Eigenfaces

Simon Schmetz and Florencia Luque

2025-02-12

```
# library Imports
library(OpenImageR)
library(EBImage)
```

```
##
## Adjuntando el paquete: 'EBImage'
```

```
## The following objects are masked from 'package:OpenImageR':
##
##     readImage, writeImage
```

```
library(grid)
library(ggplot2)
library(caret)
```

```
## Cargando paquete requerido: lattice
```

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Adjuntando el paquete: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(tictoc)
library(dplyr)
```

```
##
## Adjuntando el paquete: 'dplyr'
```

```
## The following object is masked from 'package:EBImage':
##
##     combine
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(MASS)
```

```
##
## Adjuntando el paquete: 'MASS'

## The following object is masked from 'package:dplyr':
##
##     select
```

```
library(gtools)
library(reshape2)
library(gridExtra)
```

```
##
## Adjuntando el paquete: 'gridExtra'

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:EBImage':
##
##     combine
```

README

The following Document is Split into three Main Parts: - Functions (only definitions) - Code execution to find optimal classification set up - Classification execution for new data (Last code chunk of Rmd) <- This is the code that should be used to evaluate the classification

Introduction

The following document contains the work done for assignments 1 and 2 of the course on statistical learning at Universidad Carlos III de Madrid. This assignment aims to implement two facial recognition classifiers based on the K-nearest neighbor algorithm and using Principal Component Analysis (PCA) and Fisher Discriminant Analysis (FDA). Both techniques aim to reduce the dimensionality of the image data while preserving critical information for classification.

In Part A, we employ PCA to construct a facial recognizer. PCA is an unsupervised method that finds the principal components of the data, capturing the directions of maximum variance. Using PCA, we project the images onto a lower-dimensional space and classify them using a k-nearest neighbors (k-NN) classifier. The

classification process involves determining the optimal number of principal components to retain, selecting an appropriate number of neighbors in k-NN, defining a similarity metric, and establishing a threshold for determining whether a given image belongs to the database.

In Part B, we implement a facial recognizer using Fisher Discriminant Analysis (FDA). Unlike PCA, FDA is a supervised method that finds a projection maximizing the class separability. This ensures that images of the same person remain close together while different individuals are more distinguishable. Similar to Part A, we use a k-NN classifier on the Fisher-discriminant-transformed data and determine the appropriate hyperparameters to optimize performance.

We begin with defining the functions required for general use, PCA, FDA and KNN as well as cross validation and evaluation. Afterwards we use these functions to complete Part A and Part B of the assignment

Funcions

In the following section, we define all the functions that will be used later on for both Part A and Part B.

Data Import

The R function "read_all_images" reads all image files from a specified folder, extracts labels and filenames from the image file names, and converts the image data into a data frame format. It processes each image by reading its pixel values, combining the RGB channels into a single row, and appending the extracted labels and filenames to the data frame.

```
### Read Data functions
read_all_images <- function(folder_path) {
  image_files <- list.files(folder_path, full.names = TRUE, pattern = "\\.(jpg|png|jpeg|tiff|bmp)$", ig

  # Sort filenames
  image_files <- mixedsort(image_files)

  # Extract labels from filenames
  extract_label <- function(filename) {
    base_name <- tools::file_path_sans_ext(basename(filename))
    label <- gsub("[^0-9]", "", base_name)  # Extract numeric part
    return(label)
  }
  extract_filename <- function(filepath) {
    return(basename(filepath))
  }
  read_data <- function(image_path) {
    img <- readImage(image_path)
    red_aux   <- as.vector(img[,,1])
    green_aux <- as.vector(img[,,2])
    blue_aux  <- as.vector(img[,,3])

    # Combine all channels into a single row
    img_vector <- c(red_aux, green_aux, blue_aux)

    return(as.data.frame(t(img_vector)))  # Transpose to make it a row
  }

  image_list <- lapply(image_files, function(file) {
```

```
    img_data <- read_data(file)
    img_data$Label <- extract_label(file)
    img_data$ID <- extract_filename(file)
    return(img_data)
  })

  ax <- do.call(rbind, image_list)
  return(ax)
}
```

Distance Histograms

The function "distance_distributions" calculates pairwise distances between rows in a dataset using a specified distance metric and plots histograms of these distances, distinguishing between distances within the same group and distances between different groups.

```
### Distance Histograms
distance_distributions <- function(data, distance_metric="euclidean"){

  ids <- data[, ncol(data)]

  # set up distance matrix/data frame
  dist_matrix <- as.matrix(dist(data[, -ncol(data)], method = distance_metric))
  dist_df <- data.frame(
    row1 = rep(1:nrow(data), each = nrow(data)),
    row2 = rep(1:nrow(data), times = nrow(data)),
    distance = as.vector(dist_matrix)
  )
  dist_df <- dist_df[dist_df$row1 != dist_df$row2, ] # remove self distances

  dist_df$id1 <- ids[dist_df$row1] # Assign IDs
  dist_df$id2 <- ids[dist_df$row2]

  dist_df_same_id <- dist_df[dist_df$id1 == dist_df$id2, ] # In group distances
  dist_df_same_id <- dist_df_same_id[order(dist_df_same_id$id1), ]


  dist_df_diff_id <- dist_df[dist_df$id1 != dist_df$id2, ] # Outside group distances

  dist_df_diff_id$id1 = 0
  dist_df_diff_id$id2 = 0

  dist_df_split <- rbind(dist_df_same_id,dist_df_diff_id)


  # Plot histogram
  plot = ggplot(dist_df_split, aes(x = distance, fill = factor(id1))) +
    geom_density(alpha = 0.6, position = "identity") +
    labs(title = "Histogram of Pairwise Distances by ID",
         x = "Distance", y = "Frequency", fill = "ID Group") +
    theme_minimal()

  return(plot)
```

```
}
```

Principal Component Analysis (PCA)

The following section contains a function to perform Principal Component Analysis (PCA) on any kind of Data. The function is first defined and then applied to the image data and its results are compared to the base R pca function.

This function is created using the following formula.

$$\Sigma_s = \frac{1}{n-1} GG^t$$

Where G is the matrix of features less the mean of each column. The eigen vectors of this new matrix are the same eigen vector of the original matrix and if you multiply $G^t \cdot eigen\_vector$ you will get the eigen vector of the typical matrix.

```
### PCA Function
PCA.fun = function(X,matrix_bool = F){
  if (!matrix_bool){
    X = X %>% dplyr::select(-c("Label","ID"))
    print(dim(X))
    X = as.matrix(X)
  } else{
    X = X[,-ncol(X)]
    print(dim(X))
  }

  n = nrow(X)
  mu = colMeans(X)
  # center data
  G = sweep(X, 2, mu, FUN = "-")
  # Compute the covariance matrix
  cov_matrix =(1/(n-1))*(G %*% t(G))
  ## Calculate Eigenvalues and Eigenvectors and sort
  eig = eigen(cov_matrix)
  eig_val = eig$values
  eig_vec = eig$vectors
  ei_vec_large = t(G)%*%eig_vec
  # sort
  sort_index <- order(eig_val, decreasing = TRUE)
  eig_val_sorted <- eig_val[sort_index]
  eig_vec_sorted <- ei_vec_large[, sort_index]
  #variability of each PC
  D = eig_val_sorted/sum(eig_val_sorted)
  return(list("Eigen Vector"= eig_vec_sorted,"D"=D))
  }
```

Fisher Discriminant Analysis (FDA)

This function implements Fisher Discriminant Analysis to find a projection matrix P that maximizes class separability. It computes the within-class scatter matrix S_w and between-class scatter matrix S_b, then

solves the generalized eigenvalue problem for $S_w^{-1}S_b$. Since S_w can be singular or ill-conditioned, direct inversion is unstable. Instead, we use Cholesky decomposition $S_w = LL^T$ to transform the problem into a standard eigenvalue decomposition, ensuring numerical stability. The function returns the mean vector μ, the projection matrix P (top eigenvectors), and the variance explained D by each discriminant.

```
FDA.fun = function(data, labels, reg = 1e-6) {
  if(!is.matrix(data)){
    data = as.matrix(data)
  }


  labels = as.factor(labels)

  num_classes = length(unique(labels))
  num_features = ncol(data)
  #overall mean
  mean_total = colMeans(data)
  #mean by class
  class_levels = levels(labels)
  class_means = matrix(0, nrow = num_classes, ncol = num_features)
  rownames(class_means) = class_levels

  for (i in 1:num_classes) {
    class_means[i, ] = colMeans(data[labels == class_levels[i], , drop = FALSE])
  }
  #within classes matrix
  Sw = matrix(0, num_features, num_features)
  for (i in 1:num_classes) {
    class_data = data[labels == class_levels[i], , drop = FALSE]
    mean_diff = sweep(class_data, 2, class_means[i, ], "-")
    Sw = Sw + t(mean_diff) %*% mean_diff
  }

  Sw = Sw + reg * diag(num_features)

  Sb = matrix(0, num_features, num_features)
  #between classes matrix
  for (i in 1:num_classes) {
    mean_diff = matrix(class_means[i, ] - mean_total, ncol = 1)
    Sb = Sb + sum(labels == class_levels[i]) * (mean_diff %*% t(mean_diff))
  }

  L = chol(Sw)
  L_inv = solve(L)
  S_transformed = t(L_inv) %*% Sb %*% L_inv

  eig = eigen(S_transformed)
  idx = order(Re(eig$values), decreasing = TRUE)
  eigenvalues = Re(eig$values[idx])
  P = L_inv %*% Re(eig$vectors[, idx])

  num_components = min(num_classes - 1, num_features)
  eigenvalues = eigenvalues[1:num_components]
  P = P[, 1:num_components, drop = FALSE]
```

```
  D = eigenvalues / sum(eigenvalues)

  transform_data = data %*% P
  return(list(mean = mean_total, P = P, D = D,transform_data= transform_data,labels=labels))
}
```

Test/Train Split

The "train_test_split" function splits a dataset into training and testing sets based on a specified ratio.

```
### Train/Test Split
train_test_split <- function(data, train_ratio = 0.8, seed = NULL) {
  if (!is.null(seed)) {
    set.seed(seed)
  }
  # Split
  n = nrow(data)
  train_indices <- sample(1:n, size = floor(train_ratio * n))
  train_data <- data[train_indices, , drop = FALSE]  # Train set
  test_data <- data[-train_indices, , drop = FALSE]  # Test set

  return(list(train = train_data, test = test_data))
}
```

Distance Metrics

In the following chunk, two distance metrics are defined.

```
### Distances

# Euclidean distance
euclidean_distance <- function(x, y) {
  sqrt(sum((x - y)^2))
}

# Manhattan distance
manhattan_distance <- function(x, y) {
  sum(abs(x - y))
}
```

Classificaion Threshholds

The function "estimate_thresholds" calculates thresholds for a number of groups per group and returns the threshholds with their corresponding group ID. The function that is used for calculation is interchangeable, with one defined as the quantile of the empirical distribution of the given data.

```
### Threshhold functions

## precentile threshold function
estimate_threshold_via_percentile <- function(data, percentile = 0.90, distance_metric = "euclidean") {
```

```
  if (distance_metric == "euclidean") {
    dist_1 <- as.matrix(dist(data, method = "euclidean"))
  } else if (distance_metric == "manhattan") {
    dist_1 <- as.matrix(dist(data, method = "manhattan"))
  } else {
    stop("Invalid method. Choose either 'euclidean' or 'manhattan'.")
  }

  threshold <- quantile(dist_1[upper.tri(dist_1)], percentile) # *1.5 would be an option to increase th

  return(threshold)
}

## generate Treshhold functions
estimate_thresholds <- function(train_data, estimate_func, percentile, distance_metric) {
  thresholds <- numeric()
  train_data_ids_unique <- unique(train_data[, ncol(train_data)])

  for (id in train_data_ids_unique) {
    train_data_grouped <- train_data[train_data[, ncol(train_data)] == id, , drop = FALSE]
    if (nrow(train_data_grouped) > 1) {
      thresholds[id] <- estimate_func(train_data_grouped[,-ncol(train_data_grouped)],percentile = percer
    } else{ # drop as to not enough data to set threshold
      train_data_ids_unique =  setdiff(train_data_ids_unique, id)
    }
  }

  names(thresholds) <- train_data_ids_unique


  return(thresholds)
}
```

KNN Classifier Function

The "knn_classifier" function classifies test data based on the k-nearest neighbors (KNN) algorithm, using a specified distance metric and thresholds for each group. It computes distances between each test data point and all training data points, then assigns the most frequent label among the k-nearest neighbors if the distance threshold is met. If no threshold is met, the test data point is assigned to a default group. It return a id based classification and a "binary classification" which only considers weather the to be classified element is within the data base or not (e.g. if it meets the treshhold or not).

```
### KNN Classifier
knn_classifier = function(test_data, train_data, thresholds, dist_metric = "euclidean", k, knn_bool = T
  # init results array
  predicted_ids = c()

  # For train data, get true ID and drop id col
  train_data_ids <- train_data[, ncol(train_data)]  # Extract ID column
  train_data_no_last_col <- train_data[, -ncol(train_data), drop = FALSE]


  # for all rows (images) run
```

```r
for (i in 1:nrow(test_data)) {

  ### Prepare Distances

  # for test data, get row
  test_row = test_data[i,]

  # Compute distances between the test row and each row in the training data and assign IDs
  distances <- apply(train_data_no_last_col, 1, function(train_row) {
    test_row <- as.numeric(test_row)
    train_row <- as.numeric(train_row)

    if (dist_metric == "euclidean") {
      return(euclidean_distance(test_row, train_row))
    } else if (dist_metric == "manhattan") {
      return(manhattan_distance(test_row, train_row))
    }
  })

  names(distances) <- train_data_ids

  ### Classification
  predicted_label <- 0
  # fot all groups IDs, check if threshhold is passed for at least one
  for (train_ids in names(thresholds)) {
    # perform knn as soon as distance threshhod is passed for one
    if (any(distances[names(distances) == train_ids] < thresholds[train_ids])) {
      if (knn_bool){

        k_neighbors <- order(distances)[1:min(k, length(distances))] # with k >= training points
        neighbor_labels <- train_data_ids[k_neighbors]  # Get labels of k-nearest neighbors

        # Get most frequent label
        predicted_label <- names(sort(table(neighbor_labels), decreasing = TRUE))[1]

        # store result
        predicted_ids = c(predicted_ids, predicted_label)
        break  # Stop once a group match is found

      } else{ # in case no KNN and only checking if in data or not
        predicted_label = 1
        predicted_ids = c(predicted_ids, predicted_label)
        break
      }
    }
  }

  # if not passing any threshholds, assign group 0 (no group)
  if (predicted_label == 0) {
      predicted_ids = c(predicted_ids, predicted_label)
  }
}
```

```
  # return predictions
  return(predicted_ids)
}
```

Score Classification

The score function calculates performance metrics for both ID classification and binary classification. It computes the match and fail rates for ID classification, and for binary classification, it calculates the false positive and false negative rates using a confusion matrix.

```
### Scoring
score <- function(df_id,df_binary) {

  ### ID Classification Scoring

  # match & fail rate
  match_rate <- mean(df_id$true == df_id$predicted)
  fail_rate <- 1 - match_rate

  ### Binary Classification Scoring
  # factorize
  df_binary$true = factor(df_binary$true, levels = c(0, 1))
  df_binary$predicted = factor(df_binary$predicted, levels = c(0, 1))

  # confusion with case of only values either 1 or 0
  unique_vals <- unique(c(df_binary$true, df_binary$predicted))
  if (all(unique_vals == 1) | all(unique_vals == 0)){
    return(list(binary_false_positive_rate = 0,
                binary_false_negative_rate = 0,
                classification_match_rate = match_rate,
                classification_fail_rate = fail_rate))
  } else {
    conf_matrix <- table(df_binary$true, df_binary$predicted)
  }

  # Extract values
  TP <- conf_matrix[2, 2]
  TN <- conf_matrix[1, 1]
  FP <- conf_matrix[1, 2]
  FN <- conf_matrix[2, 1]

  # Compute Metrics
  type1_error <- ifelse((FP + TN) == 0, 0, FP / (FP + TN))
  type2_error <- ifelse((FN + TP) == 0, 0, FN / (FN + TP))


  return(list(binary_false_positive_rate = type1_error,
              binary_false_negative_rate = type2_error,
              classification_match_rate = match_rate,
              classification_fail_rate = fail_rate))
}
```

Classification Pipeline

The classification_pipeline function performs a full start to finish processes for classification. To do so, it processes training and test data for classification, optionally applying PCA or FDA for dimensionality reduction. It replaces IDs in the test data that are not present in the training data with "0", estimates classification thresholds, and runs a KNN classifier. The function then calculates and returns classification results, including match rates and binary classification error rates. This pipeline provides a comprehensive workflow for preparing data, running classification, and evaluating performance.

```
### Pipeline
classification_pipeline <- function(train_data,
                                    test_data,
                                    estimate_func,
                                    percentile,
                                    knn_k,
                                    dist_metric,
                                    expl_var_pca = 0.95,
                                    expl_var_fda = 0.95,
                                    knn_bool = T,
                                    pca_bool = F,
                                    fda_bool =F) {

  # Replace IDs in test_data that are not in train_data with "0"
  train_ids <- train_data[, ncol(train_data)]
  test_ids <- test_data[, ncol(test_data)]

  test_data[, ncol(test_data)][!test_ids %in% train_ids] <- 0
  test_ids <- test_data[, ncol(test_data)]

  # run pca if required
  if (pca_bool){
    print("######## Run PCA ########")

    # Run PCA
    pca_result <- PCA.fun(train_data, matrix_bool = T)


    # Get results and get PC's that explain 95% Variance
    eig_vecs <- pca_result$"Eigen Vector"
    cumulative_variance <- cumsum(pca_result$D)
    num_components_pca <- which(cumulative_variance >= expl_var_pca)[1]

    # Project data onto selected PCs
    train_data <- as.matrix(train_data[,-ncol(train_data)] %*% eig_vecs[, 1:num_components_pca])
    test_data <- as.matrix(test_data[,-ncol(test_data)] %*% eig_vecs[, 1:num_components_pca])

    train_data = cbind(train_data, train_ids)
    test_data = cbind(test_data, test_ids)

  }

  if (fda_bool){
    print("######## Run FDA ########")
```

```
  # run fda
  result_FDA = FDA.fun(train_data[,-ncol(train_data)],train_ids)

  # select discriminants according to explained variance
  cumulative_variance_FDA <- cumsum(result_FDA$D)
  num_components_fda <- which(cumulative_variance_FDA >= expl_var_fda)[1]

  train_data = as.matrix(train_data[,-ncol(train_data)]) %*% result_FDA$P[, 1:num_components_fda]
  test_data = as.matrix(test_data[,-ncol(test_data)]) %*% result_FDA$P[, 1:num_components_fda]

  train_data = cbind(train_data, train_ids)
  test_data = cbind(test_data, test_ids)

}

print("######## Estimate Threshholds ########")

# Estimate thresholds
thresholds <- estimate_thresholds(train_data, estimate_func = estimate_func, percentile = percentile,

# Prepare test data
true_test_ids <- c(test_data[, ncol(test_data), drop = FALSE])
test_data_input <- test_data[, -ncol(test_data), drop = FALSE]

print("######## Run Classification########")

# Run KNN classifier
predicted_ids <- knn_classifier(test_data_input, train_data, thresholds = thresholds, dist_metric = di


# Merge results
df_classification <- data.frame(
  true = as.numeric(true_test_ids),
  predicted = as.numeric(predicted_ids)
)

# Derive binary classification [0,1] (if in data set)
df_classification_binary <- as.data.frame(lapply(df_classification, function(x) ifelse(x == 0, 0, 1)))


#print(df_classification)

# Calculate score
scores <- score(df_id = df_classification, df_binary = df_classification_binary)

print("######## Classification DONE ########")

# Return results
return(list(
  classification = df_classification,
  classification_binary = df_classification_binary,
  binary_false_positive_rate = scores$binary_false_positive_rate,
  binary_false_negative_rate = scores$binary_false_negative_rate,
```

```
    classification_match_rate = scores$classification_match_rate,
    classification_fail_rate = scores$classification_fail_rate
  ))
}
```

K-Fold Cross Validation

The knn_k_fold_cv function performs K-fold cross-validation on a dataset using a KNN classifier, with optional PCA or FDA for dimensionality reduction. It splits the data into training and testing sets for each fold, runs the classification pipeline, and stores the results for each fold. It returns the average scores across all folds.

```
### K fold Cross Validation
knn_k_fold_cv <- function(data,
                          folds,
                          k_CV,
                          estimate_func,
                          percentile,
                          knn_k,
                          dist_metric,
                          expl_var_pca = 0.95,
                          expl_var_fda = 0.95,
                          knn_bool,
                          pca_bool = F,
                          fda_bool = F,
                          print_bool = F) {


  all_results <- list()
  # Perform K-fold cross-validation
  for(i in 1:k_CV) {

    # Split data into training and testing sets based on the fold
    train_data <- data[folds != i, ]
    test_data <- data[folds == i, ]

    # Run the classification pipeline
    results <- classification_pipeline(train_data,
                                       test_data,
                                       estimate_func = estimate_func,
                                       percentile = percentile,
                                       knn_k = knn_k ,
                                       dist_metric = dist_metric,
                                       expl_var_pca,
                                       expl_var_fda,
                                       knn_bool,
                                       pca_bool,
                                       fda_bool)


    # Store results for this fold
    all_results[[i]] <- list(
      binary_false_positive_rate = results$binary_false_positive_rate,
```

```
      binary_false_negative_rate = results$binary_false_negative_rate,
      classification_match_rate = results$classification_match_rate,
      classification_fail_rate = results$classification_fail_rate
    )
  }

  # Store means
  binary_false_positive_rate_mean <- mean(sapply(all_results, function(x) x$binary_false_positive_rate))
  binary_false_negative_rate_mean <- mean(sapply(all_results, function(x) x$binary_false_negative_rate))
  classification_match_rate_mean <- mean(sapply(all_results, function(x) x$classification_match_rate))
  classification_fail_rate_mean <- mean(sapply(all_results, function(x) x$classification_fail_rate))

  avg_scores = list(binary_false_positive_rate_mean = binary_false_positive_rate_mean,
                    binary_false_negative_rate_mean = binary_false_negative_rate_mean,
                    classification_match_rate_mean = classification_match_rate_mean,
                    classification_fail_rate_mean = classification_fail_rate_mean)

  # Print all average metrics
  if (print_bool){
    cat("Average scores across all folds:")
    cat(avg_scores)
  }

  return(avg_scores)#, scores_by_fold=all_results))
}
```

KNN Grid Search Optimization

The CV_grid_search function performs a grid search over specified hyperparameters for a KNN classifier, using K-fold cross-validation to evaluate each combination. It iterates over different values of knn_k, percentile, and dist_metric, storing the average performance metrics for each combination. Finally, it identifies and prints the best hyperparameters based on classification match rate and binary false negative rate, returning the results as a data frame along with the best parameters.The parameters correspond to different parts of the classification process, e.g. percentile_values correspond to the chosen treshold (e.g binary classification) and k_knn to the id based classification, while the distance metric affects both. Therefore, they either use the binary classification scores or the id match rate as metrics for the optimization.

```
CV_grid_search <- function(data,
                           knn_k_values,
                           percentile_values,
                           dist_metrics,
                           expl_var_pca = 0.95,
                           expl_var_fda = 0.95,
                           k_CV = 5,
                           knn_bool = T,
                           pca_bool = F,
                           fda_bool = F) {
  # Store results
  results_list <- list()

  folds <- sample(1:k_CV, nrow(data), replace = TRUE)
```

```r
# Perform grid search
for (knn_k in knn_k_values) {
  for (percentile in percentile_values) {
    for (dist_metric in dist_metrics) {
          # Run cross-validation
      print(paste(knn_k,percentile,dist_metric))
          avg_scores <- knn_k_fold_cv(data,
                                      folds,
                                      k_CV = k_CV,
                                      estimate_func = estimate_threshold_via_percentile,
                                      percentile = percentile,
                                      knn_k = knn_k,
                                      dist_metric = dist_metric,
                                      expl_var_pca = expl_var_pca,
                                      expl_var_fda = expl_var_fda,
                                      knn_bool,
                                      pca_bool,
                                      fda_bool)
          print(avg_scores)

          # Store results
          results_list[[paste(knn_k, percentile, dist_metric, sep="_")]] <- list(
            knn_k = knn_k,
            percentile = percentile,
            dist_metric = dist_metric,
            expl_var_pca = expl_var_pca,
            expl_var_fda = expl_var_fda,
            binary_false_positive_rate_mean = avg_scores$binary_false_positive_rate_mean,
            binary_false_negative_rate_mean = avg_scores$binary_false_negative_rate_mean,
            classification_match_rate_mean = avg_scores$classification_match_rate_mean,
            classification_fail_rate_mean = avg_scores$classification_fail_rate_mean
          )
    }
  }
  }

# Convert results to data frame
results_df <- do.call(rbind, lapply(results_list, as.data.frame))

# Find the best parameters for threshold and for knn
best_params_knn <- results_df[which.max(results_df$classification_match_rate_mean), ]
best_params_binary <- results_df[which.min(results_df$binary_false_negative_rate_mean), ]

# Print best hyperparameters in a table
print("##################### Best Hyperparameters ####################")
best_params <- data.frame(
  Metric = c("Distance Metric", "Percentile for Quantile Threshold", "KNN K nearest neighbors"),
  Value = c(best_params_knn$dist_metric, best_params_binary$percentile, best_params_knn$knn_k)
)

table_with_best_params = knitr::kable(best_params, col.names = c("Parameter", "Value"), caption = "Bes

return(list(results_df = results_df,
```

```
              table_with_best_params = table_with_best_params))
}
```

Start-to-finish Classification

The "knn_classifier_full" function is a container for the full classification pipeline with the added step of loading train and test data added based on path to corresponding directories. It is designed for the final application of the classification to test or new data.

```
### Predict for new data
knn_classifier_full <- function(path_train,
                            path_test,
                            dist_metric,
                            knn_k,
                            result,
                            estimate_func = estimate_threshold_via_percentile,
                            percentile = percentile,
                            expl_var_pca,
                            expl_var_fda,
                            pca_bool,
                            fda_bool) {



  print("######## Read Data ########")
  data_train = read_all_images(path_train)
  data_test = read_all_images(path_test)

  image_names_test = data_test$ID

  data_train_labels <- data_train$Label
  data_test_labels <- data_test$Label

  data_train = data_train %>% dplyr::select(-ID)
  data_test = data_test %>% dplyr::select(-ID)

  data_train = as.matrix(data_train)
  data_test = as.matrix(data_test)

  data_train = apply(data_train, 2, as.numeric)
  data_test = apply(data_test, 2, as.numeric)


  print("######## Start Classification Pipeline ########")
  classification_result = classification_pipeline(train_data = data_train,
                                      test_data = data_test,
                                      estimate_func = estimate_func,
                                      percentile = percentile,
                                      knn_k = knn_k,
                                      dist_metric = dist_metric,
                                      expl_var_pca = expl_var_pca,
                                      expl_var_fda = expl_var_pca,
                                      pca_bool = pca_bool,
```

```
                                    fda_bool = fda_bool)

  classification_result$classification = cbind(image_names_test,classification_result$classification)
  classification_result$classification_binary = cbind(image_names_test,classification_result$classifica


  # Return result
  return(classification_result)
}
```

Run Codes

The following section now contains the excecution of the previous codes leading towards optimal classification.

Load Data

We start by loading the data and doing some basic data manipulation.

```
### load data
folder_path = "Training"
data = read_all_images(folder_path)

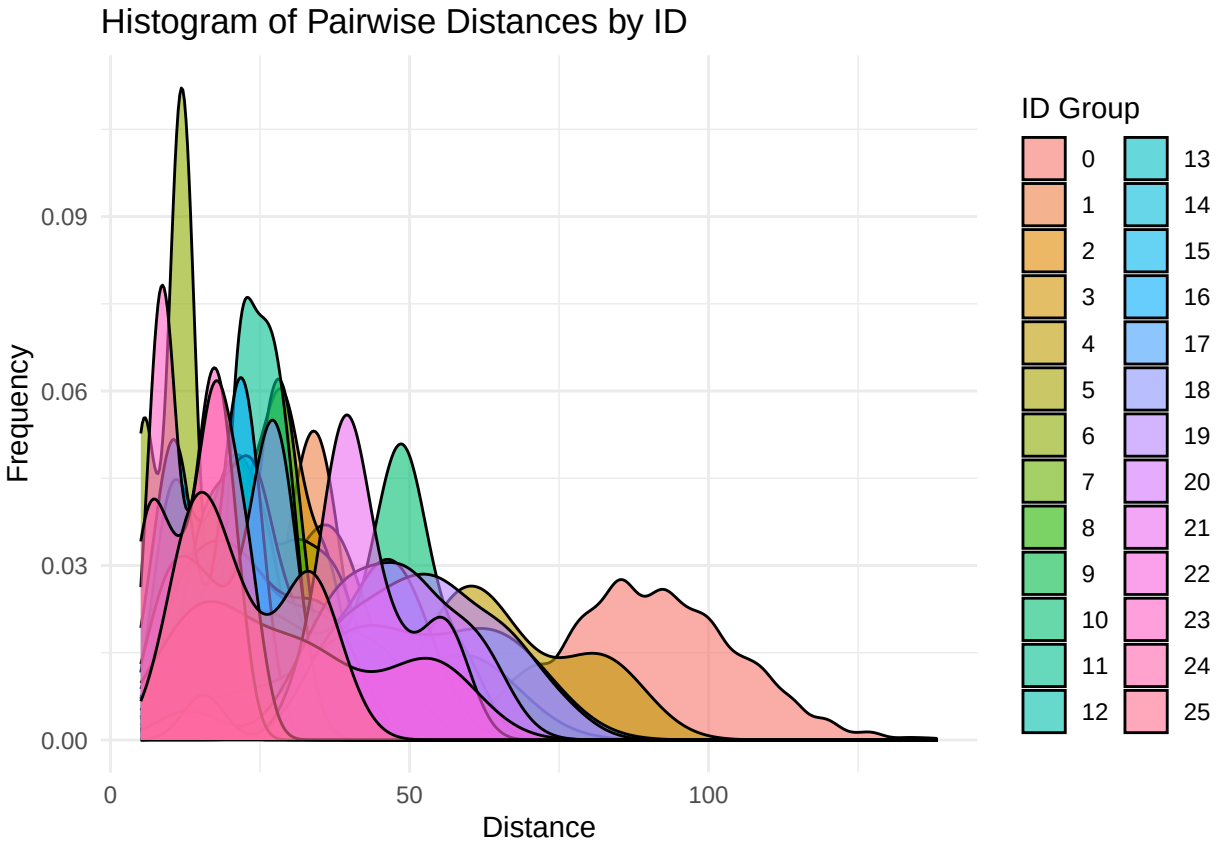
labels <- data$Label
image_names = data$ID

# turn into matrix for some application
data_asmatrix  = as.matrix(data %>% dplyr::select(-ID))
data_asmatrix <- apply(data_asmatrix, 2, as.numeric)
```

Classification on Raw Data (Part-A-1)

We start out by attempting to classify on the raw data. To get a feeling for the distances between the images of the same person, we plot the KDE of the distributions of the pairwise distances for both within-class-distance (ID's 1-m) and the outside of class distances (ID=0), e.g. distances between all elements in one class with all elements of all other classes but not with elements from its own class. The result shows a somewhat well divide between Out-of-class distances and In-Class Distances, with however some In-Class Distances moving in similar distance value range as the Out-of-Class Distances. This shows that the binary classification will probably suffer from some missclassifications.

```
distance_distributions(data_asmatrix, distance_metric = "euclidean")
```



With no other Steps required, we directly run the classification algorithm, using the implemented Cross-Validation grid search algorithm to first find optimal hyperparameters. According to the Cross Validation, the euclidean distance with a Quantile Threshhold of .95 and 3 nearest neighbors are optimal.

Run grid Search to optimize Hyperparameters

WARNING: This code takes approx 20 min to excecute

CV_grid_search_result_RAW <- CV_grid_search(data_asmatrix, k_CV = 5, knn_k_values = c(3,4,5), percentile_values = c(0.8,0.9, 0.95), dist_metrics = c("euclidean", "manhattan"), knn_bool = T)

print(CV_grid_search_result_RAW$table_with_best_params)

```
Having found the optimal hyperparameters, we rerun the crossvalidation to get the expected performance u
```

```
``` r
# Define folds for this and all other Cross falidations
set.seed(123)
k_CV = 5
folds <- sample(1:k_CV, nrow(data_asmatrix), replace = TRUE)

### Run CV for optimal parameters to get scores
avg_scores_CV_raw <- knn_k_fold_cv(data_asmatrix,
```
```

```
                            folds,
                            k_CV = k_CV,
                            estimate_func = estimate_threshold_via_percentile,
                            percentile = 0.9,
                            knn_k = 3,
                            dist_metric = "euclidean",
                            knn_bool = T,
                            pca_bool = F,
                            fda_bool = F)
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
avg_scores_CV_raw
```

```
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.07214127
##
## $classification_match_rate_mean
## [1] 0.9064302
##
## $classification_fail_rate_mean
## [1] 0.09356984
```

Classification on PCA-Data (Part-A-2)

To now perform classification on the dimension reduced Data via PCA, we first need to apply the PCA. We start by applying the PCA function defined in the previous chapter and validate the results using the base R prcomp function.

If we compare the PCA function we created with prcomp(), we can see a significant difference in execution time and system resource usage. This is because the base function assumes that $n >> p$, but for the type of data we are working with, this approach is less optimal. Instead, we use a function that is optimized for cases where $n << p$.

```
### Run manual and baseR PCA
print(system.time(PCA.fun(data)))
```

```
## [1]    150 108000
##    user  system elapsed
##    4.17    0.14    4.38
```

```
PCA_result = PCA.fun(data)
```

```
## [1]    150 108000
```

```
print(system.time(prcomp(data_asmatrix[,-ncol(data_asmatrix)],scale. = T)))
```

```
##    user  system elapsed
##    8.24    0.27    8.66
```

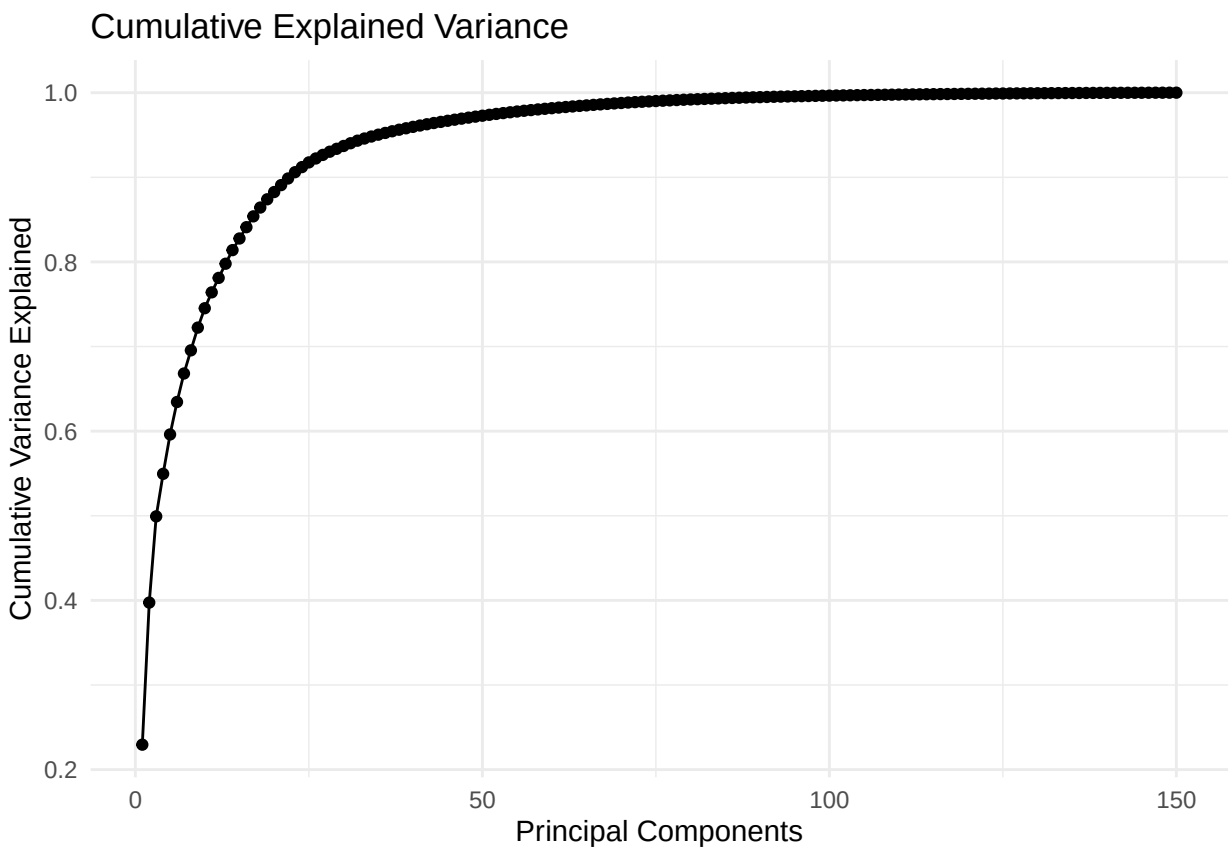
```
compare_pc = prcomp(data_asmatrix[,-ncol(data_asmatrix)],scale. = T)
print(cumsum(compare_pc$sdev^2)/sum(compare_pc$sdev^2))
```

```
##   [1] 0.2608522 0.3843942 0.4665586 0.5178641 0.5636818 0.6045098 0.6365057
##   [8] 0.6642047 0.6914699 0.7169550 0.7378602 0.7570985 0.7749734 0.7912632
##  [15] 0.8067908 0.8213244 0.8341660 0.8459340 0.8563443 0.8664791 0.8758631
##  [22] 0.8848331 0.8927398 0.8993429 0.9049319 0.9098999 0.9146311 0.9189731
##  [29] 0.9229389 0.9266306 0.9299970 0.9332179 0.9360703 0.9387086 0.9411027
##  [36] 0.9434049 0.9455344 0.9476028 0.9495309 0.9513992 0.9531669 0.9549078
##  [43] 0.9565900 0.9581321 0.9596145 0.9610585 0.9624926 0.9638430 0.9651727
##  [50] 0.9664579 0.9677117 0.9688997 0.9700328 0.9711545 0.9722388 0.9732662
##  [57] 0.9742727 0.9751982 0.9760971 0.9769652 0.9778233 0.9786458 0.9794598
##  [64] 0.9802297 0.9809697 0.9816871 0.9823754 0.9830380 0.9836902 0.9843200
##  [71] 0.9849337 0.9855210 0.9860720 0.9865911 0.9870792 0.9875598 0.9880108
##  [78] 0.9884492 0.9888750 0.9892813 0.9896767 0.9900604 0.9904409 0.9908031
##  [85] 0.9911607 0.9915109 0.9918469 0.9921724 0.9924938 0.9928098 0.9931026
##  [92] 0.9933665 0.9936235 0.9938646 0.9940969 0.9943197 0.9945375 0.9947490
##  [99] 0.9949515 0.9951492 0.9953400 0.9955284 0.9957065 0.9958826 0.9960559
## [106] 0.9962226 0.9963880 0.9965522 0.9967079 0.9968614 0.9970060 0.9971502
## [113] 0.9972882 0.9974169 0.9975396 0.9976622 0.9977767 0.9978910 0.9980036
## [120] 0.9981123 0.9982197 0.9983258 0.9984249 0.9985230 0.9986173 0.9987087
## [127] 0.9987978 0.9988844 0.9989681 0.9990509 0.9991309 0.9992104 0.9992828
## [134] 0.9993471 0.9994093 0.9994686 0.9995274 0.9995821 0.9996348 0.9996851
## [141] 0.9997303 0.9997710 0.9998099 0.9998467 0.9998816 0.9999150 0.9999457
## [148] 0.9999733 1.0000000 1.0000000
```

When comparing the eigenvalues, our custom function required fewer eigenvalues to achieve the same explained variance. In the graph you can see that if we want to get close to 90 of the variance of the data we only need to get 25 principal components.

```
df <- data.frame(
  Components = 1:length(PCA_result$D),
  CumulativeVariance = cumsum(PCA_result$D)
)
```

```r
# Plot the cumulative explained variance
ggplot(df, aes(x = Components, y = CumulativeVariance)) +
  geom_line() +
  geom_point() +
  labs(title = "Cumulative Explained Variance",
       x = "Principal Components",
       y = "Cumulative Variance Explained") +
  theme_minimal()
```

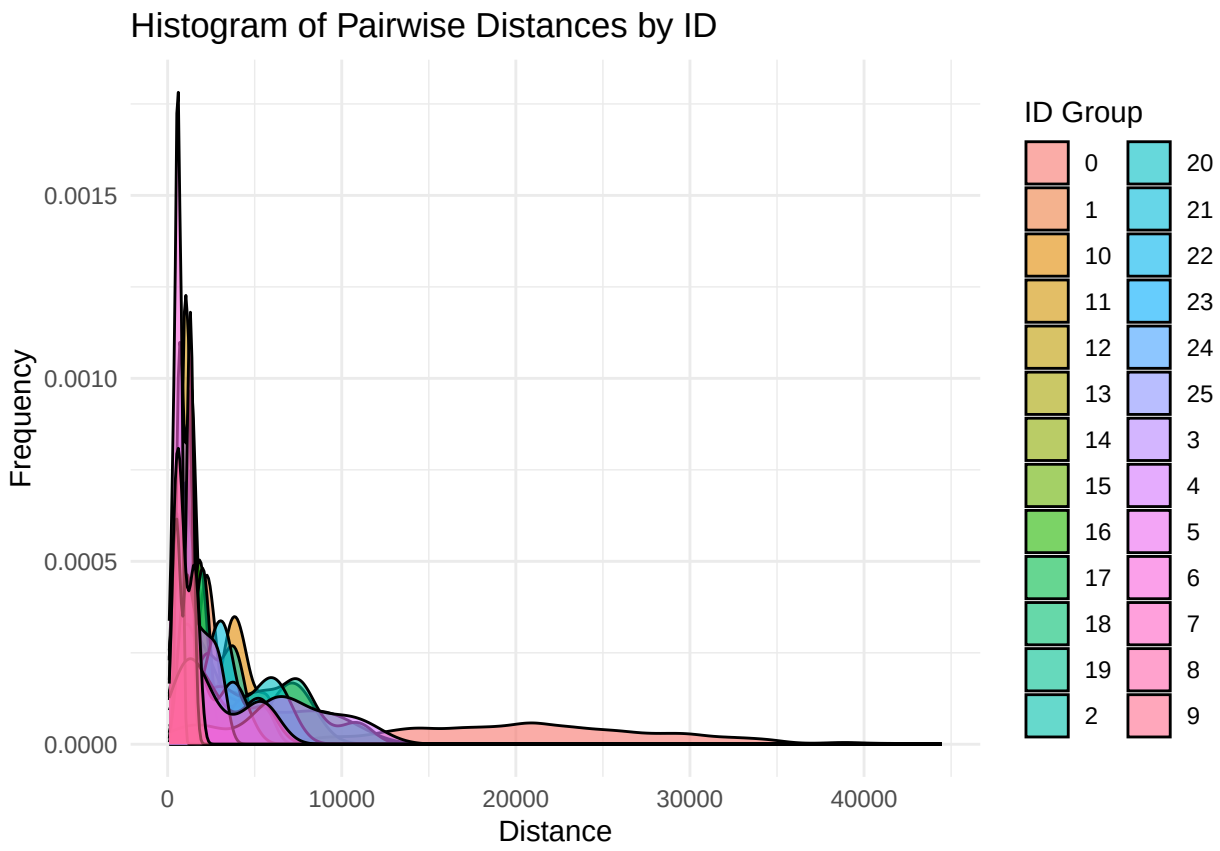


With the PCA successfully performed, corresponding to the plots shown above, we decide to fix 90 percent explained Variance as hyper parameter for the PCA and apply perform the projection of the data with the identified Principal Components.

```r
# Get results and get PC's that explain 95% Variance
eig_vecs <- PCA_result$"Eigen Vector"
cumulative_variance <- cumsum(PCA_result$D)
num_components <- which(cumulative_variance >= 0.95)[1]

# Project data onto selected PCs
data_reduced_PCA <- data_asmatrix[,-ncol(data_asmatrix)] %*% eig_vecs[, 1:num_components]
data_reduced_PCA_labeled = cbind(data_reduced_PCA, labels)
```

Using the PDA-reduced new data we can again plot the distance distributions and now see how the area of overlap between in-class-distances and out-of-class distances was already decreased by the PCA, optentially improving classification results.

```
distance_distributions(data_reduced_PCA_labeled, distance_metric = "euclidean")
```



To then proceed with classification, we again identify the optimal hyperparameters using Cross Validation Grid search. The result show that againthe euclidean distance with a Quantile Threshhold of .95 and 3 nearest neighbors are optimal.

```
### Run grid Search to optimize Hyperparameters
CV_grid_search_result_PCA <- CV_grid_search(data_reduced_PCA_labeled,
                                           k_CV = k_CV,
                                           knn_k_values = c(3, 4, 5),
                                           percentile_values = c(0.8, 0.9, 0.95),
                                           dist_metrics = c("euclidean", "manhattan"))
```

```
## [1] "3 0.8 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04888889
##
## $classification_match_rate_mean
## [1] 0.9309869
##
## $classification_fail_rate_mean
## [1] 0.06901311
##
## [1] "3 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.9167357
##
## $classification_fail_rate_mean
## [1] 0.08326432
##
## [1] "3 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
```

```
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03777778
##
## $classification_match_rate_mean
## [1] 0.942098
##
## $classification_fail_rate_mean
## [1] 0.057902
##
## [1] "3 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.9167357
##
## $classification_fail_rate_mean
## [1] 0.08326432
##
## [1] "3 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
```

```
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03222222
##
## $classification_match_rate_mean
## [1] 0.9476536
##
## $classification_fail_rate_mean
## [1] 0.05234645
##
## [1] "3 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05202899
##
## $classification_match_rate_mean
## [1] 0.9278468
##
## $classification_fail_rate_mean
## [1] 0.07215321
##
## [1] "4 0.8 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04888889
##
## $classification_match_rate_mean
## [1] 0.9082885
##
## $classification_fail_rate_mean
## [1] 0.09171153
##
## [1] "4 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.9053071
##
## $classification_fail_rate_mean
## [1] 0.09469289
##
## [1] "4 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
```

```
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03777778
##
## $classification_match_rate_mean
## [1] 0.9193996
##
## $classification_fail_rate_mean
## [1] 0.08060041
##
## [1] "4 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.9053071
##
## $classification_fail_rate_mean
## [1] 0.09469289
##
## [1] "4 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
```

```
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03222222
##
## $classification_match_rate_mean
## [1] 0.9249551
##
## $classification_fail_rate_mean
## [1] 0.07504486
##
## [1] "4 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05202899
##
## $classification_match_rate_mean
## [1] 0.9164182
##
## $classification_fail_rate_mean
## [1] 0.08358178
##
## [1] "5 0.8 euclidean"
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04888889
##
## $classification_match_rate_mean
## [1] 0.8995928
##
## $classification_fail_rate_mean
## [1] 0.1004072
##
## [1] "5 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.8536991
##
## $classification_fail_rate_mean
## [1] 0.1463009
```

```
##
## [1] "5 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03777778
##
## $classification_match_rate_mean
## [1] 0.9107039
##
## $classification_fail_rate_mean
## [1] 0.08929607
##
## [1] "5 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0631401
##
## $classification_match_rate_mean
## [1] 0.8536991
##
```

```
## $classification_fail_rate_mean
## [1] 0.1463009
##
## [1] "5 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.03222222
##
## $classification_match_rate_mean
## [1] 0.9162595
##
## $classification_fail_rate_mean
## [1] 0.08374051
##
## [1] "5 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05202899
##
## $classification_match_rate_mean
```

```
## [1] 0.8648102
##
## $classification_fail_rate_mean
## [1] 0.1351898
##
## [1] "##################### Best Hyperparameters ####################"
```

```
print(CV_grid_search_result_PCA$table_with_best_params)
```

```
##
##
## Table: Best Hyperparameters
##
## |Parameter                        |Value     |
## |:--------------------------------|:---------|
## |Distance Metric                  |euclidean |
## |Percentile for Quantile Threshold |0.95      |
## |KNN K nearest neighbors          |3         |
```

And also, like previous, we perform a last CV with the as optimal determined parameters.

```
### Run CV for optimal parameters to get scores
avg_scores_CV_PCA <- knn_k_fold_cv(data_reduced_PCA_labeled,
                            folds,
                            k_CV = k_CV,
                            estimate_func = estimate_threshold_via_percentile,
                            percentile = 0.9,
                            knn_k = 3,
                            dist_metric = "euclidean",
                            knn_bool = T,
                            pca_bool = F,
                            fda_bool = F)
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
avg_scores_CV_PCA
```

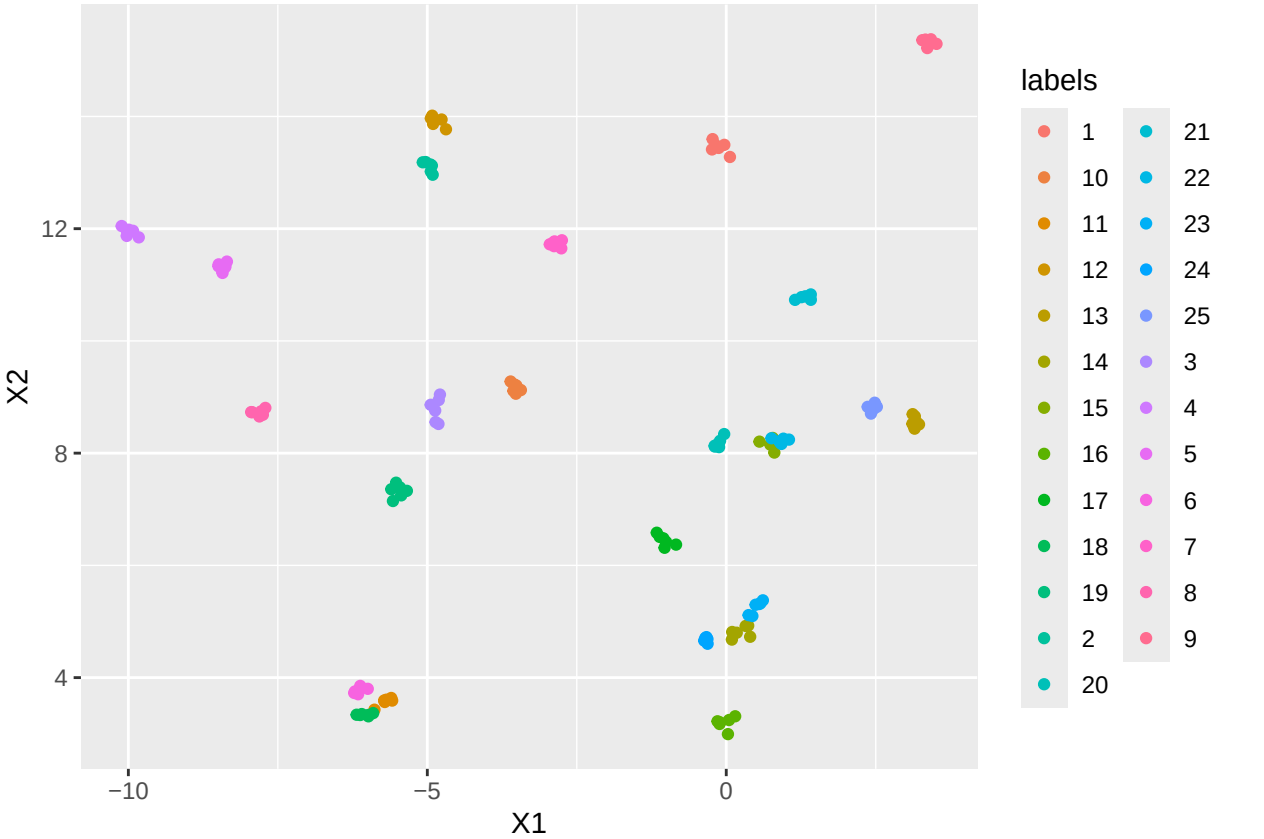
```
## $binary_false_positive_rate_mean
```

```
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.0745587
##
## $classification_match_rate_mean
## [1] 0.9182984
##
## $classification_fail_rate_mean
## [1] 0.08170156
```

Classification on FDA + PCA Data (Part B)

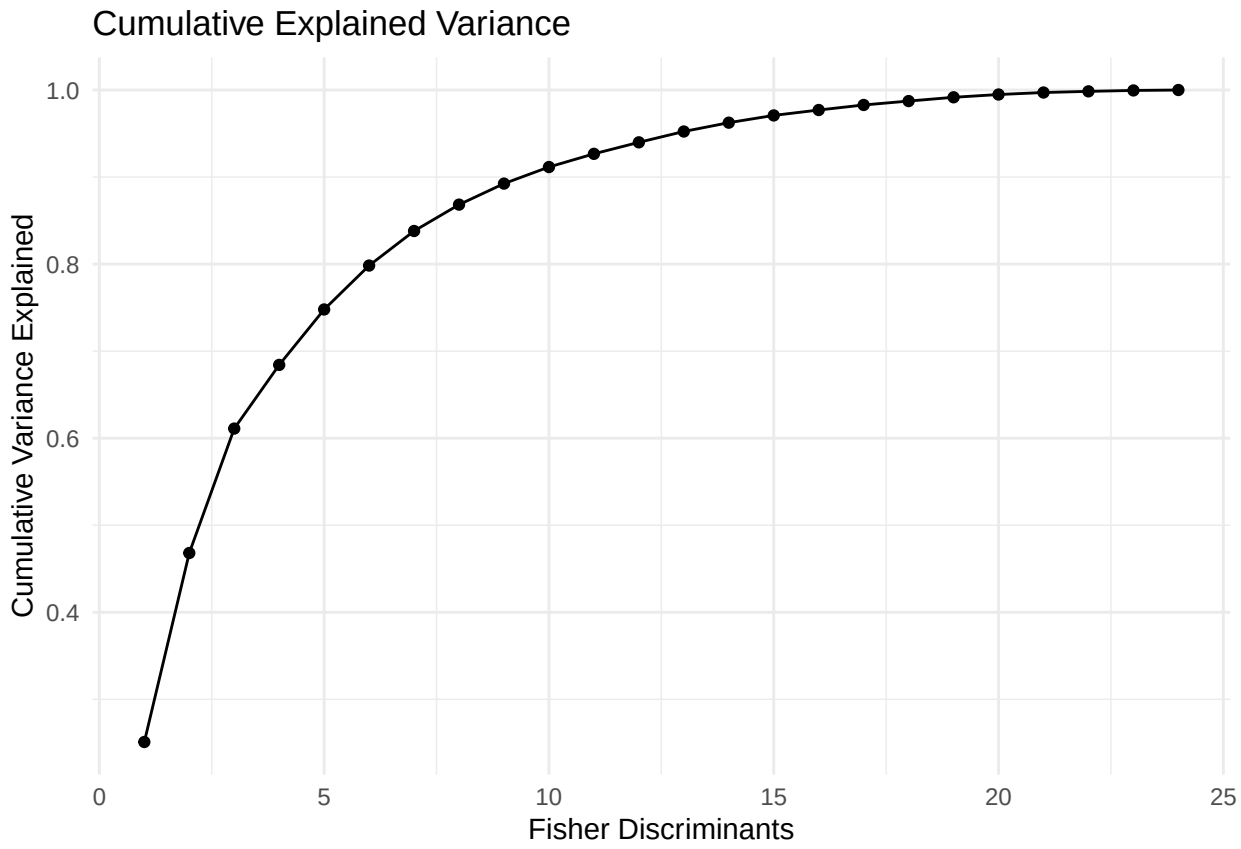
As last variation in this assignment, we apply Fisher Discriminant Analysis to the already with the PCA reduced Data. We use the function defined above and apply it to the PCA reduced data. The output of the two first dimensions can be plotted as shown in the following chunk, already giving a nice separation between classes in most cases

```
result_FDA = FDA.fun(data_reduced_PCA,labels)

data_reduced_FDA <-result_FDA$transform_data
df_plot <- data.frame(result_FDA$transform_data, labels = result_FDA$labels)
ggplot(df_plot, aes(x = X1, y = X2, color = labels)) + geom_point()
```



Taking the first 10 fisher discriminate you can get 90 of the reduce (PCA) matrix variability explain as seen below in the comulative variance plot.

```r
df_FDA <- data.frame(
  Components = 1:length(result_FDA$D),
  CumulativeVariance = cumsum(result_FDA$D)
)

# Plot the cumulative explained variance
ggplot(df_FDA, aes(x = Components, y = CumulativeVariance)) +
  geom_line() +
  geom_point() +
  labs(title = "Cumulative Explained Variance",
       x = "Fisher Discriminants",
       y = "Cumulative Variance Explained") +
  theme_minimal()
```



Looking at the performance of the FDA, we find the MASS package based implementation to be more efficient.

```r
system.time(FDA.fun(data_reduced_PCA,labels))
```

```
##    user  system elapsed
##       0       0       0
```

```r
system.time( MASS::lda(data_reduced_PCA,labels))
```

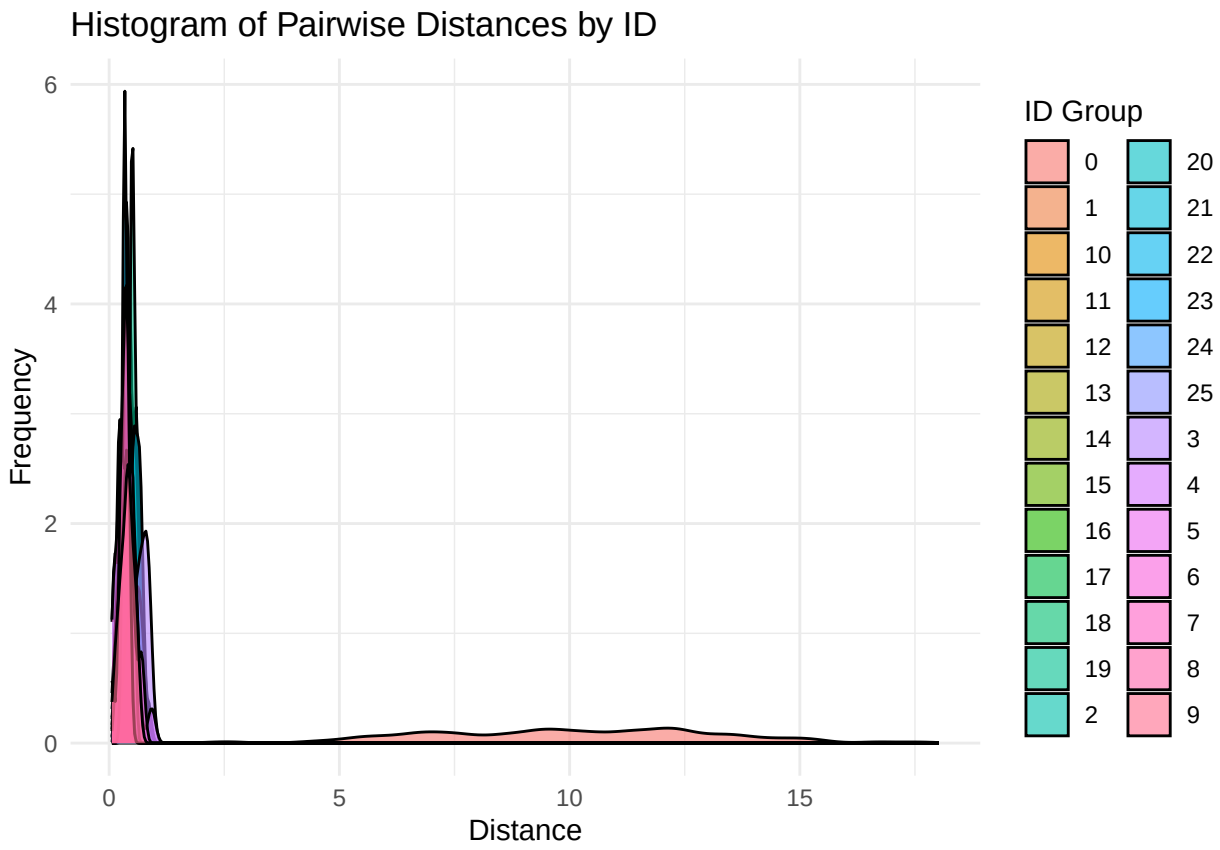
```
##    user  system elapsed
##    0.00    0.00    0.02
```

With the FDA performed successfully, we again choose 95 of explained Variance as hyperparameter and proceed with the further dimension reduced data.

```
# select discriminants according to explained variance
cumulative_variance_FDA <- cumsum(result_FDA$D)
num_components_fda <- which(cumulative_variance_FDA >= 0.95)[1]

data_reduced_fda = result_FDA$transform_data[,1:num_components_fda]
data_reduced_fda_labeld = cbind(data_reduced_fda,labels)
```

We again can plot the distribtions of the distances distances as shown below. As is very apparent, there is now a near complete separation between In-Class-Distances and Out-Of-Class Distances (ID=0).

```
distance_distributions(data_reduced_fda_labeld, distance_metric = "euclidean")
```



And to find optimal hyperparameters, we proceed with CV grid search to again find the euclidean distance with a Quantile Threshhold of .95 and 3 nearest neighbors optimal.

```
### Run grid Search to optimize Hyperparameters
CV_grid_search_result_FDA <- CV_grid_search(data_reduced_fda_labeld,
                                            k_CV = k_CV,
                                            knn_k_values = c(3, 4, 5),
                                            percentile_values = c(0.8, 0.9, 0.95),
                                            dist_metrics = c("euclidean", "manhattan"))
```

```
## [1] "3 0.8 euclidean"
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
##
## [1] "3 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
```

```
##
## [1] "3 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "3 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05957105
##
## $classification_match_rate_mean
## [1] 0.9404289
##
```

```
## $classification_fail_rate_mean
## [1] 0.05957105
##
## [1] "3 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "3 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05157105
##
## $classification_match_rate_mean
```

```
## [1] 0.9484289
##
## $classification_fail_rate_mean
## [1] 0.05157105
##
## [1] "4 0.8 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
##
## [1] "4 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
```

```
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
##
## [1] "4 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "4 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
```

```
## $binary_false_negative_rate_mean
## [1] 0.05957105
##
## $classification_match_rate_mean
## [1] 0.9404289
##
## $classification_fail_rate_mean
## [1] 0.05957105
##
## [1] "4 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "4 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
```

```
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05157105
##
## $classification_match_rate_mean
## [1] 0.9484289
##
## $classification_fail_rate_mean
## [1] 0.05157105
##
## [1] "5 0.8 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
##
## [1] "5 0.8 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
```

```
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.06497646
##
## $classification_match_rate_mean
## [1] 0.9350235
##
## $classification_fail_rate_mean
## [1] 0.06497646
##
## [1] "5 0.9 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "5 0.9 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05957105
##
## $classification_match_rate_mean
## [1] 0.9404289
##
## $classification_fail_rate_mean
## [1] 0.05957105
##
## [1] "5 0.95 euclidean"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.04616565
##
## $classification_match_rate_mean
## [1] 0.9538344
##
## $classification_fail_rate_mean
## [1] 0.04616565
##
## [1] "5 0.95 manhattan"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
```

```
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.05157105
##
## $classification_match_rate_mean
## [1] 0.9484289
##
## $classification_fail_rate_mean
## [1] 0.05157105
##
## [1] "#################### Best Hyperparameters ####################"
```

```
print(CV_grid_search_result_FDA$table_with_best_params)
```

```
##
##
## Table: Best Hyperparameters
##
## |Parameter                          |Value     |
## |:----------------------------------|:---------|
## |Distance Metric                    |euclidean |
## |Percentile for Quantile Threshold |0.9       |
## |KNN K nearest neighbors            |3         |
```

```
avg_scores_CV_FDA <- knn_k_fold_cv(data_reduced_fda_labeld,
                            folds,
                            k_CV = k_CV,
                            estimate_func = estimate_threshold_via_percentile,
                            percentile = 0.95,
                            knn_k = 3,
                            dist_metric = "euclidean",
                            knn_bool = T,
                            pca_bool = F,
                            fda_bool = F)
```

```
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
```

```
## [1] "######## Classification DONE ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
avg_scores_CV_FDA
```

```
## $binary_false_positive_rate_mean
## [1] 0
##
## $binary_false_negative_rate_mean
## [1] 0.01915323
##
## $classification_match_rate_mean
## [1] 0.9808468
##
## $classification_fail_rate_mean
## [1] 0.01915323
```

Compare Results

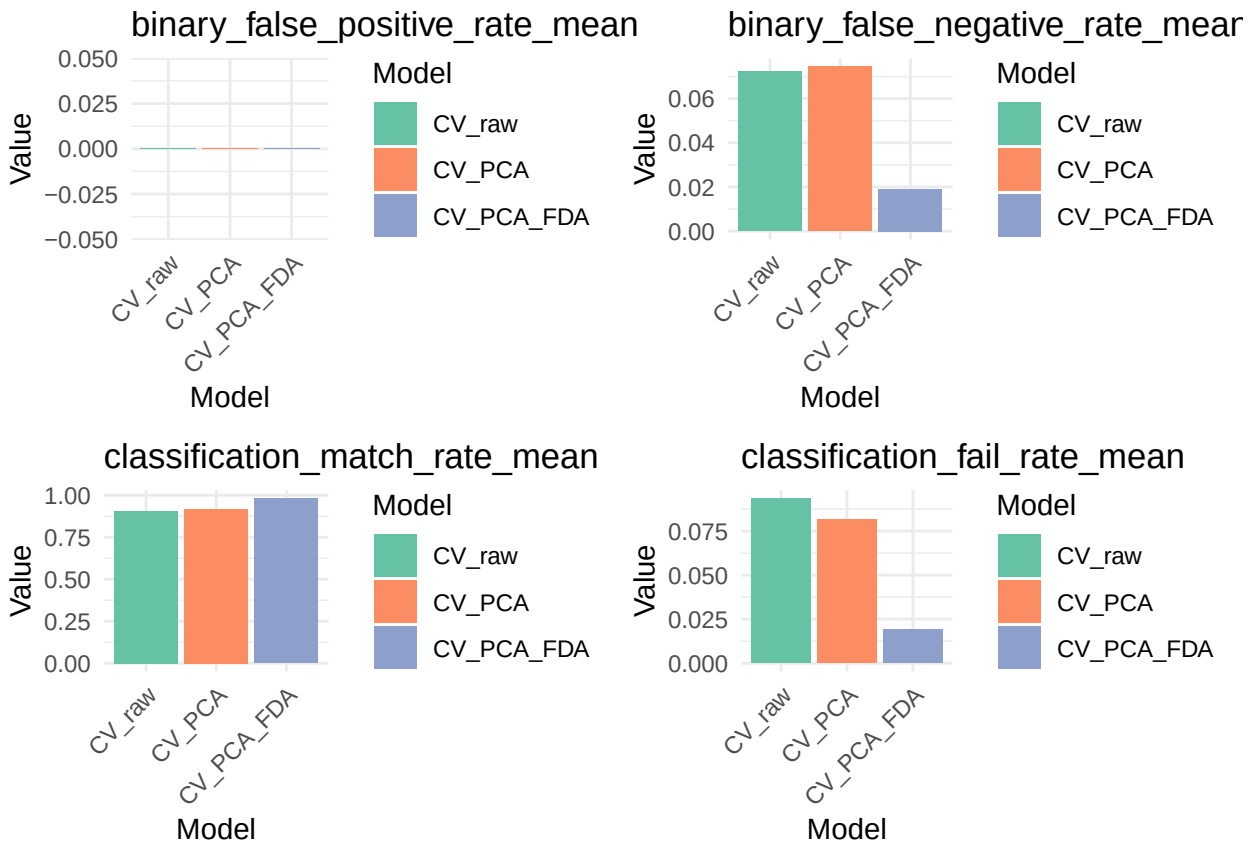
Comparing the three approaches, we find the FDA/PCA approach with hyperparameters distance metric Euclidian distance, Quantile Threshhold of .95, 3 nearest neighbors optimal to yield the best scores across all reviewed classification approaches. Looking at how well the data was clustered this is in line with what could be expected. Furthermore, both PCA and FDA/PCA approaches have significantly improved execution times compared to running classification purely on the raw data. The Pipeline PDA->FDA->Classification should thus be chosen for the classification task.

```
comparison_table <- data.frame(
  Metric = names(avg_scores_CV_raw),
  CV_raw = unlist(avg_scores_CV_raw),
  CV_PCA = unlist(avg_scores_CV_PCA),
  CV_PCA_FDA = unlist(avg_scores_CV_FDA)
)

# Print table
print(comparison_table)
```

```
##                                                       Metric     CV_raw
## binary_false_positive_rate_mean binary_false_positive_rate_mean 0.00000000
## binary_false_negative_rate_mean binary_false_negative_rate_mean 0.07214127
## classification_match_rate_mean   classification_match_rate_mean 0.90643016
## classification_fail_rate_mean     classification_fail_rate_mean 0.09356984
##                                      CV_PCA CV_PCA_FDA
## binary_false_positive_rate_mean 0.00000000 0.00000000
## binary_false_negative_rate_mean 0.07455870 0.01915323
## classification_match_rate_mean  0.91829844 0.98084677
## classification_fail_rate_mean   0.08170156 0.01915323
```

```
# Melt for ggplot
df_melted <- melt(comparison_table, id.vars = "Metric", variable.name = "Model", value.name = "Value")
```

```r
# Create individual plots
plots <- lapply(unique(df_melted$Metric), function(metric) {
  ggplot(df_melted[df_melted$Metric == metric, ], aes(x = Model, y = Value, fill = Model)) +
    geom_bar(stat = "identity", position = "dodge") +
    ggtitle(metric) +
    theme_minimal() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    scale_fill_brewer(palette = "Set2")
})

# Arrange in a 2x2 grid
print(grid.arrange(grobs = plots, ncol = 2))
```




```
## TableGrob (2 x 2) "arrange": 4 grobs
##   z     cells    name           grob
## 1 1 (1-1,1-1) arrange gtable[layout]
## 2 2 (1-1,2-2) arrange gtable[layout]
## 3 3 (2-2,1-1) arrange gtable[layout]
## 4 4 (2-2,2-2) arrange gtable[layout]
```

Looking at these results, initially is surprising that the raw data performs similarly well compared to the PCA and FDA data. The plots are indeed misleading as the our Cross Validation only was performed in-sample. After investigation into these strange results we encountered a problem with the way our threshold is set. As we use the quantile of the distributions, the maximum possible treshold is the maximum observed distance for a given class. In the case of raw data, this is not as problematic as the in-class distances are closer to the inter-class distances and thus excessively moving the treshold beyond the maximum in-class distance would

not be wise. However, for the PCA data and even more so for the FDA data, the classes are better isolated (as can be seen in the plots above) and the magnitude between in-class distances and inter-class distances is large (as can be seen in the histograms). Thus making the threshold larger would significantly benefit the classification while not risking much downside due to accepting imposters. Our current setup unfortunately does not allow to do so which is why the resulting scores make it seem like all approaches are equally good. However with some more time to modify the way the threshold is set, the FDA is certain to outperform both PCA and raw-data based approaches.That is why in the code below we recommend to use the PCA/FDA based approach, having observed in experimenting this to perform better when identifying imposters while delivering solid classification results.

Run KNN for new Data with optimal hyperparametrs

RUN THIS FOR EVALUATION

The following code chunk contains a function specifically designed for seamless classification. The only required changes are to modify the paths to directories containing training and testing data. The remaining hyperparameters have been already set up according to the optimal settings found in the previous analysis.

```
### Predict for new Data
classification_result = knn_classifier_full( path_train = "Training_alpha", # <------------ Modify
                                path_test = "Test_alpha", # <----------------- Modify
                                dist_metric = 'euclidean',
                                knn_k = 3,
                                estimate_func = estimate_threshold_via_percentile,
                                percentile = 0.95,
                                expl_var_pca = 0.95,
                                expl_var_fda = 0.95,
                                pca_bool = T,
                                fda_bool = T)
```

```
## [1] "######## Read Data ########"
## [1] "######## Start Classification Pipeline ########"
## [1] "######## Run PCA ########"
## [1]     30 108000
## [1] "######## Run FDA ########"
## [1] "######## Estimate Threshholds ########"
## [1] "######## Run Classification########"
## [1] "######## Classification DONE ########"
```

```
classification_result$classification # Returns Dataframe with True vs Predicted
```

```
##   image_names_test true predicted
## 1          1ET.jpg    1         1
## 2          1FT.jpg    1         1
## 3          2AT.jpg    2         2
## 4          7BT.jpg    0         0
## 5         20BT.jpg    0         0
```

```
# Optional: Score
classification_scores = score(classification_result$classification,classification_result$classification_
classification_scores
```

```
## $binary_false_positive_rate
## [1] 0
##
## $binary_false_negative_rate
## [1] 0
##
## $classification_match_rate
## [1] 1
##
## $classification_fail_rate
## [1] 0
```