PCA, KNN and FDA implementation

Silvana Alvarez - Sergio Quintanilla

2025-03-12

```
#install.packages("OpenImageR")
library(OpenImageR)
library(dplyr)
```

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

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

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

```
library(ggplot2)
library(patchwork)
```

```
source("function_loading.R")
```

Part A

Implement a facial recognizer based on Principal Component Analysis.

Solution:

First we show how we create the DB, then in each section we show the functions that we made to create the PCA, KNN and tune the parameters. Finally, we display the application of all the steps using the tuned parameters.

```
# Files
Files = list.files(path="Training/")

# Labels
labels <- as.numeric(gsub("[^0-9]", "", Files))

# Data Base
X = matrix(NA, nrow=150, ncol = 108000)
```

```
for(i in seq_along(Files)){
  Im = readImage(paste0("Training/",Files[i]))
  ri=as.vector(Im[,,1])
  gi=as.vector(Im[,,2])
  bi=as.vector(Im[,,3])
  X[i,] = cbind(ri,gi,bi)
}
dim(X)
```

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

With the previous process we obtain a matrix of 150 rows (images), and 108.000 columns, that represents each pixel of the image, in each one of the RGB colors.

A)

Build a function that implements the Principal Component Analysis. This function takes as input a set of observations and returns the mean of these observations, the matrix P containing the eigenvectors and a vector D containing the variance explained by each principal axis. It is only allowed to use the function eigen.

Solution:

We create a function where we implement the following steps:

```
PCA
```

```
## function (X_train, n_comp)
## {
##     M <- colMeans(X_train)
##     G <- X_train - M
##     G_ <- t(G)
##     n_train <- nrow(X_train)
##     Sigma_s <- (G %*% G_)/(n_train - 1)
##     eig = eigen(Sigma_s)
##     e_vec_s = eig$vectors
##     P = G_ %*% e_vec_s
##     L = diag(eig$values)
##     D = round(eig$values/sum(eig$values), 3)
##     top_comp <- P[, 1:n_comp]
##     list(components = top_comp, mean = M, vectors = P, values = eig$values,
##         var_exp = D)
## }
```

Then a small one to project the train and test data, taking into account the PCA made over the train data before:

```
project_pca
```

```
## function (data, pca_model)
## {
##     centered_data <- data - pca_model$mean
##     projected_data <- as.matrix(centered_data) %*% pca_model$components
##     return(projected_data)
## }
```

B)

Built a classifier (function) that takes as input an image and an object with the parameters of the classifier. Internally, the function uses a k-nn classifier and the PCA representation of the images. If the person in the image belongs to the database, it returns the person's identifiers. Otherwise it returns 0. In order to build this, you will need to consider:

- The percentage of the variance retaining by the PCs
- The number of neighbors of the k-nn
- The similarity metric
- The threshold to determine when the person belongs to the database

Solution:

To begin, we define a special function to define the train/test split.

First we take some labels out in order to define the "impostors" and be allowed to test the threshold to determine when the person belongs to the database.

With the remaining labels we take some images to the test split (that are also in the train) in order to train the KNN as is normally done.

Finally we join the observations obtained in the two previous steps and define them as the test dataset. The remaining observations are going to be the train dataset.

```
create_train_test_split
```

```
## function (data, labels, num_persons_out, split_seed = NULL)
## {
##     labels <- as.character(labels)
##     unique_persons <- unique(labels)
##     if (!is.null(split_seed)) {
##         set.seed(split_seed)
##     }
##     test_persons <- sample(unique_persons, num_persons_out)
##     remaining_persons <- setdiff(unique_persons, test_persons)
##     test_indices_p1 <- which(labels %in% test_persons)
##     remaining_indices <- which(labels %in% remaining_persons)
##     if (!is.null(split_seed)) {
##         set.seed(split_seed)
##     }
##     test_indices_p2 <- sample(remaining_indices, round(length(remaining_indices) *
##         0.2))
##     test_indices <- c(test_indices_p1, test_indices_p2)
##     test_data <- data[test_indices, ]
##     test_labels <- labels[test_indices]
##     train_data <- data[-test_indices, ]
##     train_labels <- labels[-test_indices]
##     return(list(train_data = train_data, train_labels = train_labels,
##         test_data = test_data, test_labels = test_labels, test_indices = test_indices))
## }
```

With that done, we define the KNN for different possible types of distances (Mahalanobis, SSE Modified and W angle).

```
knn_classifier
```

```
## function (train_data, train_labels, test_data, k, percent_threshold,
##     pca_model = NULL, n_comp = NULL, distance_func = mahalanobis_distance)
## {
##     within_distances <- c()
##     between_distances <- c()
##     distances <- matrix(NA, nrow(train_data), nrow(train_data))
##     for (i in 1:nrow(train_data)) {
##         test_point <- train_data[i, ]
##         distances[i, ] <- distance_func(test_point, train_data,
##             pca_model, n_comp)
##     }
##     n <- nrow(train_data)
##     for (i in 1:(n - 1)) {
##         for (j in (i + 1):n) {
##             if (train_labels[i] == train_labels[j]) {
##                 within_distances <- c(within_distances, distances[i,
##                   j])
##             }
##             else {
##                 between_distances <- c(between_distances, distances[i,
##                   j])
##             }
##         }
##     }
##     max_within <- max(within_distances)
##     q10_between <- quantile(between_distances, 0.1)
##     if (max_within < q10_between) {
##         threshold <- max_within + percent_threshold * (q10_between -
##             max_within)
##     }
##     else {
##         threshold <- quantile(within_distances, 1 - percent_threshold)
##     }
##     cat("Threshold value:", threshold, "\n")
##     predictions <- character(nrow(test_data))
##     min_distances <- numeric(nrow(test_data))
##     for (i in 1:nrow(test_data)) {
##         test_point <- test_data[i, ]
##         distances_test <- distance_func(test_point, train_data,
##             pca_model, n_comp)
##         min_dist <- min(distances_test)
##         min_distances[i] <- min_dist
##         if (min_dist > threshold) {
##             predictions[i] <- "0"
##             cat("Sample", i, "distance", min_dist, "> threshold",
##                 threshold, "→ label 0\n")
##         }
##         else {
##             neighbors <- order(distances_test)[1:k]
##             neighbor_labels <- train_labels[neighbors]
##             label_counts <- table(neighbor_labels)
```

```
##             most_common_idx <- which.max(label_counts)
##             predicted_label <- names(label_counts)[most_common_idx]
##             predictions[i] <- as.character(predicted_label)
##             cat("Sample", i, "distance", min_dist, "<= threshold",
##                 threshold, "→ KNN label", predicted_label,
##                 "\n")
##         }
##     }
##     return(list(within = within_distances, between = between_distances,
##         thres = threshold, pred = predictions, min_distances = min_distances))
## }
```

C)

Explain how you have determined the previous parameters

Solution:

We made a tuning function where a grid search is done for selecting the parameters:

- K: Number of neighbors
- percent_thresholds: How far is going to be the threshold from the within distances taking into account the between distances
- n_components: How many components of the PCA we are going to keep taking into account the % of variance explained

```
# Define parameter grid for tuning
n_comp_thresholds <- c(0.80, 0.85, 0.90, 0.95)  # Cumulative variance thresholds for PCA
k_values <- c(1, 3, 5)  # Number of neighbors for KNN
percent_thresholds <- c(0.1, 0.2, 0.3)  # Thresholds for novel person detection
num_splits <- 5  # Number of cross-validation splits
num_persons_out <- 2  # Number of persons to leave out for testing

results <- lppo_tuning(
  data = X,
  labels = labels,
  num_persons_out = num_persons_out,
  n_comp_thresholds = n_comp_thresholds,
  k_values = k_values,
  percent_thresholds = percent_thresholds,
  num_splits = num_splits
)
```

```
results %>% saveRDS("results_pca_tunning.RDS")
```

```
results <- readRDS("results_pca_tunning.RDS")

average_distance <- results$avg_results %>%
  group_by(distance) %>%
  summarise(av_accuracy = mean(accuracy))

average_results <- results$avg_results %>%
  group_by(distance, k, n_comp) %>%
```

```
  summarise(av_accuracy = mean(accuracy)) %>%
  arrange(desc(av_accuracy))
```

```
## `summarise()` has grouped output by 'distance', 'k'. You can override using the
## `.groups` argument.
```

```
average_distance
```

```
## # A tibble: 3 x 2
##   distance    av_accuracy
##   <chr>             <dbl>
## 1 mahalanobis       0.752
## 2 sse_mod           0.631
## 3 w_angle           0.596
```

```
average_results %>% head(10)
```

```
## # A tibble: 10 x 4
## # Groups:   distance, k [5]
##    distance        k n_comp av_accuracy
##    <chr>       <dbl>  <int>       <dbl>
##  1 mahalanobis     5     21       0.975
##  2 mahalanobis     1     21       0.95
##  3 mahalanobis     3     21       0.95
##  4 mahalanobis     1     14       0.883
##  5 mahalanobis     3     14       0.883
##  6 mahalanobis     5     14       0.883
##  7 sse_mod         1     23       0.862
##  8 mahalanobis     1      9       0.85
##  9 mahalanobis     3      9       0.85
## 10 sse_mod         3     23       0.85
```

With the previous results we see that the best combination of parameters is:

```
average_results %>% head(1)
```

```
## # A tibble: 1 x 4
## # Groups:   distance, k [1]
##   distance        k n_comp av_accuracy
##   <chr>       <dbl>  <int>       <dbl>
## 1 mahalanobis     5     21       0.975
```

Application

```
# Train test split
split_data <- create_train_test_split(data=X, labels = labels, num_persons_out = 2,
                        split_seed = 782)
```

```r
train_data <- split_data$train_data
train_labels <- split_data$train_labels

test_data <- split_data$test_data
test_labels <- split_data$test_labels

# PCA model with n_comp found
pca_model <- PCA(train_data, results$n_components)

# Projection of train and test
train_pca <- project_pca(train_data, pca_model)
test_pca <- project_pca(test_data, pca_model)

# Application of the KNN with the parameters tunned
predictions <- knn_classifier(train_data = train_pca, train_labels = train_labels,
                              test_data = test_pca,   k = results$k,
                              percent_threshold = results$percent_threshold,
                              pca_model = pca_model,  n_comp = results$n_components)
```

```
## Threshold value: 575.9565
## Sample 1 distance 584.3218 > threshold 575.9565 → label 0
## Sample 2 distance 585.0363 > threshold 575.9565 → label 0
## Sample 3 distance 584.8462 > threshold 575.9565 → label 0
## Sample 4 distance 581.8401 > threshold 575.9565 → label 0
## Sample 5 distance 600.8346 > threshold 575.9565 → label 0
## Sample 6 distance 589.2034 > threshold 575.9565 → label 0
## Sample 7 distance 550.3868 <= threshold 575.9565 → KNN label 10
## Sample 8 distance 548.0005 <= threshold 575.9565 → KNN label 10
## Sample 9 distance 559.0127 <= threshold 575.9565 → KNN label 10
## Sample 10 distance 553.4407 <= threshold 575.9565 → KNN label 10
## Sample 11 distance 540.9048 <= threshold 575.9565 → KNN label 10
## Sample 12 distance 583.5596 > threshold 575.9565 → label 0
## Sample 13 distance 401.8625 <= threshold 575.9565 → KNN label 16
## Sample 14 distance 376.0496 <= threshold 575.9565 → KNN label 5
## Sample 15 distance 375.9324 <= threshold 575.9565 → KNN label 3
## Sample 16 distance 398.9304 <= threshold 575.9565 → KNN label 15
## Sample 17 distance 393.8407 <= threshold 575.9565 → KNN label 3
## Sample 18 distance 383.5101 <= threshold 575.9565 → KNN label 7
## Sample 19 distance 409.7618 <= threshold 575.9565 → KNN label 13
## Sample 20 distance 412.8414 <= threshold 575.9565 → KNN label 15
## Sample 21 distance 397.3591 <= threshold 575.9565 → KNN label 1
## Sample 22 distance 368.1117 <= threshold 575.9565 → KNN label 7
## Sample 23 distance 438.161 <= threshold 575.9565 → KNN label 21
## Sample 24 distance 322.9687 <= threshold 575.9565 → KNN label 9
## Sample 25 distance 402.1309 <= threshold 575.9565 → KNN label 20
## Sample 26 distance 386.8464 <= threshold 575.9565 → KNN label 21
## Sample 27 distance 441.8901 <= threshold 575.9565 → KNN label 4
## Sample 28 distance 419.2743 <= threshold 575.9565 → KNN label 17
## Sample 29 distance 435.0473 <= threshold 575.9565 → KNN label 17
## Sample 30 distance 419.102 <= threshold 575.9565 → KNN label 4
## Sample 31 distance 346.831 <= threshold 575.9565 → KNN label 8
## Sample 32 distance 359.1682 <= threshold 575.9565 → KNN label 10
## Sample 33 distance 397.0317 <= threshold 575.9565 → KNN label 10
```

```
## Sample 34 distance 364.1759 <= threshold 575.9565 → KNN label 22
## Sample 35 distance 402.3062 <= threshold 575.9565 → KNN label 12
## Sample 36 distance 413.6447 <= threshold 575.9565 → KNN label 25
## Sample 37 distance 402.4196 <= threshold 575.9565 → KNN label 12
## Sample 38 distance 404.2643 <= threshold 575.9565 → KNN label 24
## Sample 39 distance 447.5274 <= threshold 575.9565 → KNN label 19
## Sample 40 distance 410.2997 <= threshold 575.9565 → KNN label 19
```

```
#-------------------------------------------------------------------------------
# Accuracy of the example
table(predictions$pred, test_labels)
```

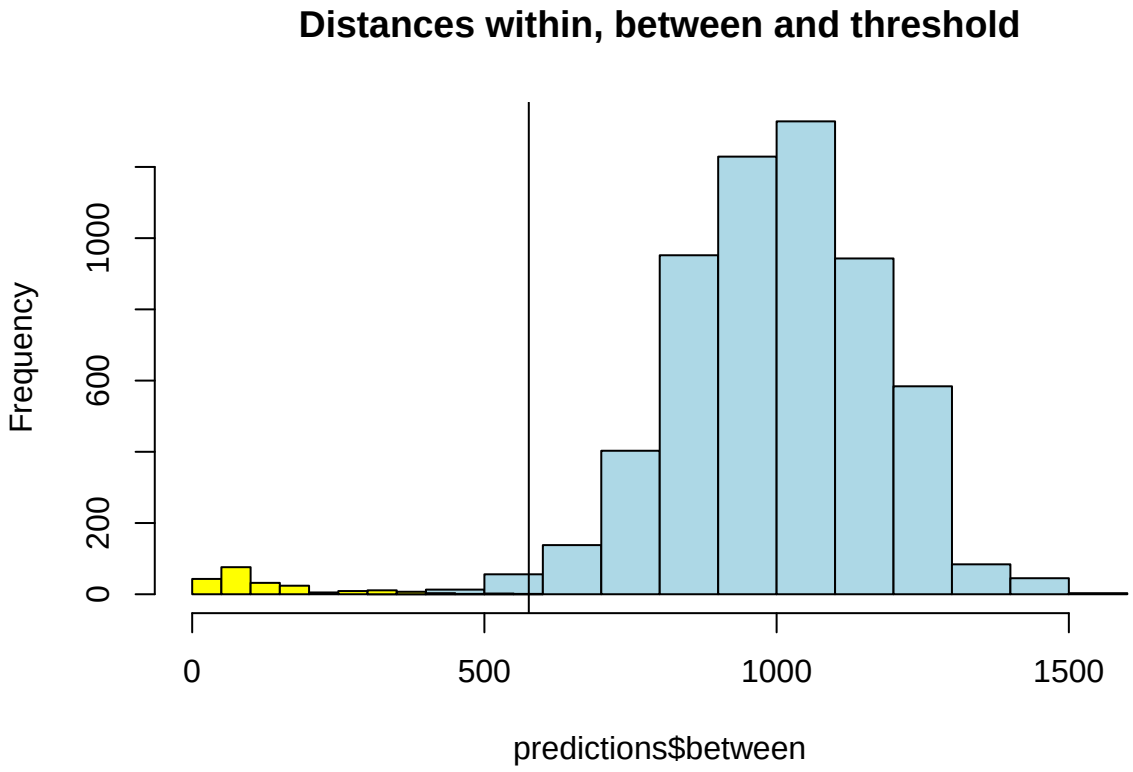
```
##     test_labels
##      1 10 12 13 14 15 16 17 18 19 2 20 21 22 25 3 4 5 7 8 9
##   0  0  0  0  0  0  0  0  0  6  0 1  0  0  0  0 0 0 0 0 0 0
##   1  1  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   10 0  2  0  0  0  0  0  0  0  0 5  0  0  0  0 0 0 0 0 0 0
##   12 0  0  2  0  0  0  0  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   13 0  0  0  1  0  0  0  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   15 0  0  0  0  0  2  0  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   16 0  0  0  0  0  0  1  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   17 0  0  0  0  0  0  0  2  0  0 0  0  0  0  0 0 0 0 0 0 0
##   19 0  0  0  0  0  0  0  0  0  2 0  0  0  0  0 0 0 0 0 0 0
##   20 0  0  0  0  0  0  0  0  0  0 0  1  0  0  0 0 0 0 0 0 0
##   21 0  0  0  0  0  0  0  0  0  0 0  0  2  0  0 0 0 0 0 0 0
##   22 0  0  0  0  0  0  0  0  0  0 0  0  0  1  0 0 0 0 0 0 0
##   24 0  0  0  0  1  0  0  0  0  0 0  0  0  0  0 0 0 0 0 0 0
##   25 0  0  0  0  0  0  0  0  0  0 0  0  0  0  1 0 0 0 0 0 0
##   3  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 2 0 0 0 0 0
##   4  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 2 0 0 0 0
##   5  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 0 1 0 0 0
##   7  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 0 0 2 0 0
##   8  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 0 0 0 1 0
##   9  0  0  0  0  0  0  0  0  0  0 0  0  0  0  0 0 0 0 0 0 1
```

```
impostors <- names(table(test_labels)[table(test_labels)==6]) # all the images in the test set

correct_predictions <- sum(predictions$pred == test_labels)
correct_impostors <- sum(predictions$pred == 0 & test_labels %in% impostors)

accuracy <- (correct_predictions + correct_impostors) / length(predictions$pred)
cat("accuracy:", accuracy)
```

```
## accuracy: 0.85
```

Also, the distribution of the distances looks as follows:

```
hist(predictions$between, xlim = c(0,1600), col = "lightblue",
     main="Distances within, between and threshold")
hist(predictions$within, xlim = c(0,1600), col = "yellow", add=T )
abline(v=predictions$thres)
```



PART B

Implement a facial recognizer based on Fisher discriminant analysis.

A)

Build a function that implements the Fisher Discriminant Analysis. This function takes as input a set of observations and returns the mean of these observations, the matrix P containing the eigen vector of the appropriate matrix and a vector D containing the variance explained by each fisher discriminant. It is only allowed to use the function eigen.

Solution:

We build a function that takes PCA data and labels for class identification. It first calculates the mean for each individual class, then gets the Sb(between) and Sw(within) matrices with the equations from the notes provided. Lastly, it produces the eigenvalues and eigenvectors of our conversion matrix w. After this, we also have a function to project a set of data onto a w matrix, with project_fisher.

```
Fisher
```

```
## function (PCA, labels)
## {
##     overallmeans = colMeans(PCA)
##     labels = as.numeric(labels)
##     n_features = ncol(PCA)
##     n_classes = length(labels)
##     cmeans = matrix(NA, nrow = max(labels), ncol = n_features)
```

```
##     for (label in unique(labels)) {
##         cmeans[label, ] = colMeans(PCA[(labels == label), ])
##     }
##     Sb = matrix(0, nrow = n_features, ncol = n_features)
##     for (label in unique(labels)) {
##         base_matrix = cmeans[label, ] - overallmeans
##         unit_m = sum(labels == label) * base_matrix %*% t(base_matrix)
##         Sb = Sb + unit_m
##     }
##     Sw = matrix(0, nrow = n_features, ncol = n_features)
##     for (label in unique(labels)) {
##         class_matrix = PCA[(labels == label), ]
##         B_matrix = sweep(class_matrix, 2, cmeans[label, ], "-")
##         D_matrix = matrix(0, nrow = n_features, ncol = n_features)
##         for (j in 1:sum(labels == label)) {
##             C_matrix = B_matrix[j, ] %*% t(B_matrix[j, ])
##             D_matrix = D_matrix + C_matrix
##         }
##         Sw = Sw + D_matrix
##     }
##     w = eigen(solve(Sw) %*% Sb)
##     variance_matrix = round(w$values/sum(w$values), 3)
##     list(mean = overallmeans, vectors = Re(w$vectors), var_exp = variance_matrix,
##         values = Re(w$values))
## }
```

```
project_fisher
```

```
## function (PCA, w)
## {
##     proj = as.matrix(PCA) %*% w$vectors
##     proj
## }
```

B)

Built a classifier (function) that takes as input an image and an object with the parameters of the classifier. Internally, the function uses a k-nn classifier and the Fisher discriminant analysis representation of the images. If the person in the image belongs to the database, it returns the person's identifiers. Otherwise it returns 0. In order to build this, you will need to consider:

- The percentage of the variance retaining by the Fisher discriminant dimensions
- The number of neighbors of the k-nn
- The similarity metric
- The threshold to determine when the person belongs to the database

Solution:

We take our previous train-split and knn classifier functions and reuse them here.

```
create_train_test_split
```

```
## function (data, labels, num_persons_out, split_seed = NULL)
## {
##     labels <- as.character(labels)
##     unique_persons <- unique(labels)
##     if (!is.null(split_seed)) {
##         set.seed(split_seed)
##     }
##     test_persons <- sample(unique_persons, num_persons_out)
##     remaining_persons <- setdiff(unique_persons, test_persons)
##     test_indices_p1 <- which(labels %in% test_persons)
##     remaining_indices <- which(labels %in% remaining_persons)
##     if (!is.null(split_seed)) {
##         set.seed(split_seed)
##     }
##     test_indices_p2 <- sample(remaining_indices, round(length(remaining_indices) *
##         0.2))
##     test_indices <- c(test_indices_p1, test_indices_p2)
##     test_data <- data[test_indices, ]
##     test_labels <- labels[test_indices]
##     train_data <- data[-test_indices, ]
##     train_labels <- labels[-test_indices]
##     return(list(train_data = train_data, train_labels = train_labels,
##         test_data = test_data, test_labels = test_labels, test_indices = test_indices))
## }
## <bytecode: 0x000001f85df3ca88>
```

```
knn_classifier
```

```
## function (train_data, train_labels, test_data, k, percent_threshold,
##     pca_model = NULL, n_comp = NULL, distance_func = mahalanobis_distance)
## {
##     within_distances <- c()
##     between_distances <- c()
##     distances <- matrix(NA, nrow(train_data), nrow(train_data))
##     for (i in 1:nrow(train_data)) {
##         test_point <- train_data[i, ]
##         distances[i, ] <- distance_func(test_point, train_data,
##             pca_model, n_comp)
##     }
##     n <- nrow(train_data)
##     for (i in 1:(n - 1)) {
##         for (j in (i + 1):n) {
##             if (train_labels[i] == train_labels[j]) {
##                 within_distances <- c(within_distances, distances[i,
##                   j])
##             }
##             else {
##                 between_distances <- c(between_distances, distances[i,
##                   j])
##             }
##         }
##     }
##     max_within <- max(within_distances)
##     q10_between <- quantile(between_distances, 0.1)
```

```
##     if (max_within < q10_between) {
##         threshold <- max_within + percent_threshold * (q10_between -
##             max_within)
##     }
##     else {
##         threshold <- quantile(within_distances, 1 - percent_threshold)
##     }
##     cat("Threshold value:", threshold, "\n")
##     predictions <- character(nrow(test_data))
##     min_distances <- numeric(nrow(test_data))
##     for (i in 1:nrow(test_data)) {
##         test_point <- test_data[i, ]
##         distances_test <- distance_func(test_point, train_data,
##             pca_model, n_comp)
##         min_dist <- min(distances_test)
##         min_distances[i] <- min_dist
##         if (min_dist > threshold) {
##             predictions[i] <- "0"
##             cat("Sample", i, "distance", min_dist, "> threshold",
##                 threshold, "→ label 0\n")
##         }
##         else {
##             neighbors <- order(distances_test)[1:k]
##             neighbor_labels <- train_labels[neighbors]
##             label_counts <- table(neighbor_labels)
##             most_common_idx <- which.max(label_counts)
##             predicted_label <- names(label_counts)[most_common_idx]
##             predictions[i] <- as.character(predicted_label)
##             cat("Sample", i, "distance", min_dist, "<= threshold",
##                 threshold, "→ KNN label", predicted_label,
##                 "\n")
##         }
##     }
##     return(list(within = within_distances, between = between_distances,
##         thres = threshold, pred = predictions, min_distances = min_distances))
## }
## <bytecode: 0x000001f85fda2238>
```

C)

Explain how you have determined the previous parameters

Solution:

We adapted the previous lppo_tuning function to a version that can take as input PCA data we produce and apply the Fisher functions to it, then use them to calculate accuracies and cross-validate.

```
# Define parameter grid for tuning
n_comp_thresholds <- c(0.80, 0.85, 0.90, 0.95)  # Cumulative variance thresholds for PCA
k_values <- c(1, 3, 5)  # Number of neighbors for KNN
percent_thresholds <- c(0.1, 0.2, 0.3)  # Thresholds for novel person detection
num_splits <- 5  # Number of cross-validation splits
num_persons_out <- 2  # Number of persons to leave out for testing
```

```r
PCA_model = PCA(X,21)
proj = project_pca(X,PCA_model)[,1:21]

# Run the tuning function
source("function_loading.R")

tuning_results_fisher <- lppo_tuning_FISHER(
  data = proj,
  labels = labels,
  num_persons_out = num_persons_out,
  n_comp_thresholds = n_comp_thresholds,
  k_values = k_values,
  percent_thresholds = percent_thresholds,
  num_splits = num_splits
)
```

```r
tuning_results_fisher %>% saveRDS("results_fisher_tuning.RDS")
```

```r
tuning_results_fisher <- readRDS("results_fisher_tuning.RDS")
```

```r
average_distance <- tuning_results_fisher$avg_results %>%
  group_by(distance) %>%
  summarise(av_accuracy = mean(accuracy))

average_results <- tuning_results_fisher$avg_results %>%
  group_by(distance, k, n_comp) %>%
  summarise(av_accuracy = mean(accuracy)) %>%
  arrange(desc(av_accuracy))
```

```
## `summarise()` has grouped output by 'distance', 'k'. You can override using the
## `.groups` argument.
```

```r
average_distance
```

```
## # A tibble: 3 x 2
##   distance    av_accuracy
##   <chr>             <dbl>
## 1 mahalanobis       0.959
## 2 sse_mod           0.906
## 3 w_angle           0.510
```

```r
average_results %>% head(10)
```

```
## # A tibble: 10 x 4
## # Groups:   distance, k [4]
##    distance        k n_comp av_accuracy
##    <chr>       <dbl>  <int>       <dbl>
##  1 mahalanobis     1     11       1
##  2 mahalanobis     3     11       1
##  3 sse_mod         1     11       1
```

```
##  4 sse_mod         3     11       1
##  5 mahalanobis     1     10       0.985
##  6 mahalanobis     3     10       0.985
##  7 mahalanobis     1      6       0.978
##  8 mahalanobis     1      8       0.977
##  9 mahalanobis     3      8       0.977
## 10 mahalanobis     3      6       0.973
```

We see that also the mahalanobis function is giving the best results, but for large values of fisher components is overfitting. For that reason we decide to take the parameters:

- Distance: Mahalanobis
- K = 1
- n_comp = 6

That also gives a good classification.