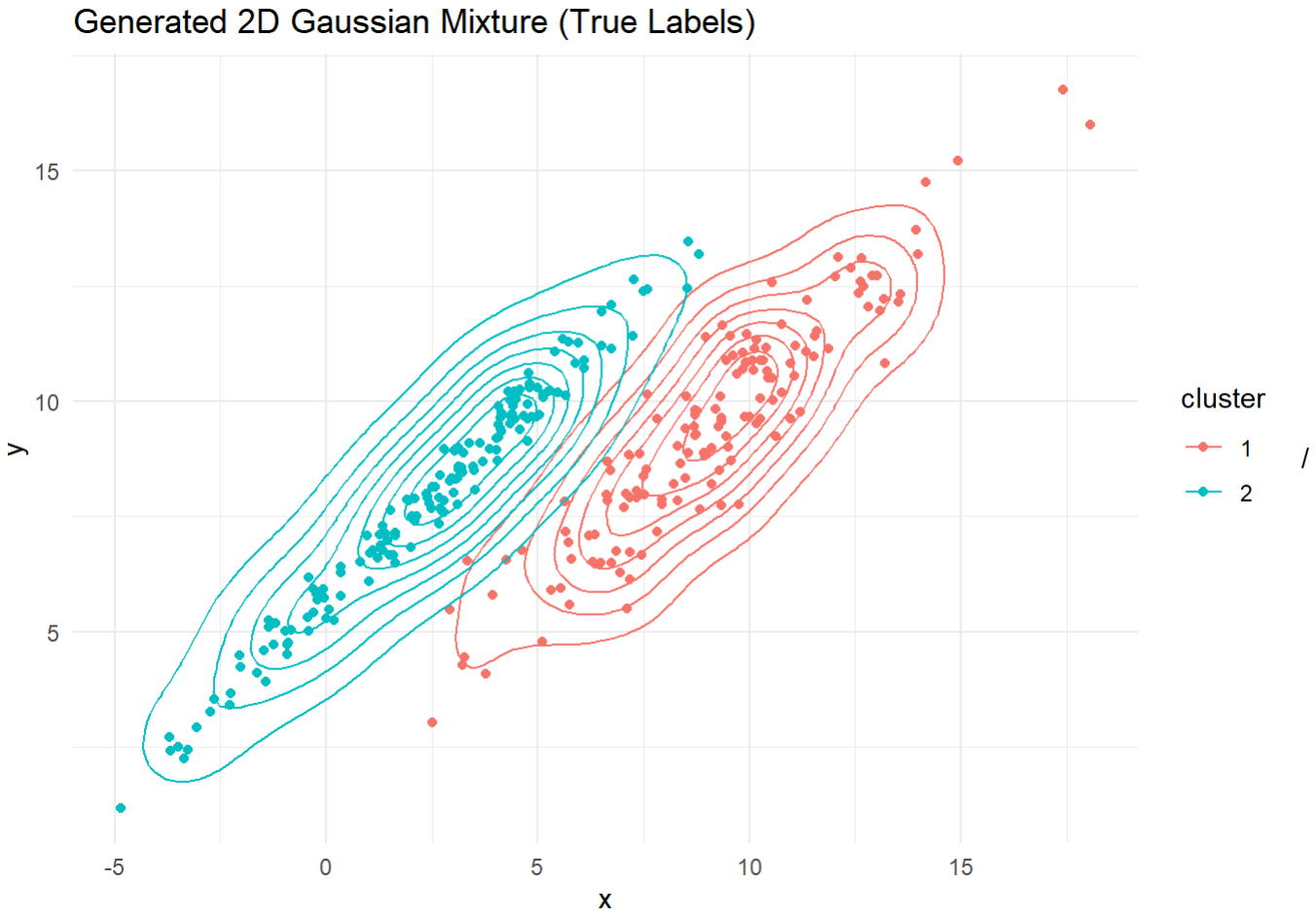
4.2. 2 Dimension Visualizations

We repeat the process for 2 dimensions.

```
# 1) Generate 2D data
d <- 2
K <- 2
n <- 300
set.seed(563)
data_info_2d <- generate_gaussian_mixture_data(n, d, K)
data_2d <- data_info_2d$data    # shape: (n x 2)
true_labels_2d <- data_info_2d$z

df_2d <- data.frame(
  x = data_2d[,1],
  y = data_2d[,2],
  cluster = factor(true_labels_2d)
)

# Plot the generated data (before EM)
ggplot(df_2d, aes(x, y, color = cluster)) +
  geom_point() +
  geom_density_2d() +
  theme_minimal() +
  ggtitle("Generated 2D Gaussian Mixture (True Labels)")
```



We visualize the data with scatter points plus density contours for each cluster.

```
# 2) Initialize parameters
init_2d <- initialize_values(data_2d, nrow(data_2d), d, K)

# 3) Run EM Algorithm
res_2d <- EM_algorithm(data_2d, init_2d$Pis, init_2d$Mus, init_2d$Sigmas, tol = 1e-4)
cat("2D EM converged in", res_2d$n_iterations, "iterations.\n")
```

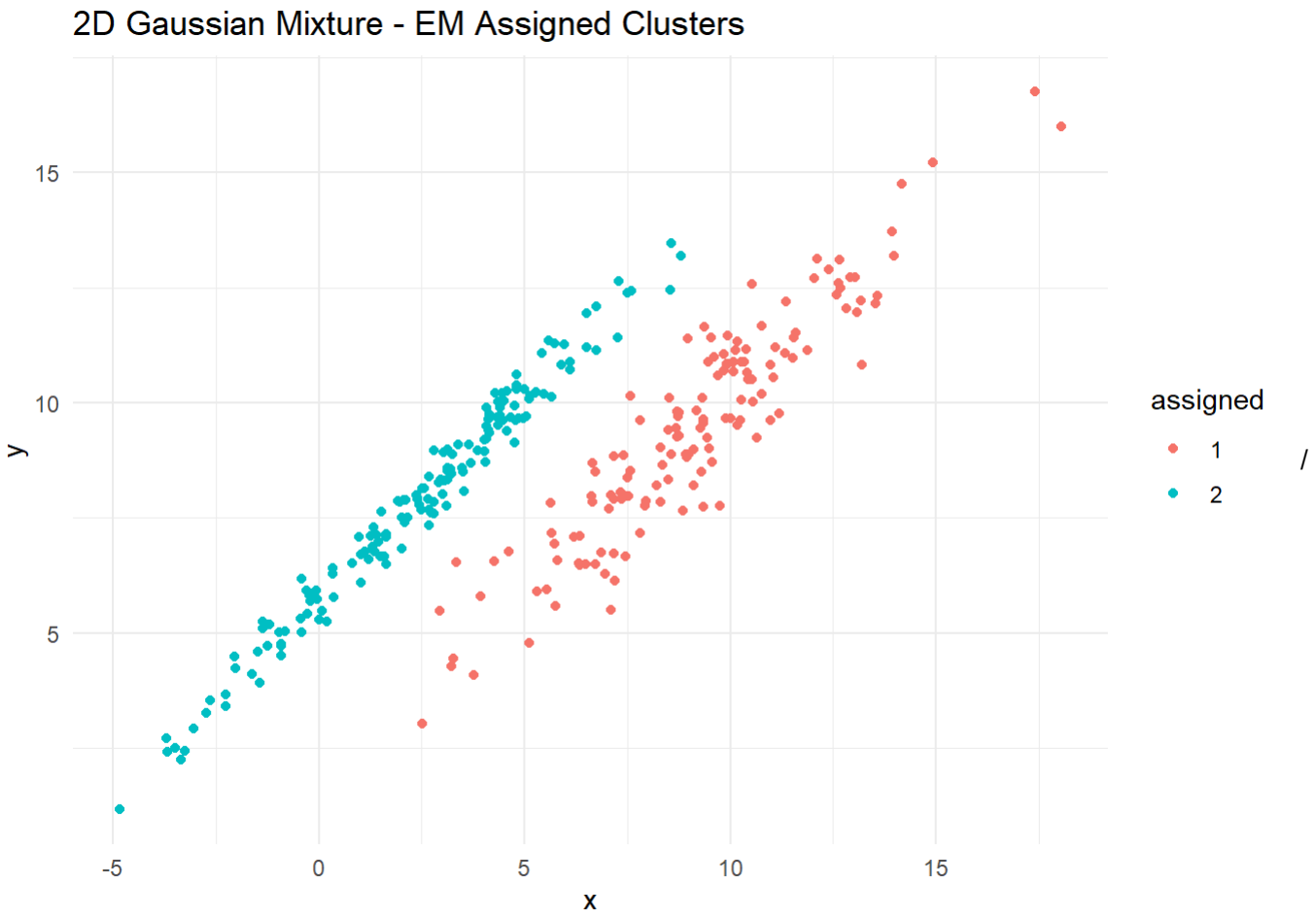
```
## 2D EM converged in 8 iterations.
```

This code follows the same idea as with 1 dimension.

```
# 4) Final cluster assignment
gamma_2d <- res_2d$gamma
assigned_2d <- apply(gamma_2d, 1, which.max)

# Create confusion matrix to re-label if necessary
cm <- table(true_labels_2d, assigned_2d)
if (cm[1,1] + cm[2,2] < cm[1,2] + cm[2,1]) {
  assigned_2d <- ifelse(assigned_2d == 1, 2, 1)
}
df_2d$assigned <- factor(assigned_2d)

# Plot the final clustering (after EM)
ggplot(df_2d, aes(x, y, color = assigned)) +
  geom_point() +
  theme_minimal() +
  ggtitle("2D Gaussian Mixture - EM Assigned Clusters")
```



In this case, it seems that the classification has worked perfectly. this is due to working in one more dimension, being able to capture better where is each point located.

We coded a `for` loop that iterates over each element in `res_2d$gamma_history`, which stores the responsibilities (posterior probabilities) from the first eight iterations of the EM algorithm. Inside the loop, we create a data frame containing the original coordinates (`x` and `y`) along with `prob_cluster1`, which is the probability that each point belongs to cluster 1 in the current iteration. We then build a scatter plot with `ggplot`, mapping `prob_cluster1` to a blue-to-red color scale, and store each plot in the list `plots_gamma`.

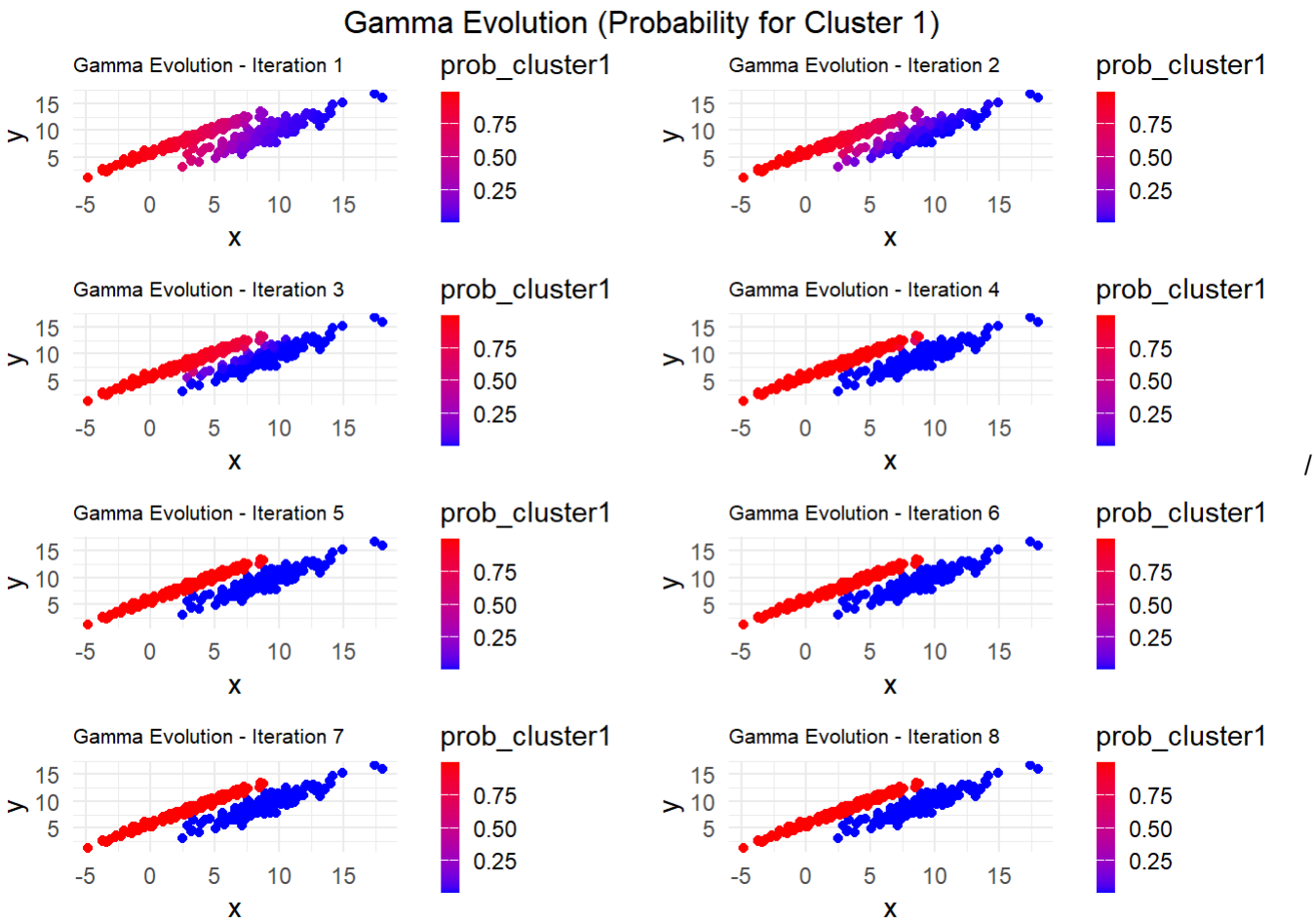
```
# 5) Gamma Evolution (During EM)

plots_gamma <- list()
for (i in 1:length(res_2d$gamma_history)) {
  gamma_iter <- res_2d$gamma_history[[i]]
  df_gamma <- data.frame(
    x = data_2d[,1],
    y = data_2d[,2],
    prob_cluster1 = gamma_iter[,1]
  )

  p_gamma <- ggplot(df_gamma, aes(x = x, y = y, color = prob_cluster1)) +
    geom_point() +
    scale_color_gradient(low = "blue", high = "red") +
    # Here's the minimal addition to shrink the colorbar
    guides(color = guide_colorbar(barwidth = 0.5, barheight = 3.5)) +
    labs(title = paste("Gamma Evolution - Iteration", i)) +
    theme_minimal() +
    theme(plot.title = element_text(size = 8))  # Reduce iteration title size

  plots_gamma[[i]] <- p_gamma
}

grid.arrange(
  grobs = plots_gamma,
  ncol = 2,
  top = textGrob(
    "Gamma Evolution (Probability for Cluster 1)",
    gp = gpar(fontsize = 12)
  )
)
```



When we arrange these plots, we see how the probability of belonging to cluster 1 evolves across the iterations. Points closer to cluster 1 gradually shift to a higher probability (red), while those closer to cluster 2 appear in blue.

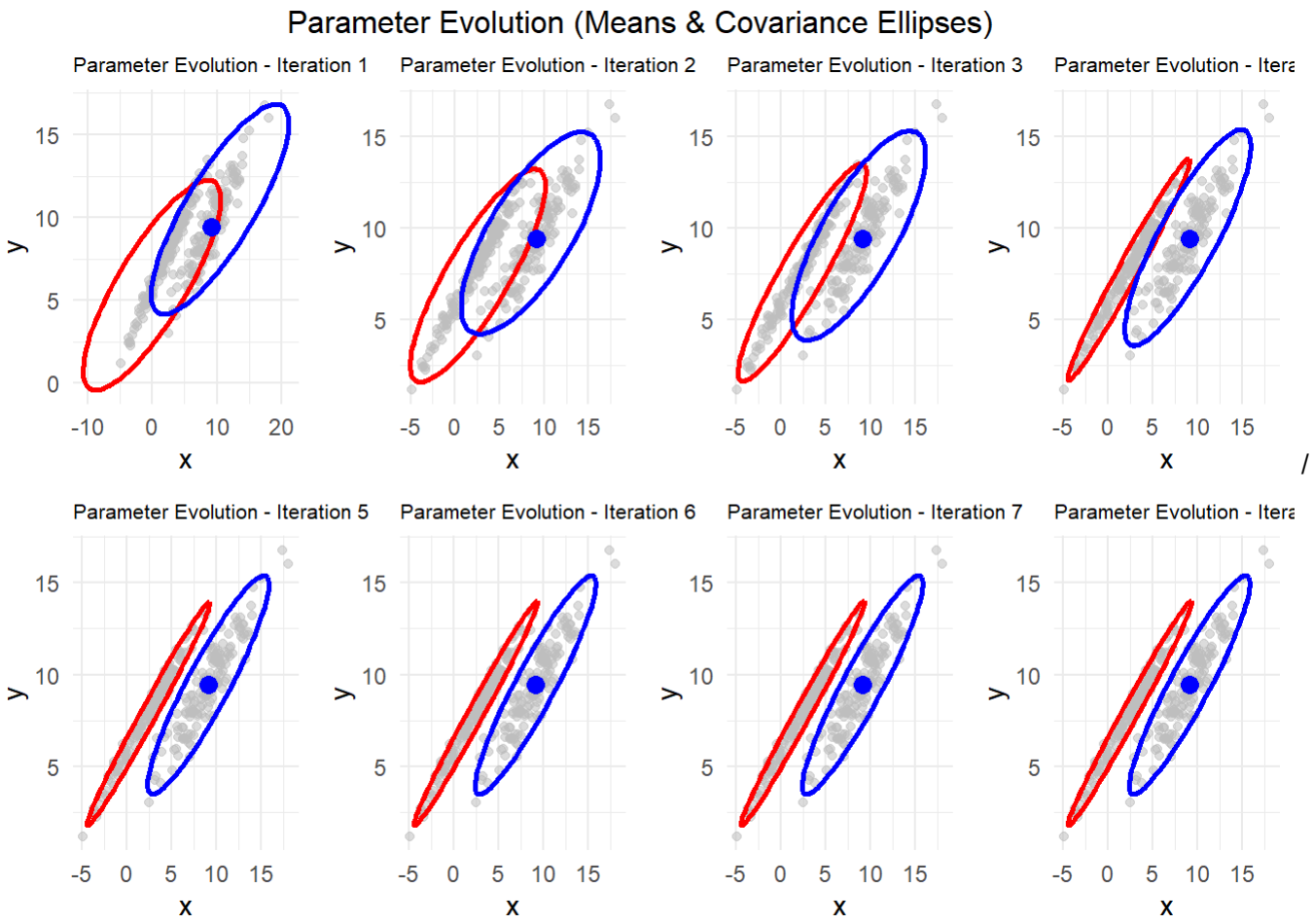
```r
# 6) Parameter Evolution (During EM)

plots_params <- list()
for (i in 1:length(res_2d$mu_history)) {
  current_mu <- res_2d$mu_history[[i]]
  current_sigma <- res_2d$sigma_history[[i]]

  p_param <- ggplot(df_2d, aes(x = x, y = y)) +
    geom_point(alpha = 0.5, color = "gray") +
    labs(title = paste("Parameter Evolution - Iteration", i)) +
    theme_minimal() +
    theme(plot.title = element_text(size = 8))

  for (k in 1:K) {
    ellipse_points <- as.data.frame(
      ellipse(current_sigma[[k]], centre = current_mu[k,], level = 0.95)
    )
    p_param <- p_param +
      geom_path(data = ellipse_points, aes(x = x, y = y),
                color = ifelse(k == 1, "red", "blue"), size = 1) +
      geom_point(aes(x = current_mu[k, 1], y = current_mu[k, 2]),
                 color = ifelse(k == 1, "red", "blue"), size = 3)
  }
  plots_params[[i]] <- p_param
}

grid.arrange(grobs = plots_params, ncol = 4,
  top = textGrob("Parameter Evolution (Means & Covariance Ellipses)",
                 gp = gpar(fontsize = 12))
)
```



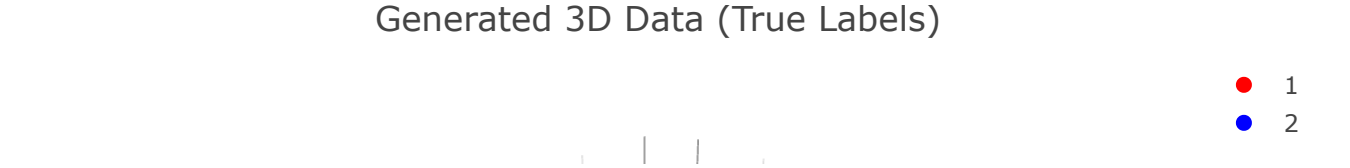
These subplots illustrate how the EM algorithm's estimated means and 95% confidence ellipses (in red and blue) evolve over the first eight iterations. Early on, the ellipses may overlap or be misplaced, but as the iterations proceed, each cluster's mean and covariance adapt to the data, converging to a stable configuration that better captures the underlying distribution.

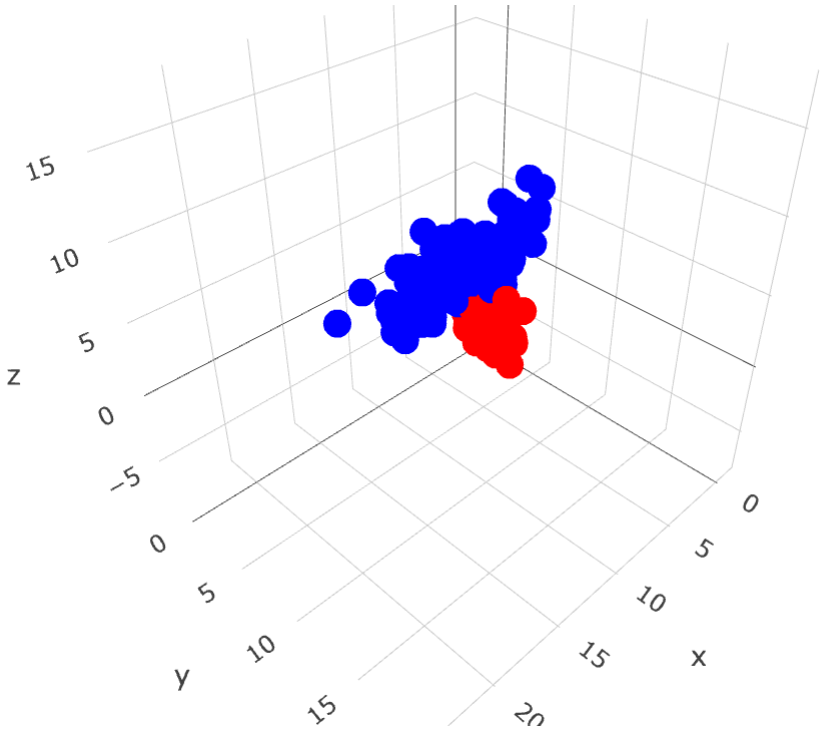
4.3. Dimension Visualizations

Once again we repeat the process for 3 dimensions.

```
# 1) 3D Data Generation
set.seed(563)
d <- 3
K <- 2
n <- 300
data_info_3d <- generate_gaussian_mixture_data(n, d, K)
data_3d <- data_info_3d$data    # (n x 3)
true_labels_3d <- data_info_3d$z
df_3d <- data.frame(x = data_3d[,1], y = data_3d[,2], z = data_3d[,3],
                    cluster = factor(true_labels_3d))

# 2) Before EM
p_before <- plot_ly(df_3d, x = ~x, y = ~y, z = ~z, color = ~cluster,
                    colors = c("red", "blue"), type = "scatter3d", mode = "markers") %>%
  layout(title = "Generated 3D Data (True Labels)")
p_before
```



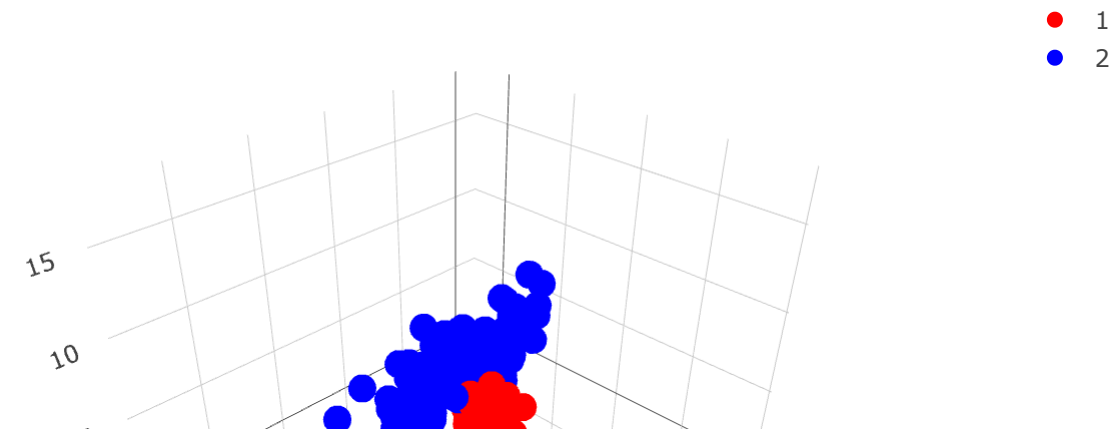


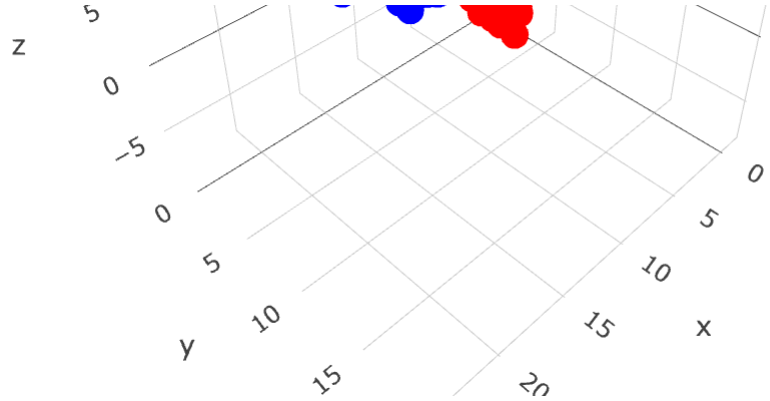
The output shows once again the original distributions, but this time using a 3 dimension.

```
# 3) Run EM Algorithm
init_3d <- initialize_values(data_3d, nrow(data_3d), d, K)
res_3d <- EM_algorithm(data_3d, init_3d$Pis, init_3d$Mus, init_3d$Sigmas, tol = 1e-4, verbose
= FALSE)
cat("2D EM converged in", res_3d$n_iterations, "iterations.\n")
```

```
## 2D EM converged in 20 iterations.
```

```
# 4) Final Clustering: After EM
gamma_3d <- res_3d$gamma
assigned_3d <- apply(gamma_3d, 1, which.max)
df_3d$assigned <- factor(assigned_3d)
p_after <- plot_ly(df_3d, x = ~x, y = ~y, z = ~z, color = ~assigned,
                   colors = c("red", "blue"), type = "scatter3d", mode = "markers") %>%
  layout(title = "Final Clustering after EM (3D)")
p_after
```

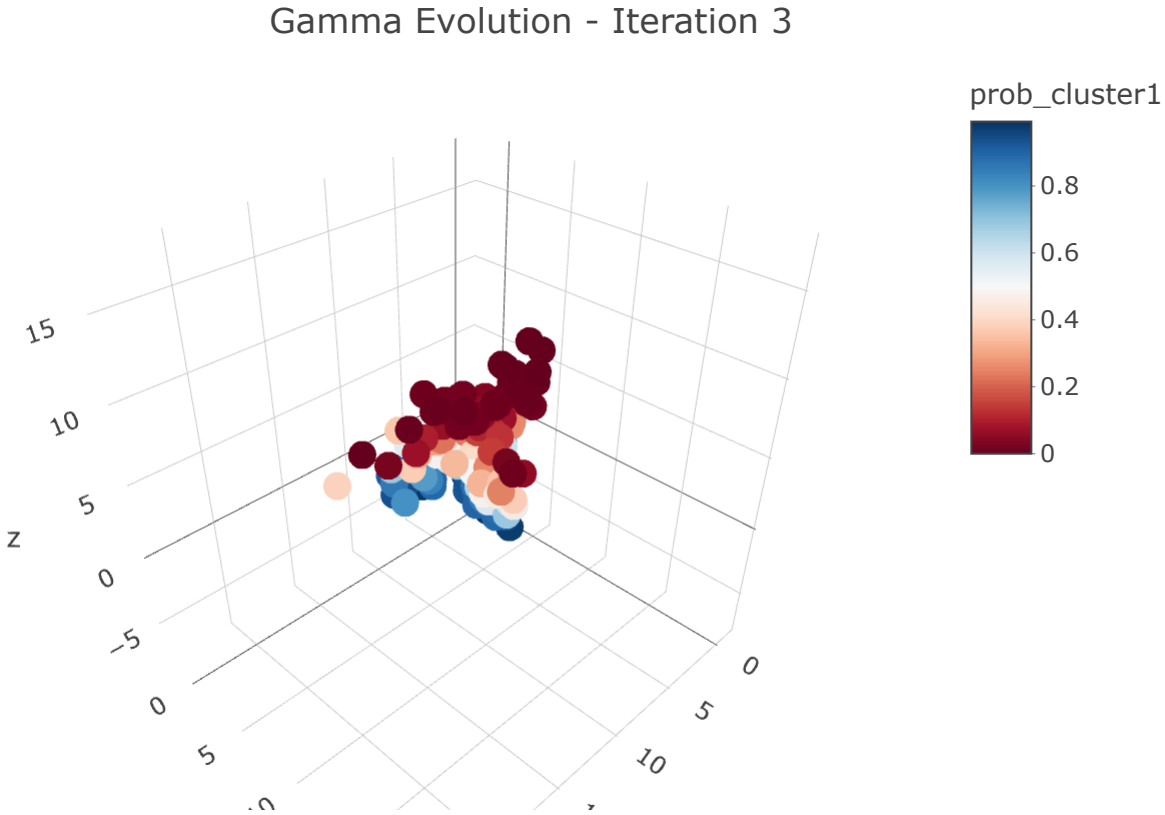




The algorithm clearly classifies the data correctly, assigning each observation to the correct class.

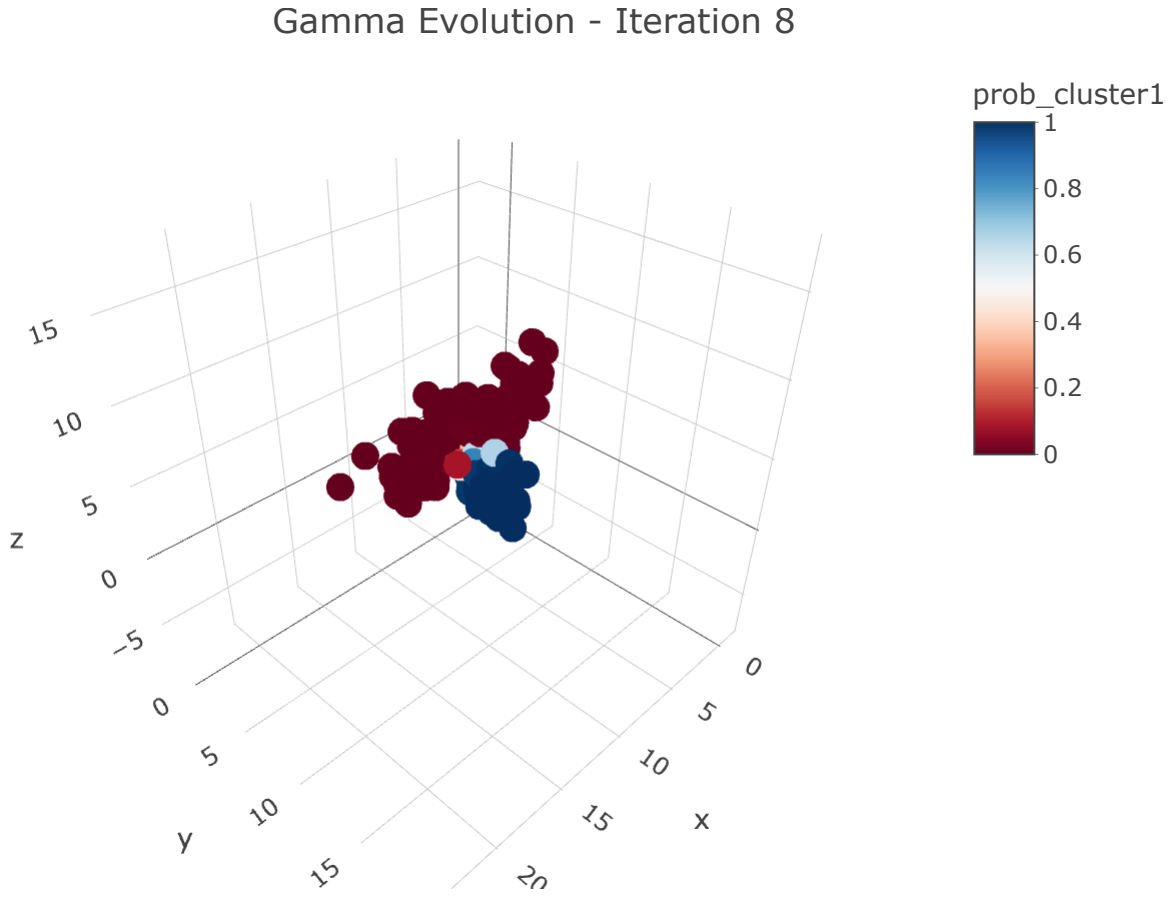
As before, we visualize the gamma evolution in the first 8 iterations. In this case, as we are working with the library `plotly`, each iteration is saved in a separated graph, so we are going to visualize only iterations **3 and 8** for summing up purposes.

```
# 5) Gamma Evolution Visualization (During EM)
gamma_plots_3d <- list()
for (i in 1:length(res_3d$gamma_history)) {
  gamma_iter <- res_3d$gamma_history[[i]]
  df_gamma <- data.frame(x = data_3d[,1], y = data_3d[,2], z = data_3d[,3],
                         prob_cluster1 = gamma_iter[,1])
  p_gamma <- plot_ly(df_gamma, x = ~x, y = ~y, z = ~z,
                     color = ~prob_cluster1, colors = "RdBu", type = "scatter3d", mode = "mar
kers") %>%
    layout(title = paste("Gamma Evolution - Iteration", i))
  gamma_plots_3d[[i]] <- p_gamma
}
# Display the gamma evolution plots separately
gamma_plots_3d[[3]]
```



In the third iteration, most of the points have been classified already, but classifying the intermediate ones is the convoluted part. Let's see how does it evolve.

```
gamma_plots_3d[[8]]
```



Now, at iteration 8, even if there are left 12 of them, we can appreciate clear improvement in the classification, with just few points still in that classification "limbo".

5. Images

In this section, we apply our EM-based approach to real photographs, allowing us to cluster pixels into K segments. By transforming each image into a matrix of color values, running the EM algorithm, and then mapping each pixel back to its assigned cluster, we can visualize the original and segmented images side by side. This provides an intuitive understanding of how well the algorithm separates different regions or objects in a picture.

```
segment_image_with_em <- function(image_path, K, tol = 10^-2, rgb = TRUE) {

  # Load Image
  img <- readImage(image_path)

  # Convert image into a matrix of pixel values
  img_matrix <- apply(img, 3, as.vector)  # Convert each color channel to a vector
  img_matrix <- matrix(img_matrix, ncol = 3, byrow = FALSE)  # Ensure correct format

  # Initialize values for EM algorithm
  init_values <- initialize_values(img_matrix, nrow(img_matrix), 3, K)

  # Run EM Algorithm
  results <- EM_algorithm(img_matrix, init_values$Pis, init_values$Mus, init_values$Sigmas, t
ol = tol)

  # Assign each pixel to a cluster based on gamma
  segmented <- apply(results$gamma, 1, which.max)

  # Reshape segmented data back into an image format
  segmented_img <- matrix(segmented, nrow = dim(img)[1], ncol = dim(img)[2])

  # Convert segmented image into an RGB format or normalize grayscale
  if (rgb) {
    segmented_img <- convert_to_rgb(K, segmented_img)
  } else {
    segmented_img <- (segmented_img - min(segmented_img)) / (max(segmented_img) - min(segment
ed_img))
  }

  # Convert images to ggplot objects
  original_plot <- ggplot() +
    annotation_custom(rasterGrob(img), xmin=-Inf, xmax=Inf, ymin=-Inf, ymax=Inf) +
    ggtitle("Original Image") + theme_void()

  segmented_plot <- ggplot() +
    annotation_custom(rasterGrob(segmented_img), xmin=-Inf, xmax=Inf, ymin=-Inf, ymax=Inf) +
    ggtitle("Segmented Image") + theme_void()

  # Arrange both images side by side
  grid.arrange(original_plot, segmented_plot, ncol = 2)
}
```

We coded the function `segment_image_with_em` to unify the entire process of loading, segmenting, and displaying an image. First, we read the photograph from `image_path` using `readImage`, then convert its three color channels into a single matrix `img_matrix`, where each row corresponds to one pixel and each column corresponds to one of the red, green, or blue channels.

Next, we call `initialize_values` to produce initial parameters for the EM algorithm—namely the mixing proportions, means, and covariance matrices. We then pass these parameters, along with the pixel matrix, to `EM_algorithm` to cluster the pixels into K groups based on their color values. Once the EM algorithm finishes, we reshape the cluster labels into a matrix `segmented_img` that matches the original image dimensions.

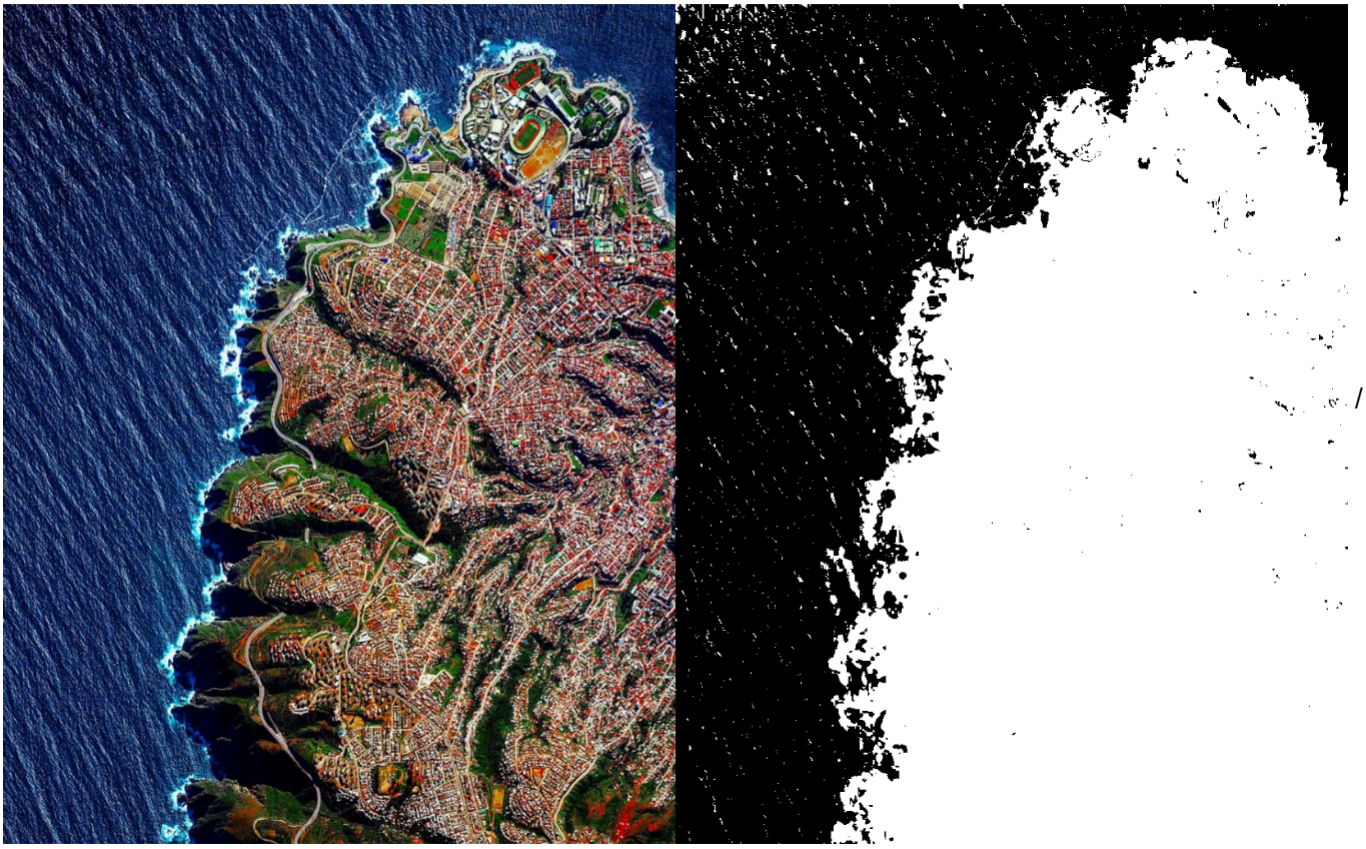
If the `rgb` argument is `TRUE`, we map each cluster label to a distinct color using `convert_to_rgb`; otherwise, we normalize the labeled image for grayscale display. Finally, we construct two `ggplot` objects: `original_plot` to show the unaltered image and `segmented_plot` to show the clustered version. By arranging these two plots side by side, we can easily compare the original photograph with its segmented output and observe how the algorithm partitions the image into different regions.

5.1 Landscapes photographs

```
segment_image_with_em("mar.jpg", K = 2, tol = 1e-3, rgb = FALSE)
```

Original Image

Segmented Image

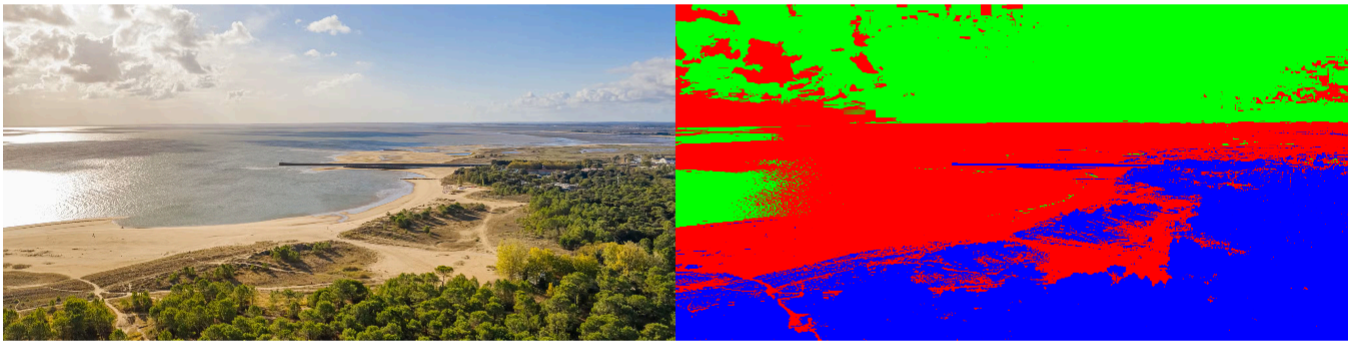


In the first image (`mar.jpg`), we set $K = 2$ and turned off `rgb` mode (`rgb = FALSE`), producing a black-and-white segmentation that cleanly separates the ocean from the landmass. With only two clusters, the algorithm highlights the major boundary between water and terrain, which is why we chose a simple grayscale display.

```
segment_image_with_em("landscape2.jpg", K = 3, tol = 10^-3)
```

Original Image

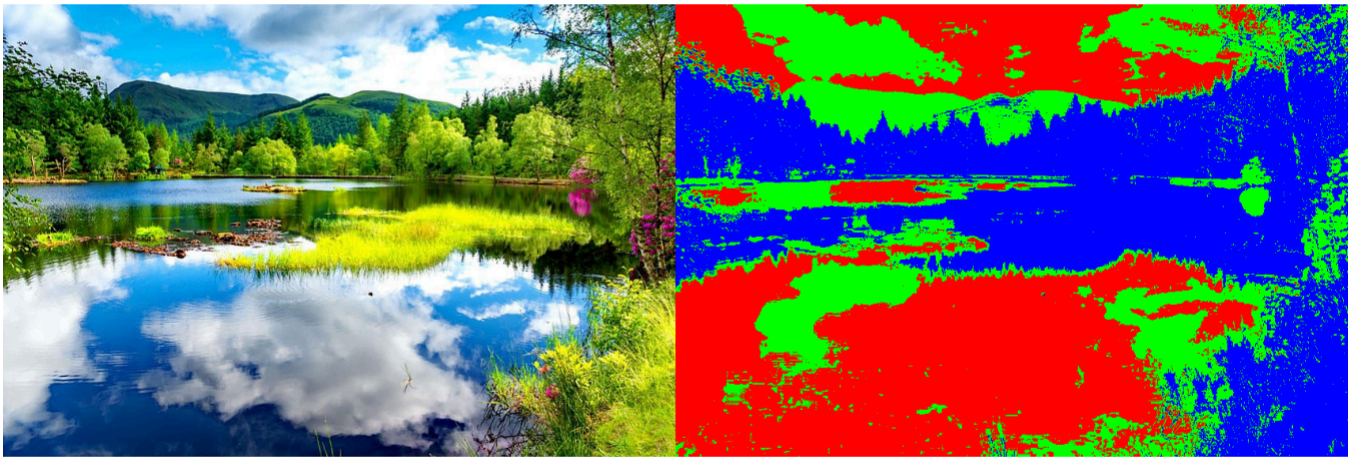
Segmented Image



```
segment_image_with_em("landscape3.jpg", K = 3, tol = 10^-3)
```

Original Image

Segmented Image



/

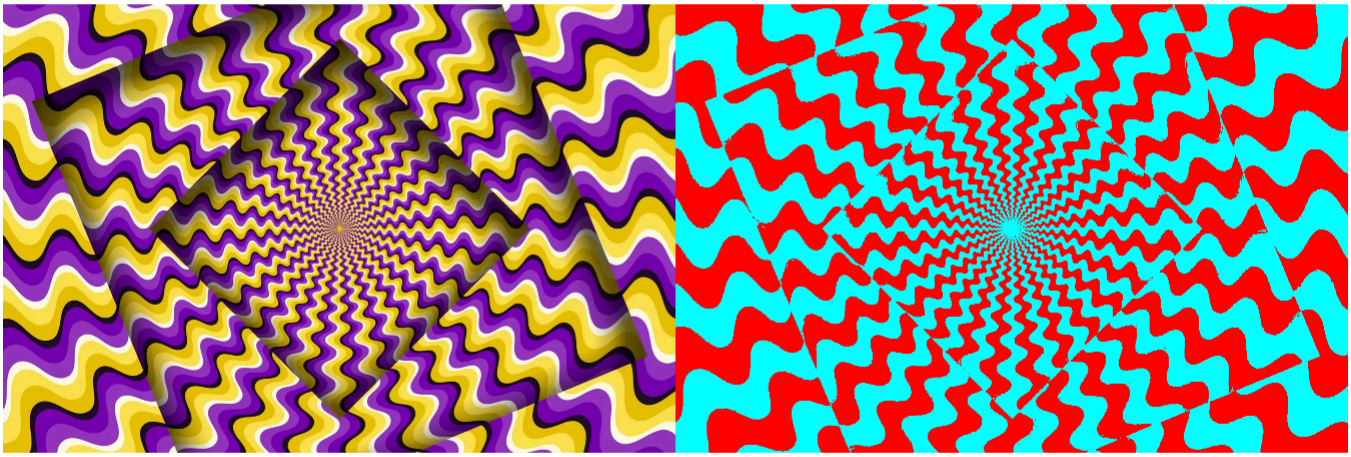
For the second (`landscape2.jpg`) and third (`landscape3.jpg`) images, we increased K to 3 and allowed `rgb = TRUE` . This choice provides enough clusters to capture different features like sky, water, vegetation, ground and clouds still keeping the segmentation relatively interpretable. As a result, each region is colored distinctly, revealing how the algorithm partitions more complex landscapes into three dominant color-based clusters. For obvious reasons, it is impossible for the algorithm to distinguish between sky and water. Also, in landscape number 2, the clouds are misinterpreted, probably due to the tone similarity with the ground.

5.2. Patterns

```
segment_image_with_em("ilusion.jpg", K = 2, tol = 10^-3)
```

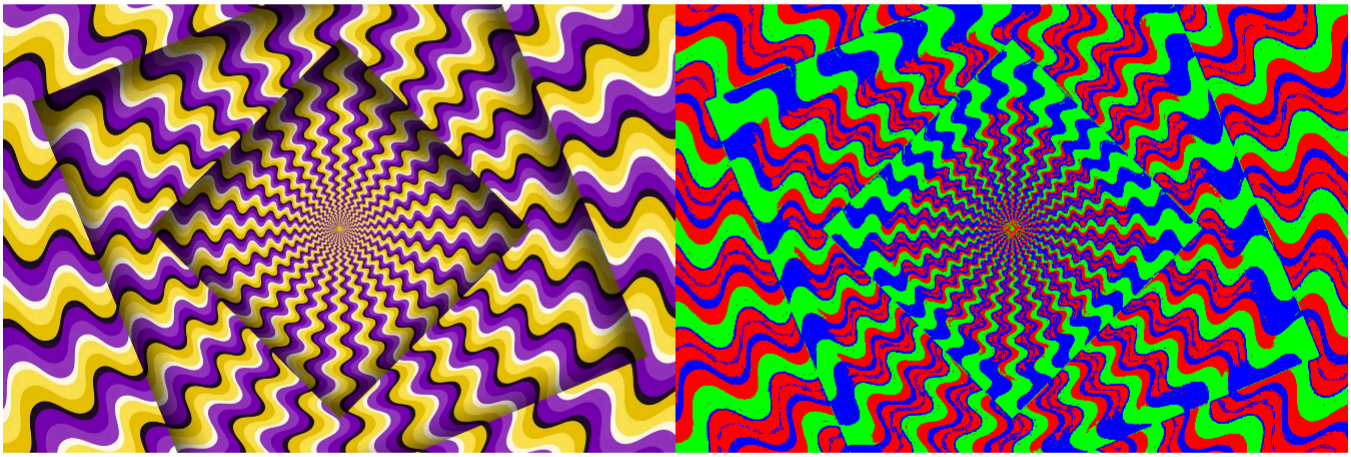
Original Image

Segmented Image



```
segment_image_with_em("ilusion.jpg", K = 3, tol = 10^-2)
```

Original Image Segmented Image

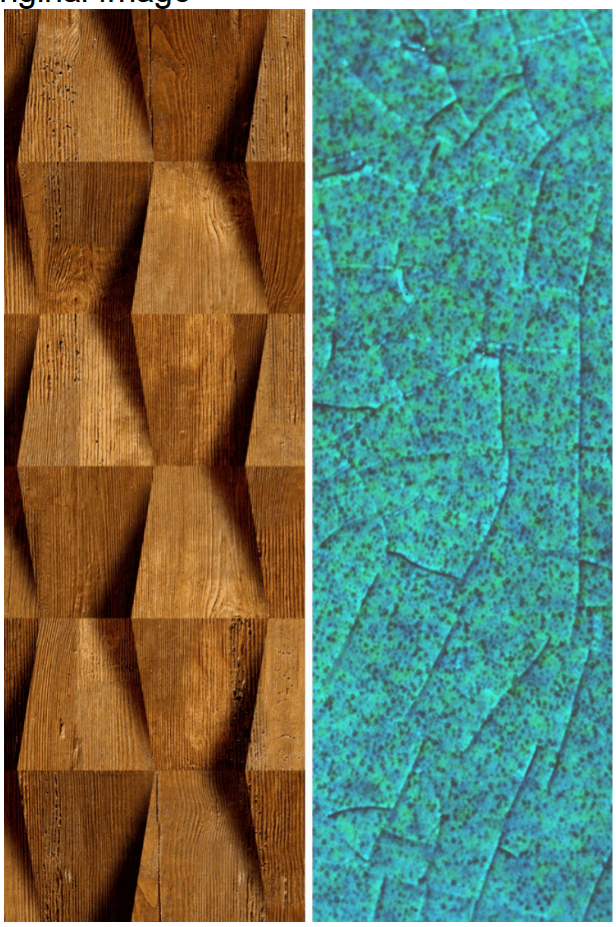


/

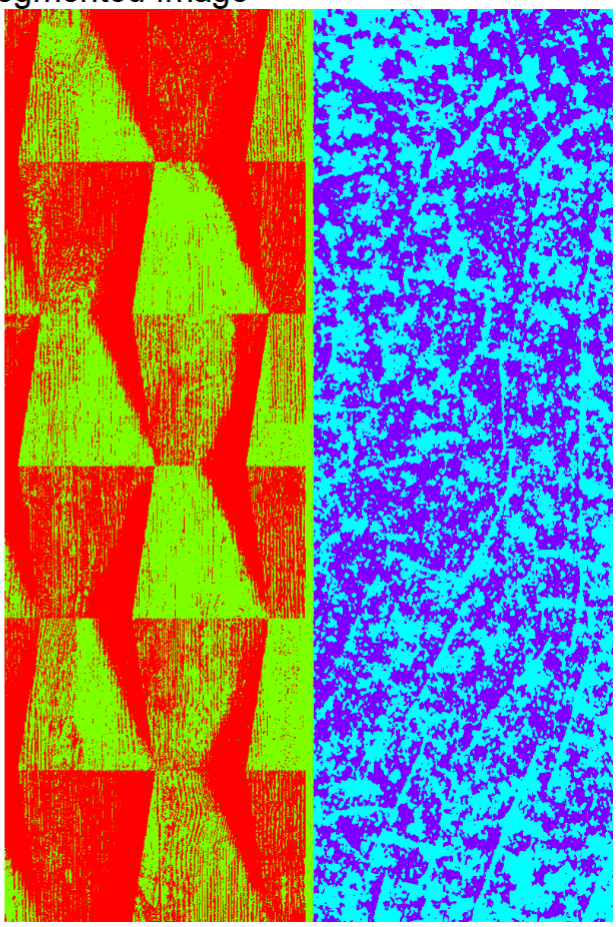
When we segment `ilusion.jpg` with $K = 2$, the algorithm divides the entire image into two broad color categories, which simplifies the spiral illusion to a binary color scheme. Increasing to $K = 3$ adds a third color cluster, so the spiral appears more intricate, capturing extra color transitions and producing a more nuanced segmentation. Essentially, a higher K allows the EM algorithm to partition the image into additional color-based clusters, revealing more of the illusion's structure.

```
segment_image_with_em("patrones.jpg", K = 4, tol = 1e-1)
```

Original Image



Segmented Image



/

Next, we segment `patrones.jpg` with $K = 4$ and a tolerance of 1×10^{-1}. Because we use four clusters, the algorithm can separate multiple repeating patterns more distinctly, highlighting different textures or color regions in red, green, blue, and yellow. The relatively high tolerance makes the EM algorithm converge quickly, possibly at the expense of fine-grained distinctions, but still sufficient to capture the main visual features in the patterned image.