Group_A11

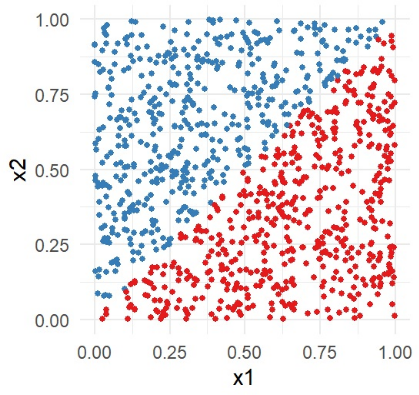
Computer lab block 2

Group A11 - Victor Guillo, Christian Kammerer, Jakob Lindner

Statement of contribution

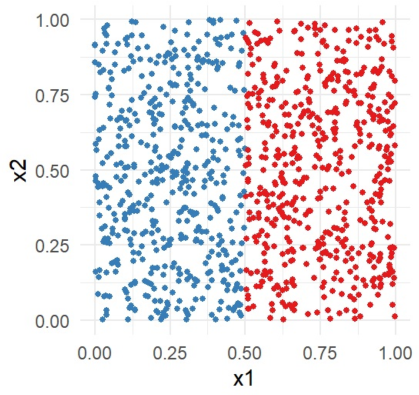
Assignment 1 was mainly contributed by Christian Kammerer, assignment 2 by Jakob Lindner. Victor Guillo answered the questions in assignment 3. Everyone was responsible for the respective parts in the report.

1. Ensemble Methods



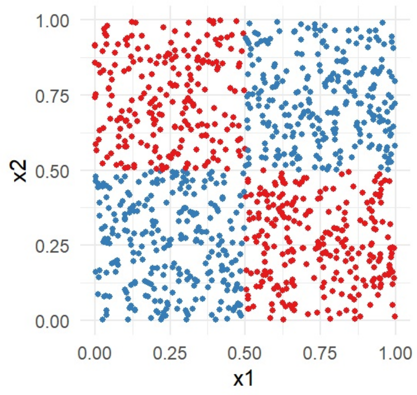
Results for Condition: x1 < x2

Tree Count	Mean	Variance
1	0.070534	0.0002977
10	0.035053	0.0000532
100	0.027425	0.0000468



Results for Condition: x1 < 0.5

Tree Count	Mean	Variance
1	0.009349	0.0002819
10	0.000537	0.0000015
100	0.000284	0.0000003



Results for Condition: Checkerboard

Tree Count	Mean	Variance
1	0.033669	0.0004747
10	0.004370	0.0000091
100	0.002910	0.0000045

Question: What happens with the mean error rate when the number of trees in a forest increases? Why?
Answer: In this experiment, as the number of trees increases from 1 to 100, we observe a decreasing trend in the mean error rate. This is due to the random forest's ability to average predictions from multiple trees, reducing variance and improving overall accuracy.

Question: 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.
Answer: Decision trees rely on axis-aligned splits (e.g. x1 < a, x2 > b), these splits lead to the creation of rectangular regions. A linear boundary such as x > y is not easily represented through these splits. As such, random forests are poorly suited to model linear relationships. Through feature engineering a new feature z with $z = x_1 - x_2$, we could capture this linear relationship and allow the model to use axis-aligned splits to capture the linear relationship between x_1 and x_2. This demonstrates the importance of understanding the problem you are dealing with and making informed decisions regarding model and feature choice. While the checkerboard is a more complex relationship, it is easy to capture through rectangular regions and thus, as soon as there is a sufficient number of trees, the performance is very good.

2. Mixture models

Implementation

At first, the missing parts in the code template were implemented following the given formulas. In the E-step the weights are computed using the first two formulas. For each datapoint and model the weights are computed, so we iterate them in two loops and calculate them over all dimensions. To get the correct results it is necessary to normalise the weights in the end, even though it is not in the formulas. This is due to the split of the weight on the different models.

```
# E-step: Computation of the weights
  # iterate all datapoints
  for (i in 1:n){
    # iterate all models
    for (m in 1:M){
      bern <- 1 # starting value for bernoulli
      for (d in 1:D){
        bern_d <- mu[m,d]^(x[i,d]) * (1-mu[m,d])^(1-x[i,d]) # compute bernoulli for every dimension
        bern <- bern * bern_d # update the product
      }
      w[i,m] <- pi[m] * bern # every datapoint gets a weight for each model
    }
    w[i,] <- w[i,]/(sum(w[i,])) # normalise the weights (not in forumlas)
  }
```

The log likelihood is computed according to the given formula. To get $p(x_i)$ the the weights over the models are summed using sum() in the end.

```
llik[it] <- sum(colSums(log(w)))
```

The break criterion is checked beginning with the second iteration, as there is a value to compare for the first time. It simply checks if there has been a significant change in the logliklihood compared to the last iteration.

```
if (it > 1){
  if (abs(llik[it] - llik[it-1]) <= 0.001){
    cat("No significant change in likelihood:", llik[it-1], "to", llik[it])
    break
  }
}
```

In the M-step the parameters are updated following the formulas in the lecture slides.

```
#M-step: ML parameter estimation from the data and weights
for (m in 1:M){
  # update pis
  pi[m] <- sum(w[,m]) / n

  # update mus
  weighted_x <- vector(length=D)
  for (i in 1:n){
    # update the sum with the product of every datapoint with its weigth
     weighted_x <- weighted_x + sum(w[i,m]) * x[i,]
  }
  mu[m,] <- 1/sum(w[,m]) * weighted_x
}
```

Results

For the correct amount of clusters, 3, the algorithm approximates really good. In the resulting matrix row 1 belongs to row 2 in true_mu belongs, row 2 to row 3 and row 3 to row 1.

```
## Learned mu for 3 clusters:
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4739837 0.3795954 0.6283379 0.3060690 0.6967346 0.1963714 0.7895256
## [2,] 0.4901669 0.4799835 0.4735775 0.4791284 0.5328966 0.4883912 0.4698887
## [3,] 0.5092049 0.5835304 0.4191029 0.7160556 0.2903244 0.7681524 0.2305951
##           [,8]      [,9]      [,10]
## [1,] 0.1325377 0.8940952 0.01393594
## [2,] 0.4858342 0.4972243 0.39873386
## [3,] 0.8529426 0.1056945 0.99999659
```

```
## True mu:
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  0.5  0.6  0.4  0.7  0.3  0.8  0.2  0.9  0.1   1.0
## [2,]  0.5  0.4  0.6  0.3  0.7  0.2  0.8  0.1  0.9   0.0
## [3,]  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5   0.5
```

For too few clusters, in this case only 2, the tendency of the two mus still have similarity with some of the true mus (learned mu[1] is fairly close to true_mu[2], and learned mu[2] fairly close to true_mu[1], but it is already ~0.1 off. The reason is in the underfitting of the model, as it can’t represent the data belonging to the missing clusters.

```
## Learned mu for 2 clusters:
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4777588 0.4113943 0.5885943 0.3479502 0.6578675 0.2703223 0.7073108
## [2,] 0.5061864 0.5603188 0.4177348 0.6734228 0.3347549 0.7248023 0.2615478
##           [,8]      [,9]      [,10]
## [1,] 0.2140392 0.7934210 0.08781751
## [2,] 0.8008305 0.1677841 0.90460919
```

```
## True mu:
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  0.5  0.6  0.4  0.7  0.3  0.8  0.2  0.9  0.1   1.0
## [2,]  0.5  0.4  0.6  0.3  0.7  0.2  0.8  0.1  0.9   0.0
## [3,]  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5   0.5
```

For too many clusters, in this case 4, there no assignments are possible anymore. The reason is, that points of the same cluster may be assigned to different clusters and the model tries to recognise patterns where are none.

```
## Learned mu for 4 clusters:
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4633149 0.3662611 0.6253714 0.2795445 0.7044031 0.1725783 0.8036316
## [2,] 0.5673964 0.5823838 0.4691582 0.8400973 0.3167123 0.8075927 0.2437998
## [3,] 0.5099661 0.4713629 0.5221603 0.4810820 0.5743314 0.4514417 0.5326041
## [4,] 0.3384837 0.5627163 0.2690772 0.3436690 0.2535198 0.6170834 0.2166402
##           [,8]      [,9]        [,10]
## [1,] 0.1185163 0.9198230 0.0005197044
## [2,] 0.8414873 0.1130854 0.9999987408
## [3,] 0.4146392 0.5629958 0.3073588722
## [4,] 0.8294784 0.1525077 0.9124622632
```

```
## True mu:
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  0.5  0.6  0.4  0.7  0.3  0.8  0.2  0.9  0.1   1.0
## [2,]  0.5  0.4  0.6  0.3  0.7  0.2  0.8  0.1  0.9   0.0
## [3,]  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5   0.5
```

3. Theory

Question 1 : In an ensemble model, is it true that the larger the number B of ensemble members the more flexible the ensemble model?

In bagging , increasing the number B of ensemble members reduces variance by averaging predictions but does not increase flexibility, as flexibility is determined by the complexity of the base models (e.g., tree depth). This leads to more stable and generalized predictions without affecting bias (Pages 169-171). In boosting, increasingB increases flexibility by sequentially correcting errors, reducing bias, and allowing the ensemble to fit more complex patterns. However, this added flexibility also increases the risk of overfitting when B becomes too large or base models are overly complex (Pages 175-178,188). Thus, while bagging stabilizes predictions, boosting balances bias reduction and overfitting risk.

Question 2 : In AdaBoost, what is the loss function used to train the boosted classifier at each iteration?

In AdaBoost, the exponential loss function is used to train the boosted classifier at each iteration. The loss function is defined as: $L(y,f(x))=\exp(-y\cdot f(x))$, where y represents the true label (+1 or -1), and f(x) is the weighted sum of predictions made by all ensemble members up to the current iteration. The exponential loss is minimized iteratively, and each ensemble member is trained to reduce this loss further by correcting the errors of the previous ensemble members. This sequential minimization allows AdaBoost to effectively reduce bias. Pages 175-177

Question 3 : Sketch how you would use cross-validation to select the number of components (or clusters) in unsupervised learning of GMMs.

To use cross-validation for selecting the number of clusters M in Gaussian Mixture Models (GMMs), you first divide the data into training and validation sets. For each candidate M, train a GMM on the training set and compute the log-likelihood on the validation set. Repeat this for different values of M, and select the M that yields the highest validation log-likelihood. This approach helps prevent overfitting and ensures the chosen model generalizes well to unseen data. Page 267

Apendix

```
################
# Assignment 1 #
################

library(randomForest)

# Function to generate data
generate_data <- function(seed, condition) {
  if (!is.null(seed)) {
    set.seed(seed)
  }

  x1 <- runif(1000)
  x2 <- runif(1000)
  y <- eval(condition)
  data <- data.frame(
    x1 = x1,
    x2 = x2,
    y = as.factor(y)
  )
  return(data)
}

# Initialize results list
results <- list()

# Define conditions
conditions <- list(
  "x1 < x2" = list(cond = quote(x1 < x2), nodesize = 25),
  "x1 < 0.5" = list(cond = quote(x1 < 0.5), nodesize = 25),
  "checkerboard" = list(cond = quote(((x1 < 0.5 & x2 < 0.5) | (x1 > 0.5 & x2 > 0.5))),
```

```r
                        nodesize=12)
)


run_experiment <- function(){
  # Iterate through conditions
  for (condition_name in names(conditions)) {
    condition <- conditions[[condition_name]]$cond
    nodesize <- conditions[[condition_name]]$nodesize
    # Generate test data
    test_data <- generate_data(seed = 1234, condition = condition)

    # Error matrix to store errors for each seed and tree count
    error_matrix <- matrix(0, nrow = 1000, ncol = 3)

    for (i in 1:1000) { # Iterate over seeds
      # Generate training data

      data_set <- generate_data(seed = i, condition = condition)

      for (j in 1:3) { # Iterate over tree counts (1, 10, 100)
        ntree <- 10^(j - 1)
        model <- randomForest(y ~ x1 + x2, data = data_set, ntree = ntree, nodesize = nodesize)
        predictions <- predict(model, newdata = test_data)
        misclassified <- sum(test_data$y != predictions)
        error_matrix[i, j] <- misclassified / 1000
      }
    }

    # Calculate means and variances
    error_means <- colMeans(error_matrix)
    error_variances <- apply(error_matrix, 2, var)

    # Store results
    results[[condition_name]] <- list(
      means = error_means,
      variances = error_variances
    )
  }
  return(results)
}


################
# Assignment 2 #
################

set.seed(1234567890)
max_it <- 100 # max number of EM iterations
min_change <- 0.1 # min change in log lik between two consecutive iterations
n=1000 # number of training points
D=10 # number of dimensions
x <- matrix(nrow=n, ncol=D) # training data
true_pi <- vector(length = 3) # true mixing coefficients
true_mu <- matrix(nrow=3, ncol=D) # true conditional distributions
true_pi=c(1/3, 1/3, 1/3)
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")
# Producing the training data
for(i in 1:n) {
  m <- sample(1:3,1,prob=true_pi)
  for(d in 1:D) {
    x[i,d] <- rbinom(1,1,true_mu[m,d])
  }
}

M=4 # number of clusters
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
# 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)
}

pi
mu



for(it in 1:max_it) {
  plot(mu[1,], type="o", col="blue", ylim=c(0,1))
  points(mu[2,], type="o", col="red")
  points(mu[3,], type="o", col="green")
  #points(mu[4,], type="o", col="yellow")
  Sys.sleep(0.5)
  # E-step: Computation of the weights
  # iterate all datapoints
  for (i in 1:n){
    # iterate all models
    for (m in 1:M){
      bern <- 1 # starting value for bernoulli
      for (d in 1:D){
        bern_d <- mu[m,d]^(x[i,d]) * (1-mu[m,d])^(1-x[i,d]) # compute bernoulli for every dimension
        bern <- bern * bern_d # update the product
      }
      w[i,m] <- pi[m] * bern # every datapoint gets a weight for each model
    }
    w[i,] <- w[i,]/(sum(w[i,])) # normalise the weights (not in forumlas)
  }

  #Log likelihood computation.
  llik[it] <- sum(colSums(log(w)))

  cat("iteration: ", it, "log likelihood: ", llik[it], "\n")
  flush.console()
  # Stop if the log likelihood has not changed significantly
  # Your code here
  if (it > 1){
    if (abs(llik[it] - llik[it-1]) <= 0.001){
      cat("No significant change in likelihood:", llik[it-1], "to", llik[it])
      break
    }
  }
  #M-step: ML parameter estimation from the data and weights
  for (m in 1:M){
    # update pis
    pi[m] <- sum(w[,m]) / n

    # update mus
    weighted_x <- vector(length=D)
    for (i in 1:n){
       weighted_x <- weighted_x + sum(w[i,m]) * x[i,]
    }
    mu[m,] <- 1/sum(w[,m]) * weighted_x
  }

}
```