lab_block_2

Aron, Sergey, Shahin

2025-11-14

Statement of Contribution

Assignment 1 was mainly contributed by Aron. Assignment 2 was mainly contributed by Sergey. Assignment 3 was mainly contributed by Shahin. Theory questions: Q1: Aron, Q2: Sergey, Q3: Shahin. Formatting the report was done by Sergey.

Assignment 1. ENSEMBLE METHODS

Simulating for 3 different conditions

The condition for Y is x1 < x2.

```
##          1     10    100
## mu  0.2066 0.1378 0.1121
## var 0.0030 0.0010 0.0008
## sd  0.0552 0.0311 0.0288
```

The condition for Y is x1 < 0.5.

```
##          1     10    100
## mu  0.0975 0.0161 0.0068
## var 0.0187 0.0007 0.0001
## sd  0.1367 0.0264 0.0087
```

The condition for Y is divide the domain into 4 equal squares, top right and bottom left is 1, otherwise 0.

```
##          1     10    100
## mu  0.2453 0.1203 0.0736
## var 0.0137 0.0028 0.0012
## sd  0.1170 0.0532 0.0347
```

What happens with the mean error rate when the number of trees in the random forest grows? Why?

For numerical Y, the bias drops when the number of bagged learners combined are increased. For this case we use majority voting. Say X shows wether the prediction from a tree is accurate, $X = 0, 1$, X is ber(p) and X_i is i.i.d for different trees. The sum of X_i is bin(n, p).

$$P(\hat{y} = Y = 1) = P(bin(n,p) > n/2) \approx P(N(np, np(1-p) > n/2) = \phi(\frac{\frac{n}{2} - np}{\sqrt{np(1-p)}})$$

Which scales as $\sqrt{n} \cdot c$. The limits of phi is 0 and 1 as it is a cdf. Meaning the model will become very confident in predictions (0 if p<0.5, 1 otherwise). So for a p = 0.51, the model would always predict 1 if n is large enough, for 1 tree the variance of $\hat{y}$ would be larger. This is why the error rate drops, out model becomes better at confidently predicting y for p close to 0.5. But the bias doesn't stay constant like for numerical Y, say Y = 1, for one tree $Bias = p - Y$, if p >0.5, our tree is better than a random guess, then the bias drops, if our tree is worse than a random guess, the bias actually increases!

The third dataset represents a slightly more complicated classification problem than the first one. Still, you should get better performance for it when using sufficient trees in the random forest. Explain why you get better performance.

This still holds for the same nodesize (12) on both.

```
## [1] "with the others nodesize"
```

```
##          1     10    100
## mu  0.1650 0.0981 0.0812
## var 0.0027 0.0005 0.0004
## sd  0.0522 0.0230 0.0205
```

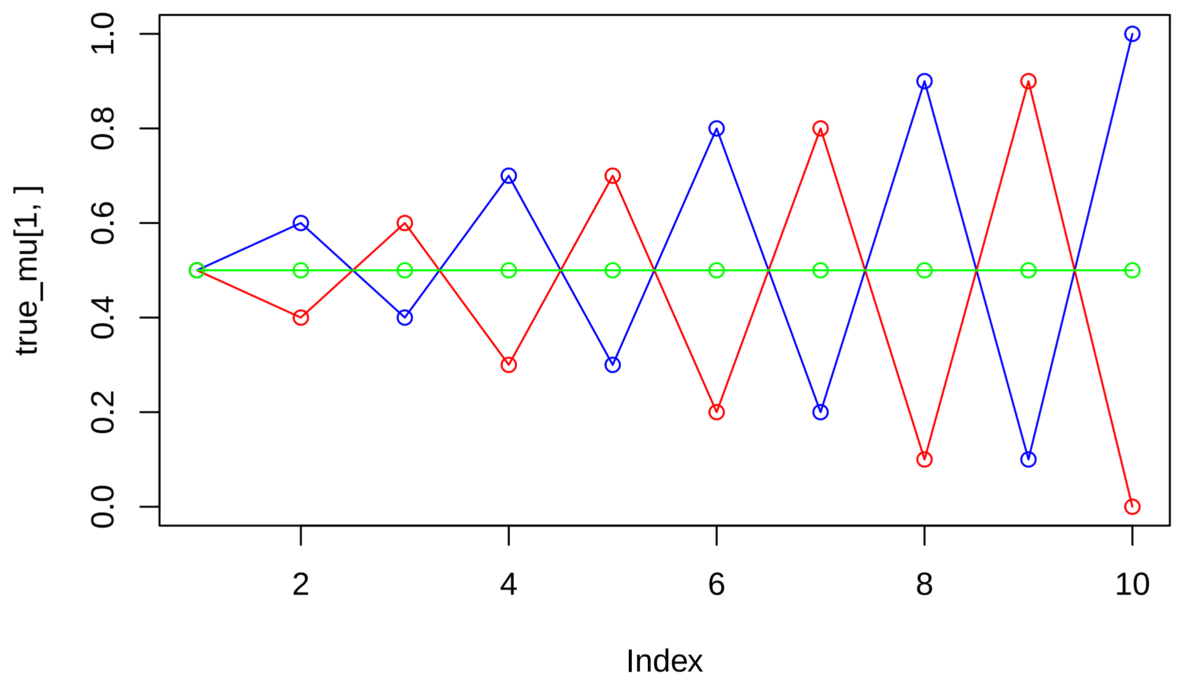
```
##          1     10    100
## mu  0.2453 0.1203 0.0736
## var 0.0137 0.0028 0.0012
## sd  0.1170 0.0532 0.0347
```

The reason for this is that lines parallell to the variable axis are drawn when a deciscion tree is trained, but in the first dataset the decision boundry is y=x, trees can still approximate this, but it requires more trees or depth, it will look like a stepwise function.

Assignment 2. MIXTURE MODELS

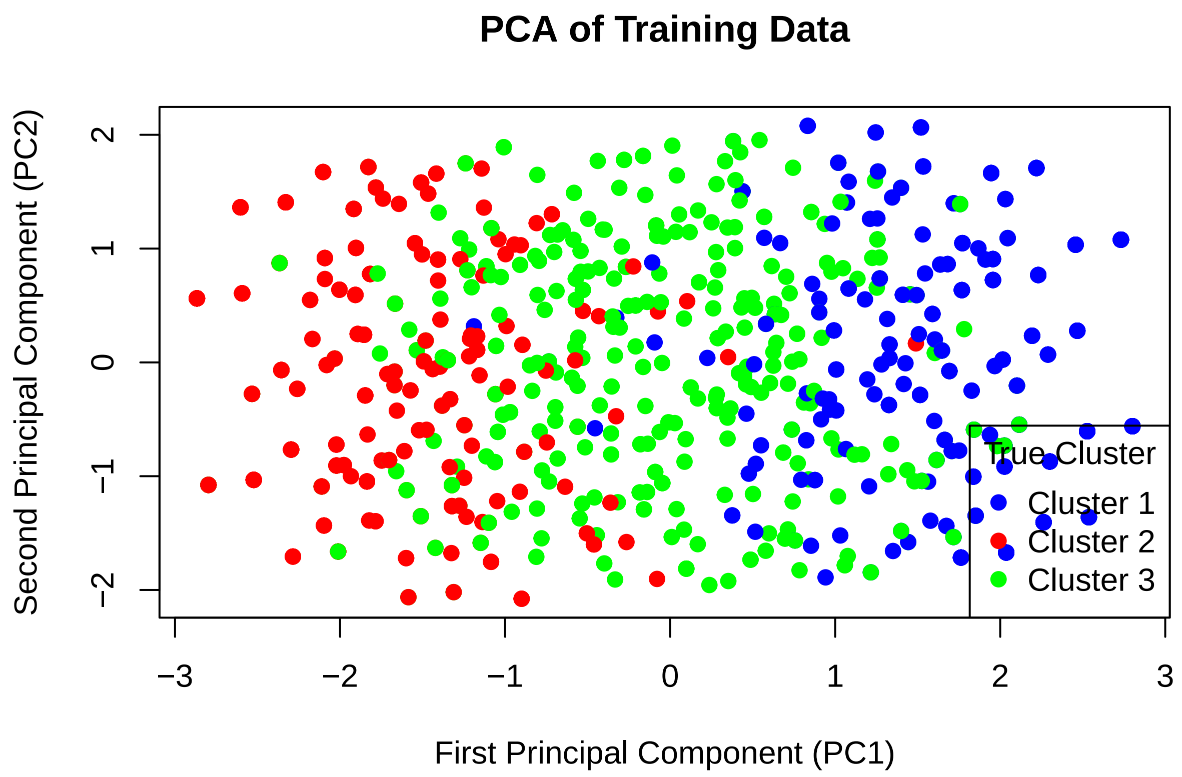
Task description

We are given three vectors of probabilities that defines the data-generating distribution for the cluster.



Every single data point is a vector consisting of 10 separate binary features. Each feature can only be a 0 or a 1, representing the absence or presence of an attribute. While a K-Means centroid is the average position of points in a cluster, the mu vector in a Bernoulli mixture model is the probability of each feature being "1" (or "yes") for a member of that cluster. The core difference is that Bernoulli centroids define the probability of features, while Gaussian centroids define the location of the cluster's center.

After generating the training data, We can compress the original 10 dimensions to 2 using PCA (Principal Component Analysis) to demonstrate that it indeed forms 3 distinct clusters:



Learning Bernoulli mixture model using EM algorithm

The EM algorithm includes the following steps:

Expectation (E) step

The goal of the **Expectation (E) step** is to calculate the "responsibilities" or weights for each data point. The weight w_{im} represents the posterior probability that a given data point $\boldsymbol{x}_i$ was generated by cluster m, using the current estimates of the model parameters (π and $\boldsymbol{\mu}$).

The formula for calculating the weight w_{im} for the i-th data point and the m-th cluster is:

$$w_{im} = \frac{\pi_m \prod_{d=1}^{D} \mu_{m,d}^{x_{i,d}} (1-\mu_{m,d})^{1-x_{i,d}}}{\sum_{k=1}^{M} \pi_k \prod_{d=1}^{D} \mu_{k,d}^{x_{i,d}} (1-\mu_{k,d})^{1-x_{i,d}}}$$

Where:

- w_{im} is the responsibility of cluster m for data point i.
- π_m is the current estimate of the mixing coefficient (prior probability) for cluster m.
- $\boldsymbol{\mu}_m = (\mu_{m,1}, ..., \mu_{m,D})$ is the vector of Bernoulli probabilities for cluster m.
- $\boldsymbol{x}_i = (x_{i,1}, ..., x_{i,D})$ is the i-th data point, which is a D-dimensional binary vector.
- The numerator is the joint probability of observing $\boldsymbol{x}_i$ from cluster m.
- The denominator is the sum of these joint probabilities over all possible clusters ($k = 1, ..., M$), which serves to normalize the weights for each data point so they sum to 1.

Calculating the Log-Likelihood

The log-likelihood is a critical measure used to assess how well the mixture model fits the entire dataset. In the EM algorithm, the objective is to maximize this value. A higher log-likelihood indicates that the current parameters make the observed data more probable.

For a dataset with n data points $\boldsymbol{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$, the total log-likelihood, denoted as $\mathcal{L}(\theta|\boldsymbol{X})$, is the sum of the log-probabilities of each individual data point:

$$\mathcal{L}(\theta|\boldsymbol{X}) = \sum_{i=1}^{n} \log\left(p(\boldsymbol{x}_i|\theta)\right)$$

The probability of a single data point, $p(\boldsymbol{x}_i|\theta)$, is the marginal probability obtained by summing over all possible clusters. For a Bernoulli mixture model, this is:

$$p(\boldsymbol{x}_i|\theta) = \sum_{m=1}^{M} \pi_m \left(\prod_{d=1}^{D} \mu_{m,d}^{x_{i,d}} (1 - \mu_{m,d})^{1-x_{i,d}} \right)$$

Combining these gives the full log-likelihood formula that is calculated at each iteration of the EM algorithm:

$$\mathcal{L}(\theta|\boldsymbol{X}) = \sum_{i=1}^{n} \log \left(\sum_{m=1}^{M} \pi_m \prod_{d=1}^{D} \mu_{m,d}^{x_{i,d}} (1 - \mu_{m,d})^{1-x_{i,d}} \right)$$

Where:

- θ represents all the model parameters, which includes all mixing coefficients π_m and all Bernoulli probability vectors $\boldsymbol{\mu}_m$.
- n is the total number of data points.
- M is the number of clusters.
- D is the dimensionality of the data.
- $x_{i,d}$ is the value of the d-th dimension for the i-th data point.

The M-Step Formulas

In the **Maximization (M) step**, the goal is to update the model parameters (π and $\boldsymbol{\mu}$) to new values that maximize the expected log-likelihood, using the responsibilities (w_{im}) calculated in the E-step. These update rules are derived by taking the derivative of the expected log-likelihood function with respect to each parameter and setting it to zero.

1. M-Step for Mixing Coefficients (π) The mixing coefficient π_m is updated to be the average responsibility that cluster m takes over all data points. It represents the new estimate for the proportion of data points belonging to cluster m.

$$\pi_m^{\text{new}} = \frac{1}{n} \sum_{i=1}^{n} w_{im}$$

Where:

- w_{im} is the responsibility of cluster m for data point i (from the E-step).
- n is the total number of data points.

2. M-Step for Conditional Probabilities (μ) The parameter $\mu_{m,d}$ (the probability of success for dimension d in cluster m) is updated to be the weighted average of the values of the d-th dimension across all data points. The weight for each data point $x_{i,d}$ is its responsibility w_{im} for that cluster.

$$\mu_{m,d}^{\text{new}} = \frac{\sum_{i=1}^{n} w_{im} x_{i,d}}{\sum_{i=1}^{n} w_{im}}$$

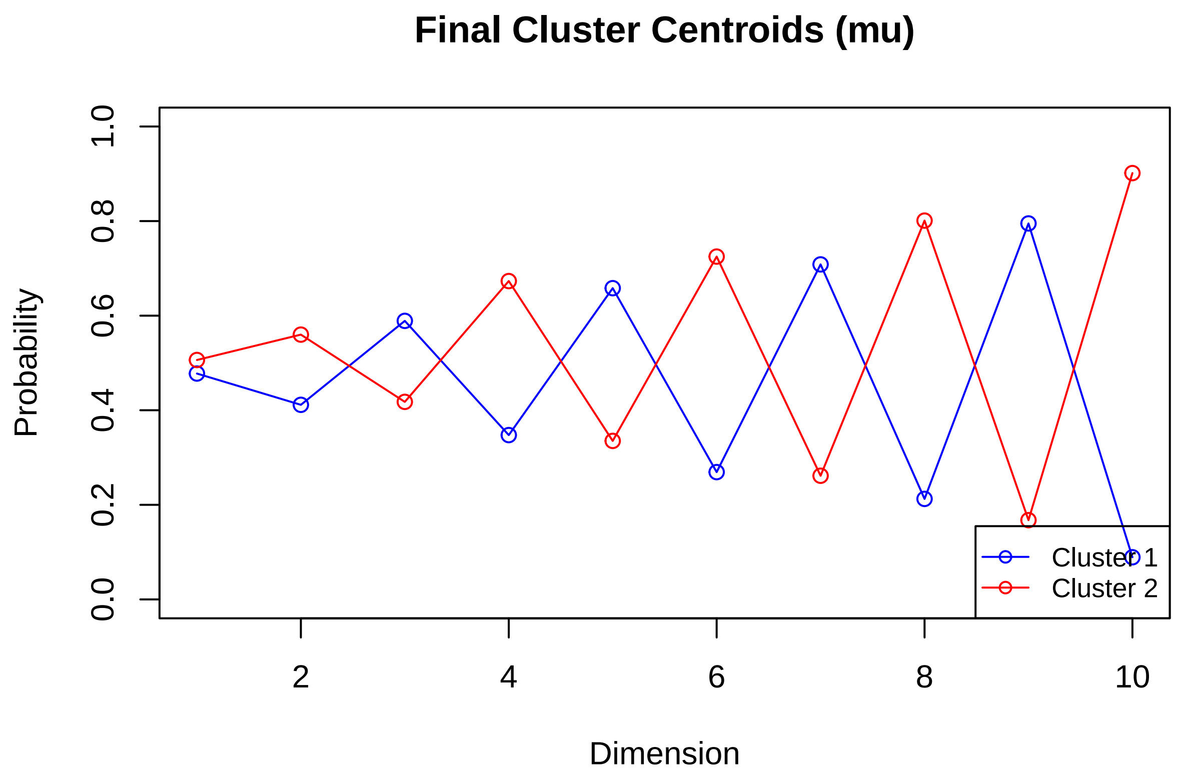
Where:

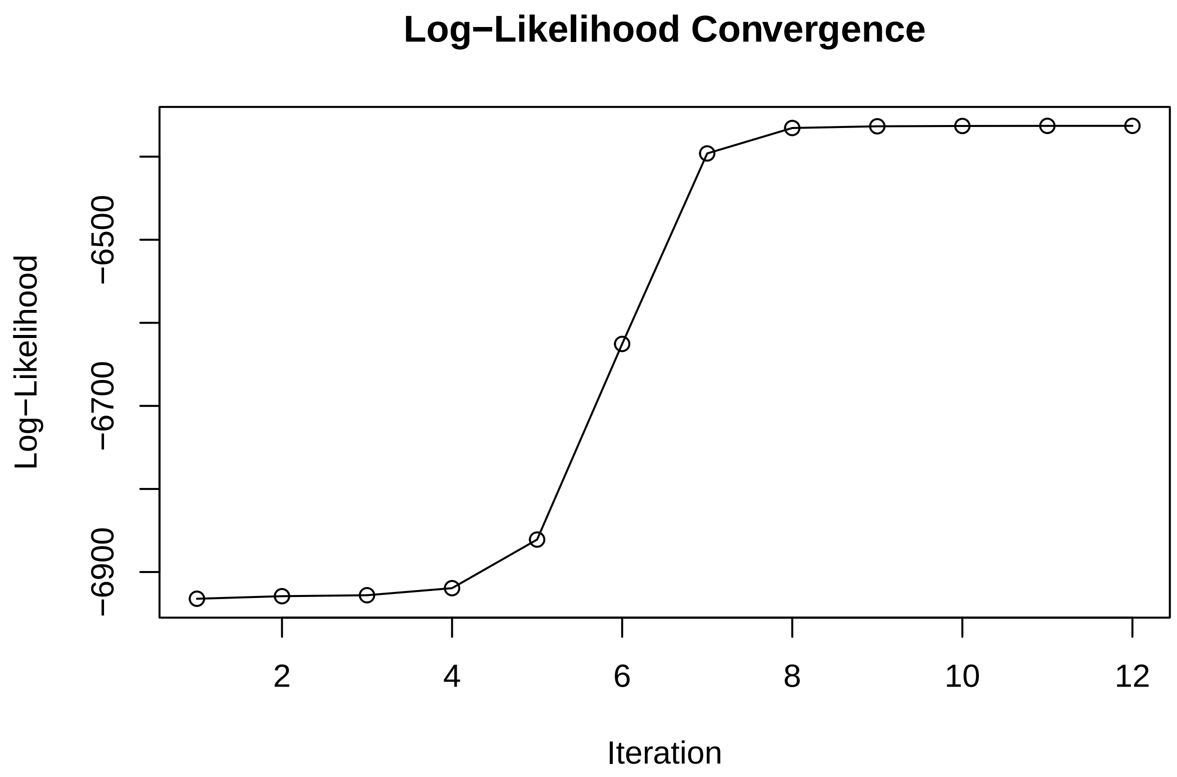
- $x_{i,d}$ is the value of the d-th dimension of the i-th data point.
- The denominator, $\sum_{i=1}^{n} w_{im}$, is the "effective" number of data points assigned to cluster m.

Model fitting results

Fitting results for 2 clusters

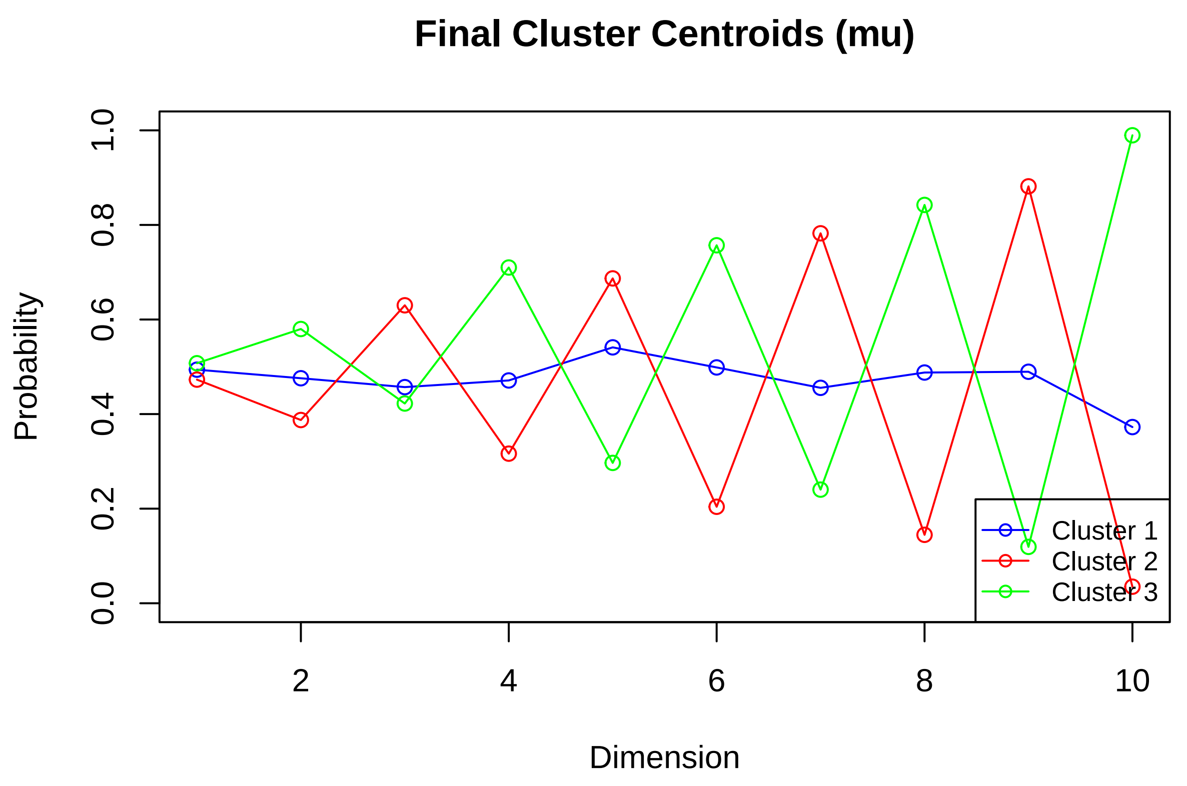
```
## [1] "Number of clusters:  2"
## [1] "Log likelihood:  -6362.88538329492"
## [1] "pi:  0.497915648485581, 0.502084351514419"
```

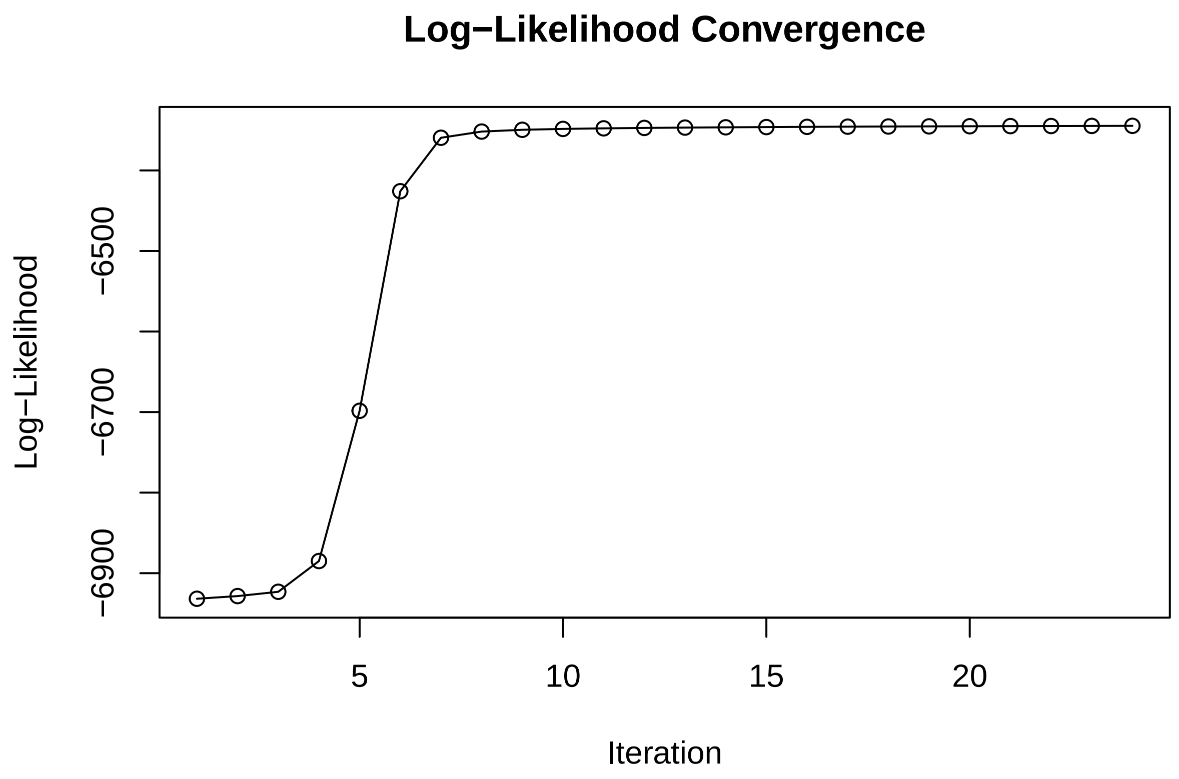




Fitting results for 3 clusters

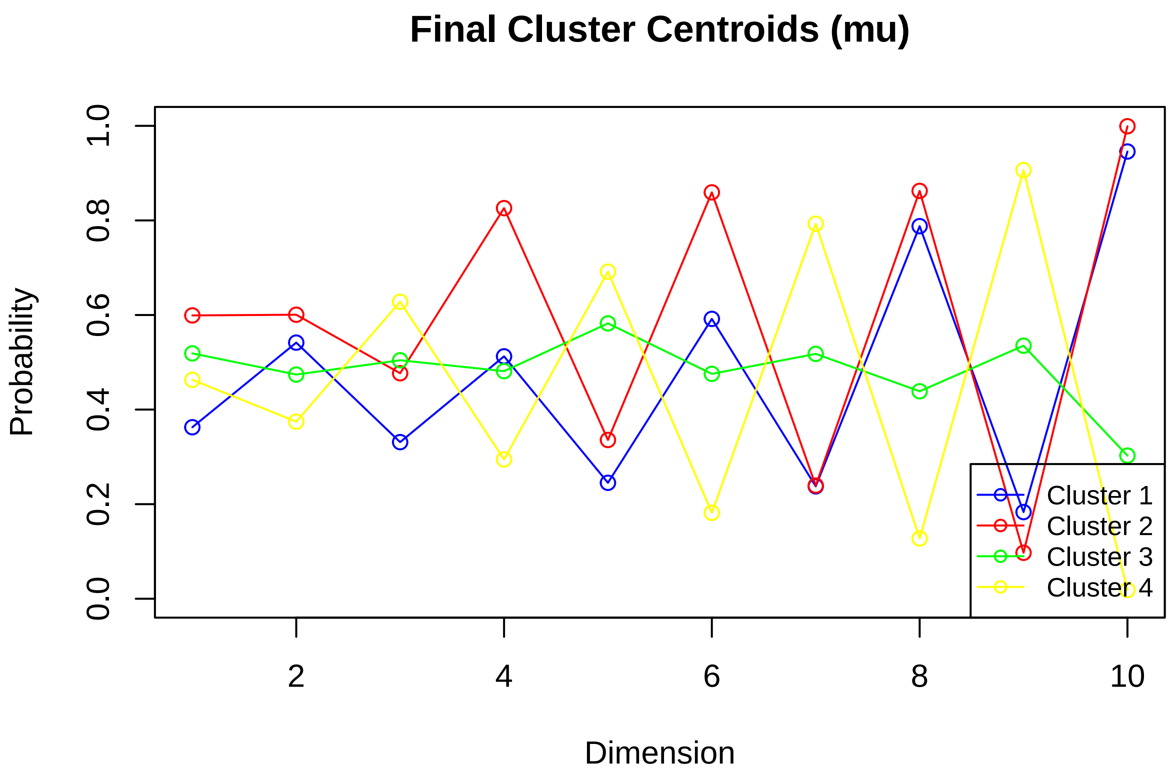
```
## [1] "Number of clusters:  3"
## [1] "Log likelihood:  -6344.64154880194"
## [1] "pi:  0.266336056408235, 0.343877971945799, 0.389785971645966"
```

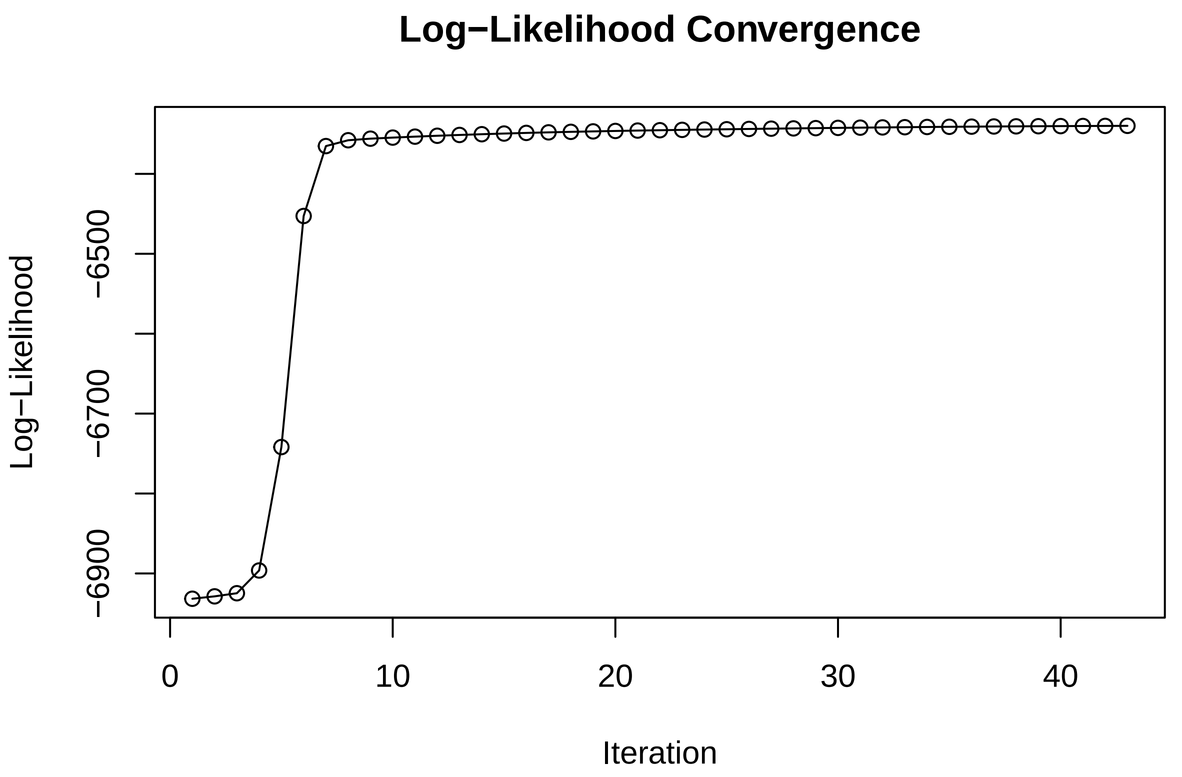

Final Cluster Centroids (mu)
Probability
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
Dimension
Cluster 1
Cluster 2
Cluster 3



Fitting results for 4 clusters

```
## [1] "Number of clusters:  4"
## [1] "Log likelihood:  -6339.89986543918"
## [1] "pi:  0.18381855672982, 0.230846580573274, 0.287418702198786, 0.29791616049812"
```


Final Cluster Centroids (mu)
Probability
Dimension
Cluster 1
Cluster 2
Cluster 3
Cluster 4
0.0
0.2
0.4
0.6
0.8
1.0
2
4
6
8
10



Analysis

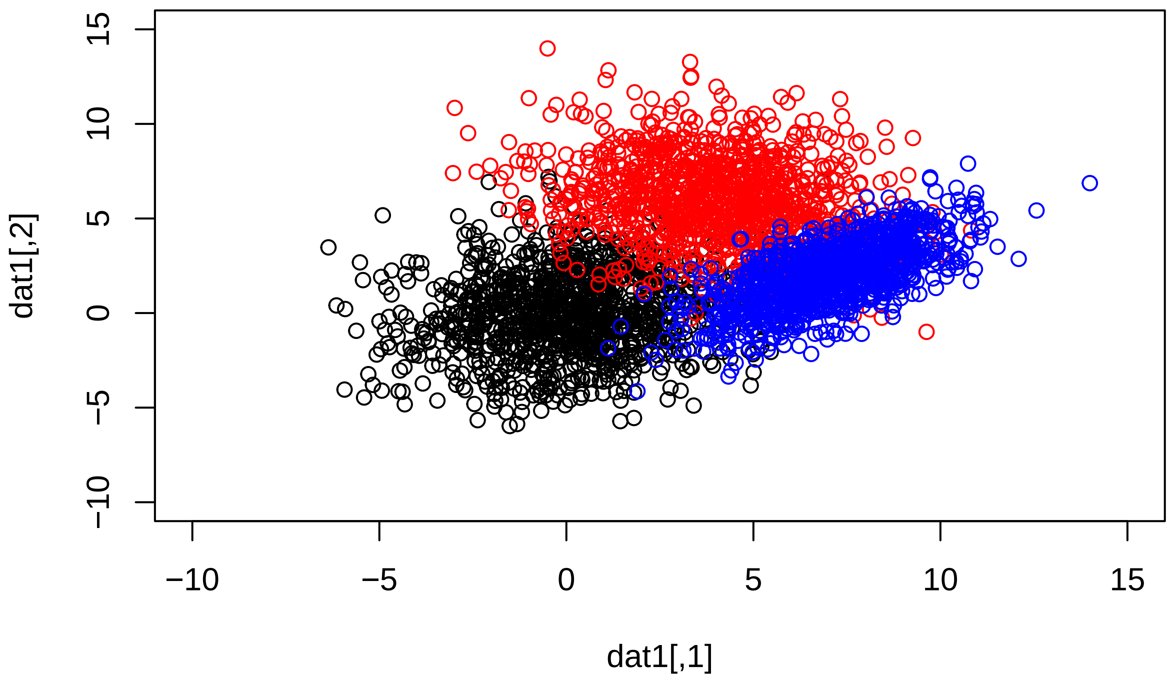
The above plots do not necessarily provide good insights into which number of clusters gives the best fit to the training data. We can say that the model with 3 clusters gives the best fit because the μ plot is similar to the original one we used for generating the data but in real life we won't have access to it. Therefore we need more objective criteria for model selection.

Akaike Information Criterion (AIC) and **Bayesian Information Criterion (BIC)** are standard tools for model selection. They balance model fit (log-likelihood) with model complexity (number of parameters). The model with the lowest AIC or BIC is considered the best.

Table 1: Model Selection Results

M	LogLikelihood	NumParams	AIC	BIC
2	-6362.89	21	12767.77	12870.83
3	-6344.64	32	12753.28	12910.33
4	-6339.90	43	12765.80	12976.83

Based on the results above, the model with 3 clusters provides the best fit as it has the lowest AIC and BIC compared to the models with 2 or 4 clusters.



```
## Task 1: Test Error (Single GMM/QDA Model) = 0.091
```

```
## Task 2: Test Error (Bagging with B = 10 ) = 0.09033333
```

Comments Bagging (bootstrap aggregation) reduces variance by majority voting or averaging many different models. But it only helps much when the base learners are high-variance (their predictions change a lot with different training samples). Bagging improves models that change a lot when the training data changes (high variance), like:

- decision trees
- neural networks
- nearest neighbors

Possible reasons bagging gave almost no improvement here:

- Low variance base model: The generative/QDA-style classifier (Gaussian per class) is fairly stable — different bootstrap samples produce very similar parameter estimates, so the B models are almost the same. Majority voting with similar models doesn't change much. Even when we bootstrap the dataset, each resample gives almost the same:
  - mean
  - covariance
  - prior

So each of the 10 models ends up nearly identical.When all models are similar, bagging gives almost no improvement.

- Strong correlation between base learners: With a small training set or when classes are well separated, bootstrap samples contain mostly the same information. The base models are highly correlated $\rightarrow$ majority vote gives little gain.

Assignment 4. THEORY.

Q1 In an ensemble model, is it true that the larger the number B of ensemble members, the more flexible the ensemble model? Page 169: No, the bias stays constant, atleast for numerical Y. The model does not fit the data better, bagging is a variance reduction techinque aimed to reduce the variance stemming from the sampling variance, by averaging over multiple simulated bootstrap samples. Hence the flexibility isn't increased.

Q2 In AdaBoost, what is the loss function used to train the boosted classifier at each iteration?

Page 177 lists the loss function used by the AdaBoost classifier as follows:

$$L(y \cdot f(\mathbf{x})) = \exp(-y \cdot f(\mathbf{x}))$$

where:

- y is the true label for the instance, typically $+1$ or -1
- $f(\mathbf{x})$ is the current margin output from the ensemble classifier for the data point $\mathbf{x}$ The product $y \cdot f(\mathbf{x})$ is known as the margin and indicates how confidently the classifier predicts the correct label.

Q3 Sketch how you would use cross-validation to select the number of components (or clusters) in unsupervised learning of GMMs.

Even though GMM learning is unsupervised, we can still do cross-validation. We simply treat the log-likelihood of held-out data as the validation score. To do this, we first split the dataset into training and validation parts (or k folds). Then, for each candidate number of components M, we train a GMM on the training portion using EM. Then we compute the held-out log-likelihood of the validation data under that model. Finally, we choose the M that gives the highest average held-out (cross-validated) likelihood (page 267).

Appendix

All code

```r
library(randomForest)
library(dplyr)
library(mvtnorm)

########################
#
# TASK 1 CODE
#
########################


# Simulating missclassification error against the same Y-labels for 1, 10 and 100 trees.
# Computing the spread and location of the missclassification error for 1000 simulations.
random_forest_sim <- function(NS = 25, condition = function(x1, x2){
  x1 < x2

}) {

  # ------------------------

  set.seed(1234)
  x1<-runif(1000)
  x2<-runif(1000)
  tedata<-cbind(x1,x2)
  y<-as.numeric(condition(x1, x2))
  telabels<-as.factor(y)



  # ------------------------



  ntree <- c(1, 10, 100)
  n <- 1000

  miss_df <- matrix(0, 3, length(ntree)) %>%  as.data.frame()
  rownames(miss_df) <- c("mu", "var", "sd")
  colnames(miss_df) <- ntree


  for (nt in ntree) {
    missrate <- rep(1, n)
    for (i in 1:n) {

      x1<-runif(100)
      x2<-runif(100)
      trdata<-cbind(x1,x2)
      y<-as.numeric(condition(x1, x2))
      trlabels<-as.factor(y)

      model <- randomForest(x = trdata, y = trlabels, ntree = nt, keep.forest = TRUE,
                            nodesize = NS)

      y_hat <- predict(model, tedata)
      missrate[i] <-  mean(y_hat != telabels)

    }

    miss_df[1, which(nt == ntree)] <- mean(missrate)
    miss_df[2, which(nt == ntree)] <- var(missrate)
    miss_df[3, which(nt == ntree)] <- sd(missrate)
  }

  return(miss_df)
}



# x1 < x2
rf1 <- random_forest_sim(NS = 25)

rf1 %>% round(4)
```

```r
# x1 < 0.5
rf2 <- random_forest_sim(condition = function(x1, x2){
  x1 < 0.5
})

rf2 %>% round(4)


# divide the domain into 4 equal squares, top right and bottom left is 1, otherwise 0.
rf3 <- random_forest_sim(NS = 12, condition = function(x1, x2){
  (x1<0.5 & x2<0.5) | (x1>0.5 & x2>0.5)
})

rf3 %>% round(4)




print("with the others nodesize")
rf1 <- random_forest_sim(NS = 12)
rf1 %>% round(4)
rf3 <- random_forest_sim(NS = 12, condition = function(x1, x2){
  (x1<0.5 & x2<0.5) | (x1>0.5 & x2>0.5)
})
rf3 %>% round(4)



#######################
#
# TASK 2 CODE
#
#######################

# --- Data Generation (with cluster tracking) ---
seed_value <- 1234567890
set.seed(seed_value)
n=1000
D=10
x <- matrix(nrow=n, ncol=D)
true_pi=c(1/3, 1/3, 1/3)
true_mu <- matrix(nrow=3, ncol=D)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)
true_mu[3,]=c(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5)
plot(true_mu[1,], type="o", col="blue", ylim=c(0,1))
points(true_mu[2,], type="o", col="red")
points(true_mu[3,], type="o", col="green")
# This vector will store the true cluster for each point
true_cluster_ids <- vector(length = n)

# Generate the training data
for(i in 1:n) {
  # Sample a cluster for the data point
  m <- sample(1:3, 1, prob = true_pi)
  # Store the cluster id
  true_cluster_ids[i] <- m
  # Generate the data point from the chosen cluster's distribution
  for(d in 1:D) {
    x[i,d] <- rbinom(1, 1, true_mu[m,d])
  }
}

# Plotting the Training Data with PCA
pca_result <- prcomp(x, center = TRUE, scale. = TRUE)
pca_plot_data <- as.data.frame(pca_result$x[, 1:2])
colnames(pca_plot_data) <- c("PC1", "PC2")
cluster_colors <- c("blue", "red", "green")
plot(pca_plot_data$PC1, pca_plot_data$PC2,
     main = "PCA of Training Data",
     xlab = "First Principal Component (PC1)",
     ylab = "Second Principal Component (PC2)",
     pch = 19,  # Use solid circles for points
     col = cluster_colors[true_cluster_ids]) # Assign colors based on true cluster IDs
legend("bottomright",
       legend = c("Cluster 1", "Cluster 2", "Cluster 3"),
       col = cluster_colors,
       pch = 19,
       title = "True Cluster")
em_algo <- function(M, x, seed=NULL) {
  n <- nrow(x)
  D <- ncol(x)
  max_it <- 100
  min_change <- 0.1
  w <- matrix(nrow=n, ncol=M) # weights
  pi <- vector(length = M) # mixing coefficients
  mu <- matrix(nrow=M, ncol=D) # conditional distributions
  llik <- vector(length = max_it) # log likelihood of the EM iterations

  # If a seed is provided, set it. This makes the run reproducible.
  # Important as the algorithm appears to be very sensitive to     initialization
  if (!is.null(seed)) {
    set.seed(seed)
  }
  # Random initialization of the parameters
  pi <- runif(M,0.49,0.51)
  pi <- pi / sum(pi)
  for(m in 1:M) {
    mu[m,] <- runif(D,0.49,0.51)
  }
  colors = c("blue", "red", "green", "yellow")
  for(it in 1:max_it) {
    # E-step: Computation of the weights
    for (i in 1:n) {
      # Calculate the probability of each data point belonging to each cluster
      for (m in 1:M) {
        # The term prod(mu[m,]^x[i,] * (1-mu[m,])^(1-x[i,])) calculates the probability
        # of observing the data point x[i,] given it was generated by cluster m.
        # This is multiplied by the mixing coefficient pi[m].
        w[i, m] <- pi[m] * prod(mu[m,]^x[i,] * (1 - mu[m,])^(1 - x[i,]))
      }
      # Normalize the weights for each data point so that they sum to 1.
      # This gives the posterior probability of cluster membership.
      w[i,] <- w[i,] / sum(w[i,])
    }
    #Log likelihood computation.
    llik[it] <- 0
    for (i in 1:n) {
      log_sum <- 0
      for (m in 1:M) {
        # The probability of a data point is the sum of the probabilities of that
        # data point being generated by each cluster, weighted by the mixing coefficients.
        log_sum <- log_sum + pi[m] * prod(mu[m,]^x[i,] * (1 - mu[m,])^(1 - x[i,]))
      }
      # We take the log of this sum and add it to the total log-likelihood.
      llik[it] <- llik[it] + log(log_sum)
    }
    flush.console()
    # Stop if the lok likelihood has not changed significantly
    if (it > 1) {
      if (abs(llik[it] - llik[it - 1]) < min_change) {
        break
      }
    }
    #M-step: ML parameter estimation from the data and weights
    # Re-estimate the mixing coefficients.
    for (m in 1:M) {
      # The new mixing coefficient for a cluster is the average of the weights
      # of that cluster over all data points.
      pi[m] <- sum(w[, m]) / n
    }
    # Re-estimate the conditional distributions (mu).
    for (m in 1:M) {
      # The new mean for a cluster is the weighted average of the data points,
      # where the weights are the posterior probabilities of cluster membership.
      mu[m,] <- colSums(x * w[, m]) / sum(w[, m])
    }
  }
  results <- list(
    pi = pi,                 # Final mixing coefficients
    mu = mu,                 # Final conditional probabilities (centroids)
    llik = llik[1:it]       # Log-likelihood values for iterations that were run
  )

  return(results)
}


plotting <- function(pi, mu, llik) {
  colors <- c("blue", "red", "green", "yellow")
  # par(mfrow = c(2, 1))
  plot(mu[1,], type="o", col="blue", ylim=c(0,1),
       main = "Final Cluster Centroids (mu)",
       xlab = "Dimension",
       ylab = "Probability")
  # grid()

  for (j in 2:nrow(mu)) {
    points(mu[j,], type="o", col=colors[j])
  }
  legend("bottomright", legend = paste("Cluster", 1:M), col = colors, lty = 1, pch = 1, cex=0.8)

  plot(llik[1:length(llik)], type="o",
       main = "Log-Likelihood Convergence",
       xlab = "Iteration",
       ylab = "Log-Likelihood")
  # grid()

  # par(mfrow = c(1, 1))
}

run_and_plot <- function(M, x, seed_value) {
  results <- em_algo(M, x, seed_value)
  pi <- results$pi
  mu <- results$mu
  llik <- results$llik
  print(paste("Number of clusters: ", M))
  print(paste("Log likelihood: ", llik[length(llik)]))
  print(paste("pi: ", paste(pi, collapse=", ")))
  plotting(pi, mu, llik)
  return(results)
}

all_models <- list()
M=2 # number of clusters
all_models[[as.character(M)]] <- run_and_plot(M, x, seed_value)
M=3 # number of clusters
all_models[[as.character(M)]] <- run_and_plot(M, x, seed_value)
M=4 # number of clusters
all_models[[as.character(M)]] <- run_and_plot(M, x, seed_value)
results_table <- data.frame(
  M = integer(),
  LogLikelihood = double(),
  NumParams = integer(),
  AIC = double(),
  BIC = double()
)

for (M in 2:4) {
  model <- all_models[[as.character(M)]]

  logL <- tail(model$llik, 1)
  k <- (M - 1) + (M * D)

  aic <- 2 * k - 2 * logL
  bic <- k * log(n) - 2 * logL

  results_table <- rbind(results_table, data.frame(
    M = M,
    LogLikelihood = logL,
    NumParams = k,
    AIC = aic,
    BIC = bic
  ))
}

knitr::kable(results_table, digits = 2, caption = "Model Selection Results")
#######################
#
# TASK 3 CODE
#
#######################



### DATA

set.seed(123)
N=300  # training
M=3000 # test
D=2    # dimensions

tr <- matrix(nrow=N, ncol=D+1)
te <- matrix(nrow=M, ncol=D+1)

B <- 10                  # bootstraps for bagging
pte <- matrix(nrow=M,ncol=B)  # store predictions from each model

mu1<-c(0,0)
Sigma1 <- matrix(c(5,1,1,5),D,D)
dat1<-rmvnorm(n = 1100, mu1, Sigma1)

mu2<-c(4,6)
Sigma2 <- matrix(c(5,-1,-1,5),D,D)
dat2<-rmvnorm(n = 1100, mu2, Sigma2)

mu3<-c(7,2)
Sigma3 <- matrix(c(3,2,2,3),D,D)
dat3<-rmvnorm(n = 1100, mu3, Sigma3)

# Plots the points
plot(dat1,xlim=c(-10,15),ylim=c(-10,15))
points(dat2,col="red")
points(dat3,col="blue")
```

```
# Training
tr[1:100,]   <- cbind(dat1[1:100,], 1)
tr[101:200,] <- cbind(dat2[1:100,], 2)
tr[201:300,] <- cbind(dat3[1:100,], 3)

# Test
te[1:1000,]    <- cbind(dat1[101:1100,], 1)
te[1001:2000,] <- cbind(dat2[101:1100,], 2)
te[2001:3000,] <- cbind(dat3[101:1100,], 3)

K = 3  # number of classes

### TRAIN GMM (Task 1)

train_gmm <- function(data) {
  x <- data[,1:D]
  y <- data[,D+1]

  pi <- numeric(K)
  mu <- matrix(0, nrow=K, ncol=D)
  Sigma <- array(0, dim=c(D,D,K))

  for(m in 1:K){
    xm <- x[y==m,,drop=FALSE]
    nm <- nrow(xm)

    pi[m] <- nm / nrow(data)                # pi_m
    mu[m,] <- colMeans(xm)                  # mu_m
    Sigma[,,m] <- cov(xm) * (nm-1)/nm       # sigma_m (ML estimate)
  }

  return(list(pi=pi, mu=mu, Sigma=Sigma))
}


### QDA Prediction using GMM

predict_qda <- function(model, X) {
  pi <- model$pi
  mu <- model$mu
  Sigma <- model$Sigma

  M_test <- nrow(X)
  scores <- matrix(0, nrow=M_test, ncol=K)

  for(m in 1:K){
    scores[,m] <- log(pi[m]) +
      dmvnorm(X, mean=mu[m,], sigma=Sigma[,,m], log=TRUE)
  }

  pred <- max.col(scores)    # argmax
  return(pred)
}
```

```r
### ------ TASK 1: SINGLE MODEL ------

model1 <- train_gmm(tr)
pred1 <- predict_qda(model1, te[,1:D])

true_labels <- te[,D+1]
error1 <- mean(pred1 != true_labels)

cat("Task 1: Test Error (Single GMM/QDA Model) =", error1, "\n")


### ------ TASK 2: BAGGING ------


### Utility: majority vote

getmode <- function(v) {
  uniqv <- unique(v)
  uniqv[which.max(tabulate(match(v, uniqv)))]
}

for(b in 1:B){

  # Bootstrap training sample
  idx <- sample(1:N, N, replace=TRUE)
  boot_data <- tr[idx,]

  # Train model on bootstrap
  model_b <- train_gmm(boot_data)

  # Predict test
  pte[,b] <- predict_qda(model_b, te[,1:D])
}

# Majority vote for each test point
final_pred <- apply(pte, 1, getmode)

error_bag <- mean(final_pred != true_labels)

cat("Task 2: Test Error (Bagging with B =",B,") =", error_bag, "\n")
```