Assignment

2025-02-25

```
library(archive)
library(dplyr)
library(plotly)
library(knitr)

COMPUTE_GRID_SEARCH <- FALSE
```

0. Input data

Let's begin by discussing the first step of our code, which handles extracting the image data stored in a compressed archive. We start by specifying the file name `"Training.rar"`, which contains all of our images. To extract the contents, we use the `archive_extract()` function from the archive package. We wrap this call in a `try()` block with the parameter `silent = TRUE`. This is a safeguard: if an error occurs during extraction(for example, if the archive has already been extracted), and we can smoothly continue with the rest of the processing.

```
# Specify the file path
rar_file <- "Training.rar"

# Extract the contents
#unpacked_files <- archive::archive_extract(rar_file)
try(archive_extract("Training.rar"), silent = TRUE)
```

Following the extraction, we define two helper functions to manage and organize our image files. The first function, `extract_id`, takes a filename and uses a regular expression to extract the first sequence of digits from it. We then convert that sequence into a numeric value, which serves as a unique identifier for each image (this identifier typically corresponds to a subject's ID). The second function, `get_sorted_filenames`, makes use of `list.files()` to retrieve all JPEG files in the "Training" folder. We then apply our `extract_id` function to each filename to obtain their numeric IDs, sort the filenames based on these IDs, and return the sorted list.

```
extract_id <- function(filename) {
    matches <- regmatches(basename(filename), regexpr("[0-9]+", basename(filename)))
    return(as.numeric(matches))  # Convert to numeric
}

get_sorted_filenames <- function(){
  # Get all .jpg files
  image_files <- list.files("Training", pattern = "\\.jpg$", full.names = TRUE)

  sorted_indices <- order(sapply(image_files, extract_id))
  image_files <- image_files[sorted_indices]
  return(image_files)
}
```

Once the filenames are sorted, we proceed to load and process the images. We store the sorted list of image file paths in the variable `image_files` and extract the numeric IDs again for potential labeling or further processing. Next, we iterate over each file, loading the image using `OpenImageR::readImage()`. Since our images are in color, we need to handle each of the three channels—red, green, and blue—separately.

For each channel, the two-dimensional array that represents the pixel intensities is flattened into a one-dimensional vector using `as.vector()`. We then concatenate these three vectors into a single vector, effectively transforming each image into a long vector that contains all the pixel values sequentially. Mathematically, if an image is represented by a 3D array

$$I \in \mathbb{R}^{H \times W \times 3},$$

then the red, green, and blue channels are flattened into vectors

$$r, g, b \in \mathbb{R}^{HW},$$

respectively, and concatenated as:

$$\mathbf{x} = \begin{bmatrix} r \\ g \\ b \end{bmatrix} \in \mathbb{R}^{3HW}.$$

Each of these vectors is stored in a list, using the file name as the key. This organized storage allows us to later combine all the image vectors into a single matrix for further analysis.

```
# Get all .jpg files in the Training folder
image_files <- get_sorted_filenames()
#Remove the files
#image_files <- image_files[!image_files %in% c("Training/1AT.jpg", "Training/3BT.jpg")]

image_ids <- sapply(image_files, extract_id)

image_rows  <- list()
for (file in image_files) {
  img <- OpenImageR::readImage(file)  # Load image

  # Flatten RGB channels and concatenate into a single row
  red <- as.vector(img[,,1])  # Flatten red channel
  green <- as.vector(img[,,2]) # Flatten green channel
  blue <- as.vector(img[,,3])  # Flatten blue channel

  # Combine R, G, B into one row and store it
  image_rows[[file]] <- c(red, green, blue)
}

# Convert the list of rows into a matrix
M <- do.call(rbind, image_rows)

# Check dimensions (rows = number of images, columns = total pixels * 3)
dim(M)
```

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

1. PCA

The function `compute_pca` begins by calculating the mean vector of the input data matrix, where each column corresponds to a feature (in our case, pixel values). We use the `colMeans` function to compute this mean and store it in the variable `mean_m`. Immediately afterward, we center the data by subtracting this mean from every observation. We achieve this centering using the `sweep` function, which subtracts the mean from each column of the data matrix, ensuring that the new data matrix has a zero mean along every dimension.

Once the data is centered, we compute a covariance-like matrix. However, rather than calculating the full $d \times d$ covariance matrix (with d being the number of pixels), we compute a smaller $n \times n$ matrix, where n is the number of observations. We do this by multiplying the centered data matrix by its transpose and scaling the result by $\frac{1}{n-1}$. This yields

$$C = \frac{1}{n-1} X_c X_c^\top,$$

where X_c represents our centered data matrix. This smaller covariance matrix is computationally more efficient to work with when the dimensionality is very high.

Next, we perform an eigen-decomposition on this smaller covariance matrix using the `eigen` function, which returns a set of eigenvalues and eigenvectors. Because these eigenvectors are computed in the space of the $n \times n$ matrix, they do not directly represent the directions in the original high-dimensional pixel space. To map them back, the function multiplies the transpose of the centered data by these eigenvectors. This operation effectively transfers the principal component directions back into the original space. After that, we normalize each column (each representing an eigenvector in pixel space) by dividing it by its Euclidean norm, ensuring that every principal component has unit length.

Finally, we compute the proportion of variance explained by each principal component. This is done by dividing each eigenvalue by the sum of all eigenvalues, which gives us a normalized vector of explained variances. The function then returns a list containing the mean vector (`mean_obs`), the matrix of normalized eigenvectors (`P`), the vector of explained variance proportions (`D`), and the raw eigenvalues (`E`).

```
compute_pca <- function(data) {
  # Compute mean
  mean_m <- colMeans(data)

  # Subtract the mean from each row
  centered_matrix <- sweep(data, 2, mean_m, FUN = "-")

  n <- nrow(centered_matrix)
  var_cov_small <- (1 / (n - 1)) * (centered_matrix %*% t(centered_matrix))
  eigen_dec <- eigen(var_cov_small)

  P <- t(centered_matrix) %*% eigen_dec$vectors
  P <- apply(P, 2, function(p) p / sqrt(sum(p^2)))


  D <- eigen_dec$values / sum(eigen_dec$values)

  return(list(mean_obs = mean_m, P = P, D = D, E = eigen_dec$values))
}

pca_results <- compute_pca(M)

mean_obs <- pca_results$mean_obs
P <- pca_results$P #eigenvectors in pixel space normalized #image space
D <- pca_results$D #variance explained by eigenvalue
E <- pca_results$E #eigenvalues
#V <- pca_results$V #eigenvectors in pixel space normalized

pca_results <- compute_pca(M)

mean_obs <- pca_results$mean_obs
P <- pca_results$P #eigenvectors in pixel space normalized #image space
D <- pca_results$D #variance explained by eigenvalue
E <- pca_results$E #eigenvalues
#V <- pca_results$V #eigenvectors in pixel space normalized
```

1.1. PCA Visualization

The projection of the images in the pca space will be the following:

```
Y <- sweep(M, 2, mean_obs, FUN = "-") %*% P
```

We can visualize the `pc` component of the `image_index` image as follows. For example we can see the first image in the dataset projection to the first principal component.

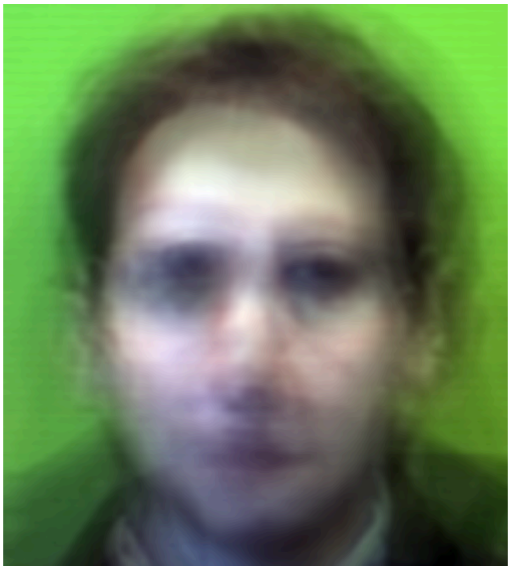
The function `visualize_pc` lets us see how a specific principal component contributes to one of our images. Essentially, we extract the desired principal component (in our case, the first one) from the matrix P, which represents the direction capturing a significant portion of the variance in our data. We then take the corresponding coordinate for our selected image from the projection matrix Y (this tells us how much of that principal component is present in the image).

To visualize this, we reconstruct the image by scaling the principal component by that coordinate and then adding back the overall mean image vector. Mathematically, we compute:

$$x_pc1 = y_1 \times p_1 + \text{mean\_obs},$$

where p_1 is the chosen principal component and y_1 is the image's coordinate along this component. Once we have this reconstructed vector, we reshape it into the original image dimensions (height, width, and 3 color channels) to form an image matrix that we can display using `OpenImageR::imageShow()`.

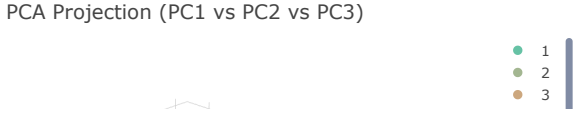
```
visualize_pc <- function(image_index, pc, mean_obs, P, Y, img_height = 200, img_width = 180) {
  # Extract the first principal component
  p1 <- P[,pc]  # First principal component (size p)

  # Get the projection of the specific image onto PC1
  y1 <- Y[image_index, 1]  # First coordinate in PCA space

  # Reconstruct only PC1 contribution
  x_pc1 <- y1 * p1 + mean_obs

  # Reshape into image dimensions
  image_matrix <- array(x_pc1, dim = c(img_height, img_width, 3))

  # Plot the PC1 contribution as an image
  return(image_matrix)

}

# Example: Visualizing the PC1 contribution for image x
image_matrix <- visualize_pc(1,1, mean_obs, P, Y)
OpenImageR::imageShow(image_matrix)
```

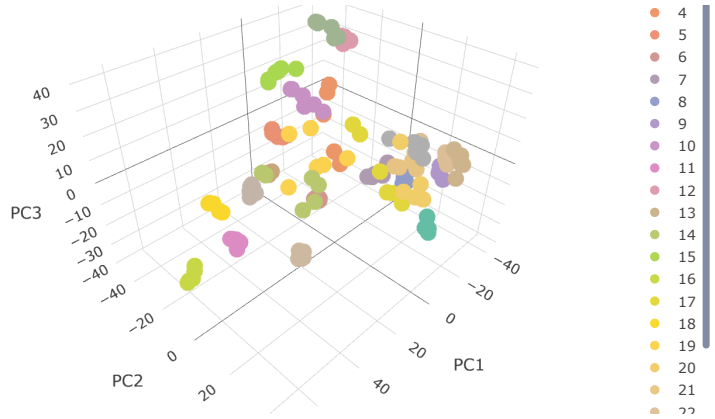


In this part, we take a closer look at how our PCA has organized the image data by visualizing the projections onto the first three principal components. By plotting all images in this 3D space, we can see that images belonging to the same person tend to group together, which is a strong indication that our PCA is capturing the underlying structure of the data effectively.

To achieve this, we define the function `plot_pca_3d_interactive`. In this function, we create a data frame that includes the first three principal components (PC1, PC2, and PC3) and assign a color based on each image's ID. This allows us to visually differentiate the images by subject. We then use the `plot_ly` function from the Plotly library to generate an interactive 3D scatter plot. Each point in the plot represents an image, and hovering over a point displays the corresponding ID. Finally, we call this function with our projection matrix `Y` and the `image_ids` to display the visualization.

```
plot_pca_3d_interactive <- function(Y, image_ids) {
  df <- data.frame(PC1 = Y[,1], PC2 = Y[,2], PC3 = Y[,3], ID = factor(image_ids))

  plot_ly(df, x = ~PC1, y = ~PC2, z = ~PC3, color = ~ID, text = ~paste("ID:", ID),
          type = "scatter3d", mode = "markers") %>%
    layout(title = "PCA Projection (PC1 vs PC2 vs PC3)")
}

plot_pca_3d_interactive(Y, image_ids)
```





2. FDA

The function `fda` is designed to perform Fisher Discriminant Analysis (FDA) on a dataset, where the input matrix Y has dimensions $n \times p$ and its last column represents the class labels. We begin by extracting the class labels from the last column of Y and converting them into a factor so that we can properly identify the different groups. The remaining columns of Y form the feature matrix X, which contains the observations that we will use for the FDA.

Once we have isolated X, we calculate the overall mean vector m of the features using the column means of X. This mean vector represents the central tendency of the entire dataset in the feature space. Next, we determine the unique classes present in the class vector and initialize two key scatter matrices: the within-class scatter matrix S_W and the between-class scatter matrix S_B. Both matrices are initialized as zero matrices of size $p \times p$, where p is the number of features.

For each unique class, we extract the submatrix X_{class} containing only the observations that belong to that class and compute the class mean (denoted as μ_{class}) for those observations. We then update the between-class scatter matrix S_B by adding the weighted outer product of the difference between the class mean and the overall mean. Mathematically, for each class with n_i observations, the contribution is

$$n_i(\mu_{\text{class}} - m)(\mu_{\text{class}} - m)^T.$$

At the same time, we compute the within-class scatter matrix S_W by iterating over each observation in the class. For each observation x_j in X_{class}, we calculate the difference $x_j - \mu_{\text{class}}$ and add the outer product of this difference with itself to S_W, that is,

$$(x_j - \mu_{\text{class}})(x_j - \mu_{\text{class}})^T.$$

After processing all the classes, we perform an eigen-decomposition on the matrix product $S_W^{-1}S_B$. This is done by inverting S_W using the `solve` function and then multiplying it by S_B. The eigenvalues and eigenvectors resulting from this operation indicate the directions in the feature space that maximize the ratio of between-class scatter to within-class scatter. We then sort the eigenvalues in decreasing order and reorder the eigenvectors accordingly so that the most discriminative directions appear first. Any imaginary part is omitted because it induces errors and typically consists of very small values.

Finally, the function returns a list containing the overall mean vector m, the sorted eigenvectors (denoted as P), and the eigenvalues (denoted as D). These eigenvectors represent the FDA directions that optimally separate the classes, while the eigenvalues reflect the discriminative power along those directions.

```r
fda <- function(Y) {
  # Y is an n x p matrix obtained from a PCA where the last column has to contain the class.

  # We extract the class column and convert it to a factor
  class_vector <- Y[, ncol(Y)]
  class_vector <- as.factor(class_vector)

  # We extract the feature matrix (all columns except the last one)
  X <- Y[, -ncol(Y), drop = FALSE]

  # We compute the overall mean of the features
  m <- colMeans(X)

  # Determine the number of classes and the unique classes
  people <- unique(class_vector)

  # We obtain the number of features (dimensions) to select the size of S_W and S_B
  p <- ncol(X)

  # We initialize the within-class scatter matrix (S_W) and the between-class scatter matrix (S_B)
  S_W <- matrix(0, nrow = p, ncol = p)
  S_B <- matrix(0, nrow = p, ncol = p)

  # For each class, we compute the class mean and we accumulate contributions to S_B and S_W
  for (cl in people) {

    # Submatrix of observations for the current class 'cl'
    X_class <- X[class_vector == cl, , drop = FALSE]
    n_i <- nrow(X_class)

    # We compute the mean of the current class
    mean_class <- colMeans(X_class)

    # We update S_B for each class
    # We perform the outer product of (mean_class - m), weighted by n_i
    S_B <- S_B + n_i * ((mean_class - m) %*% t((mean_class - m)))

    # To Update S_W we need to access every observation j of the class one by one
    # We create a for loop which goes through each observation to perform the
    # sum of (x_j - mean_class)(x_j - mean_class)^T
    for (j in 1:n_i) {
      diff_j <- as.matrix(X_class[j, ] - mean_class)
      S_W <- S_W + diff_j %*% t(diff_j)
    }
  }

  # We compute the eigenvalues and eigenvectors for the matrix solve(S_W) %*% S_B
  eig_values <- eigen(solve(S_W) %*% S_B)

  # We obtain the sorted indices of the eigenvalues in decreasing order
  sort_eig <- order(eig_values$values, decreasing = TRUE)

  # We sort the eigenvectors and eigenvalues according these indices
  eigenvectors <- Re(eig_values$vectors[, sort_eig])
  eigenvalues <- Re(eig_values$values[sort_eig])  # Ensure real values
  D <- eig_values$values / sum(eig_values$values)

  return(list(mean_fda = m, P_fda = eigenvectors, D_fda = D, E_fda = eigenvalues))
}
```

3. KNN

The face recognition will be performed using a KNN algorithm. Given a new image of a person, the system should recognize whose person it is in case we have it in the dataset, or classify it as an unknown person if it doesn't. The algorithm follows these steps:

- 1. Take the vector of the image to classify and compute the distance to all the other images of the dataset.
- 2. Sort the distances by ascending order.
- 3. Take the k closest samples and the person id corresponding to it.
- 4. The predicted person id is the majority id among the k closest neighbors.
- 5. Take the distance of the closest neighbor, and if is larger than a threshold, the person on the image is considered as an unknown person.

3.1. Computing Distances

As our KNN classifier relies on computing distances between vectors, we decided to try three different metrics that showed promising results in the paper "Distance measures for PCA-based face recognition." We implemented these metrics(" `mahalanobis` ", " `weighted_angle` ", and " `modified_sse` ") to see which one works best for our application.

The function `compute_distance` is designed to calculate the distance between two vectors, x and y, using one of the three metrics. We choose the metric via the parameter `metric` , and the function also takes a vector of eigenvalues E that is used to compute the weights in some of these distance measures.

For the `"mahalanobis"` metric, we first set a constant
$alpha = 0.25$
. The function then computes a weighting vector z as follows:

$$z = \sqrt{\max\left(0, \frac{E}{E + \alpha^2}\right)},$$

After calculating z, the function returns the negative sum of the element-wise product of z, x, and y. The negative sum is used here so that, for classification, a lower value corresponds to a closer match.

When we choose the `"weighted_angle"` metric, the function computes a different weighting vector by taking:

$$z = \sqrt{\max\left(0, \frac{1}{E}\right)},$$

It then returns the negative weighted inner product of x and y, normalized by the product of their Euclidean norms. In mathematical terms, this distance is calculated as:

$$\text{weighted angle} = -\frac{\sum_i z_i \, x_i \, y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}},$$

which is similar to a cosine similarity measure with weights; the negative sign again ensures that a higher similarity results in a lower distance.

Lastly, for the `"modified_sse"` metric, the function calculates a normalized squared error. It computes the sum of the squared differences between the corresponding elements of x and y, and then divides that sum by the product of the sums of squares of x and y:

$$\text{modified sse} = \frac{\sum_i (x_i - y_i)^2}{\left(\sum_i x_i^2\right)\left(\sum_i y_i^2\right)}.$$

```
# Computes the distance between 2 vectors. D is the eigenvalue list, required for some metrics.
compute_distance <- function(x, y, E, metric) {

    if (metric == "mahalanobis") {
      alpha <- 0.25
      z <- sqrt(pmax(0, E / (E + alpha^2)))
      return(-sum(z * x * y))

    } else if (metric == "weighted_angle") {
      z <- sqrt(pmax(0, 1 / E))
      return(-sum(z * x * y) / sqrt(sum(x^2) * sum(y^2)))

    } else if (metric == "modified_sse") {
      return(sum((x - y)^2) / (sum(x^2) * sum(y^2)))

    } else {
      stop("Invalid metric. Choose from 'mahalanobis', 'weighted_angle', or 'modified_sse'.")
    }
}
```

3.2. KNN Clasiffier

The function `knn_classifier` implements a k-nearest neighbors (KNN) classifier. It takes several parameters: the number of neighbors k, a similarity threshold, the new image's vector y_{new}, a matrix Y containing the representations of the training images (one per row), the corresponding image IDs, a string `metric` indicating which distance metric to use, and a vector E of eigenvalues required by some metrics.

The function first computes the distances between the new image y_{new} and every row of Y. This is achieved by applying the `compute_distance` function to each row, which calculates the similarity according to the selected metric. The resulting distances are stored in a vector, and a data frame is created with two columns: one for the image IDs and another for the computed distances.

After obtaining the distance values, the data frame is sorted in ascending order by distance. The assumption here is that the closer two images are in the space, the greater similarity. The function then selects the k nearest neighbors (i.e., the k rows with the smallest distances). Instead of computing an average distance, which might be misleading if the metric produces both positive and negative values, the function considers the minimum distance among these k neighbors as the decisive measure. This minimum distance is compared against a pre-set threshold: if the smallest distance is greater than the threshold, the function returns 0, signifying that the new image is too dissimilar from any training image to be confidently recognized. This relies on the assumption that if the closest image is already too far to consider the new image as part of the dataset, the rest of the images too.

If the minimum distance is within the acceptable threshold, the function then examines the class IDs of the k nearest neighbors. It counts the occurrences of each ID and returns the majority class (i.e., the ID that appears most frequently). This classification decision is based on the intuition that if the new image is similar to several images from a particular class, it likely belongs to that same class.

In summary, the decision rule can be viewed as: if

$$\min\{d(y_{\text{new}}, y_i) : y_i \in \text{k nearest neighbors}\} > \text{threshold},$$

then return 0, otherwise classify y_{new} according to the majority label among its k nearest neighbors.

```
knn_classifier <- function(k, threshold, y_new, Y, image_ids, metric, E){
  distances <- apply(Y, 1, function(y) compute_distance(y, y_new, E, metric) )
  dist_df <- data.frame(ID = image_ids, Distance = distances)

  # Sort the distance in ascending order
  dist_df <- dist_df[order(dist_df$Distance), ]

  # Take the first k distances
  k_nearest <- head(dist_df, k)

  # Average distance. If distance is larger than threshold return 0. Else continue
  #distance <- mean(k_nearest$Distance) # Doesn't make sense in similarity metrics that can have negative values or ranges t
hat go   from negatives to positive. Averaging positive and negative values might cancel out the real distance.

  # Min distance: If the closest/most similar, is too far, the rest will be farther.
  distance <- min(k_nearest$Distance)

  if (distance > threshold) {
    return(0)
  }

  # value count of the ids. Return the majority class.
  id_counts <- table(k_nearest$ID)
  majority_id <- as.numeric(names(which.max(id_counts)))
  return(majority_id)
}
```

4. Parameter Selection

We will perform a two steps grid search trying to maximize the balanced accuracy. We could do it in one step, and the results will be probably better, but this approach will highly reduce the computation time.

The procedure is the following:

- 1. Keep all the parameters fixed but the threshold and iterate a range of thresholds. Each metric will have a different range of thresholds. This will be done in `n` random train/test splits. In the test split there are always images of people not present in the train split.
- 2. Once found the threshold for each metric that maximizes the balanced accuracy among all the splits, perform a grid search among the other parameters keeping the best threshold found fixed. This will be done in `n` random train/test splits. The best parameters will be those that in average are best.

The justification of this methodology relies on the randomness of the splits. A potential issue with optimizing a threshold on a single split is that it may overfit to the specific individuals present in that split, leading to poor generalization when applied to new data. By evaluating multiple train/test splits where the test set always contains individuals not seen in training, we mitigate this risk by averaging performance across diverse scenarios. This approach helps smooth out variability and ensures that the chosen threshold is not overly sensitive to a particular split but instead represents a robust choice across different partitions of the dataset. We apply the same methodology to the other parameters, reinforcing the generalizability of the final model selection.

Now we will describe the functions used for that.

The function `create_random_splits_by_image` is designed to generate multiple random training and test splits based on image identifiers, while ensuring that a specified number of unique subjects (or classes) are completely excluded from the training set in each split. The input parameter `image_ids` represents the class labels associated with each image, and parameters such as `percent_test`, `num_splits`, and `num_people_to_exclude` allows us to control the proportion of test data, the number of splits, and the number of subjects to exclude from training and include their images in testing, respectively.

At the beginning of the function, a fixed random seed is set using `set.seed(1)` to guarantee reproducibility of the random splits. The function then extracts the unique person id labels from `image_ids` and proceeds to create the splits using `lapply`. Before splitting the data, `num_people_to_exclude` will be randomly selected (their ids), and all the images belonging to this group will be excluded from the splitting process. Then, the splits are created from the set of remaining images, and the ones excluded initially are included in the test set. In mathematical terms, if T represents the set of test indices and E represents the excluded indices, then the final test set is $T \cup E$, and the training set is defined as the complement of $T \cup E$ within the full set of image indices.

The function also provides diagnostic output by printing the excluded subjects along with the sizes of the test and training splits. It then checks for any classes that appear in the test set but not in the training set by comparing the unique class labels present in each set. If such cases are found, it prints a message indicating which subjects are missing from the training set and returns their indices as part of the split information.

In the end, the function returns a list of splits, where each split is a list containing the training indices, test indices, and any indices corresponding to missing people (people in the test set that are not present in the train set. This is used just for diagnostic). This approach is particularly useful to ensure that some subjects are completely unseen during training and can help evaluating the generalization and robustness of the classifier.

```
create_random_splits_by_image <- function(image_ids, percent_test = 0.3, num_splits = 5, num_people_to_exclude = 1) {
  set.seed(1)
  unique_people <- unique(image_ids)  # Get unique class labels

  splits <- lapply(1:num_splits, function(i) {

    # Randomly select people to exclude from train
    excluded_people <- sample(unique_people, num_people_to_exclude)

    # Get indices of images belonging to excluded people
    excluded_indices <- which(image_ids %in% excluded_people)

    # Sample test images from the remaining people
    remaining_images <- setdiff(names(image_ids), names(image_ids)[excluded_indices])  # Remove excluded images
    test_images <- sample(remaining_images, round(length(remaining_images) * percent_test))

    # Get test and train indices
    test_indices <- which(names(image_ids) %in% test_images)  # Normal test split
    test_indices <- unique(c(test_indices, excluded_indices))  # Force excluded images into the test set
    train_indices <- setdiff(seq_along(image_ids), test_indices)  # Remaining are train

    cat("\nExcluded people: ", paste(excluded_people, collapse = ", "), "\n")
    cat("length test: ", length(test_indices), "\nlength train: ", length(train_indices), "\n")

    # Extract class labels for train and test sets
    train_people <- unique(image_ids[train_indices])
    test_people <- unique(image_ids[test_indices])

    # Check if any class is in test but not in train
    missing_people <- setdiff(test_people, train_people)
    missing_people_indices <- which(image_ids %in% missing_people)

    if (length(missing_people) > 0) {
      cat("The following people are in the test set but not in the train set: ", paste(missing_people, collapse = ", "),
'\n')
    }

    return(list(train = train_indices, test = test_indices, missing_people = missing_people_indices))
  })

  return(splits)
}
```

4.1. Run KNN Clssification

The following functions will be used to find the best parameters (explained later).

`run_knn_classification` is a helper function used to implement the grid search part that is common for the KNN face recognizer that uses PCA projections from the one that it doesn't. It iterates over a grid of parameters (metric, k, threshold). It computes the known and unknown accuracies. The accuracy of people from the test set that belong to the train set, and the accuracy of people from the test set that are not present in the train set respectively. Then computes the balanced accuracy, that is the mean of the previous ones.

```
run_knn_classification <- function(Y_train, Y_test, image_ids_train, image_ids_test,
                                   metrics, param_grid, fold, n_pc, E_pc,  FDA_dims = NA, results) {
  for (metric in metrics) {
    if (!(metric %in% names(param_grid))) {
      stop(paste("Unknown metric:", metric))
    }

    for (k in param_grid$k_values) {
      for (threshold in param_grid[[metric]]) {
        cat("Testing Metric:", metric, "| k:", k, "| n_pc:", n_pc, "| fda_dims:", FDA_dims, "| threshold:", threshold, "\n")

        predictions <- sapply(1:nrow(Y_test), function(i) {
          knn_classifier(k = k, threshold = threshold, y_new = Y_test[i, ],
                         Y = Y_train, image_ids = image_ids_train, metric = metric, E = E_pc)
        })

        known_mask <- image_ids_test %in% image_ids_train
        unknown_mask <- !image_ids_test %in% image_ids_train

        known_accuracy <- ifelse(sum(known_mask) > 0, mean(predictions[known_mask] == image_ids_test[known_mask]), NA)
        unknown_accuracy <- ifelse(sum(unknown_mask) > 0, mean(predictions[unknown_mask] == 0), NA)
        balanced_accuracy <- mean(c(known_accuracy, unknown_accuracy), na.rm = TRUE)


        results <- rbind(results, data.frame(Fold = fold, n_pcs = n_pc,
                                             Metric = metric, k = k, Threshold = threshold,
                                             Known_Accuracy = known_accuracy, Unknown_Accuracy = unknown_accuracy,
                                             Balanced_Accuracy = balanced_accuracy, FDA_dims = FDA_dims))
      }
    }
  }
  return(results)
}
```

4.2. Find Best Parameters

Our function `find_best_params` plays a pivotal role in our grid search for optimal classifier parameters by separating the parts of the search process that differ between classifiers that use PCA and those that do not. This function receives a boolean parameter `compute_pca` which, if TRUE, instructs us to compute PCA using only the training images, and then to project all the images into the PCA space. We then iterate over the number of principal components specified in the parameter grid and call `run_knn_classification` for each set of components to continue with the grid search and obtain classification results. Moreover, if the boolean parameter `apply_fda` is set to TRUE, after computing the PCA projection we further apply FDA on the PCA matrix by appending the class labels and computing FDA on these principal components. It is important to note that when `apply_fda` is TRUE, only the `modified_sse` metric is used because employing either the eigenvalues from the FDA or the PCA does not make sense in that context. On the other hand, if `compute_pca` is FALSE, we also force the metric to be `modified_sse` since the other metrics rely on eigenvalues derived from PCA. In either case, `find_best_params` ultimately calls `run_knn_classification` to continue with the grid search.

```
find_best_params <- function(M, image_ids, metrics, param_grid, num_folds = 5,
                             num_people_to_exclude = 1, compute_pca = TRUE, apply_fda = FALSE) {
  # Create random splits based on image IDs
  folds <- create_random_splits_by_image(image_ids, num_splits = num_folds,
                                         num_people_to_exclude = num_people_to_exclude)
  results <- list()

  for (fold in seq_along(folds)) {
    test_indices <- folds[[fold]]$test
    train_indices <- folds[[fold]]$train

    cat("Fold:", fold, "\n")

    if (compute_pca || apply_fda) {
      cat("Computing PCA...\n")
      pca_results <- compute_pca(M[train_indices, ])
      mean_obs <- pca_results$mean_obs
      P <- pca_results$P
      E <- pca_results$E
      cat("PCA computed. Projecting data...\n")
      # Project data using PCA (all available PCA components)
      Y <- sweep(M, 2, mean_obs, FUN = "-") %*% P
    } else {
      cat("Skipping PCA. Using original data...\n")
      Y <- M  # Use original data
    }

    cat("dim(Y):", dim(Y), "\n")

    # If PCA is not computed, restrict metrics to 'modified_sse'
    if (!compute_pca || apply_fda == TRUE) metrics <- c("modified_sse")



    # Process classification using the appropriate representation
    if (compute_pca || apply_fda) {
      for (n_pc in param_grid$n_pcs) {
        # Use the first n_pc PCA components
        Y_pc <- Y[, 1:n_pc, drop = FALSE]

        if (apply_fda) {
          cat("Applying FDA on PCA matrix with", n_pc, "components...\n")
          # Append class labels to Y_pc and compute FDA on these n_pc components
          Y_pc_with_class <- cbind(Y_pc, class = image_ids)
          fda_results <- fda(Y_pc_with_class)

          # Project Y_pc into FDA space
          FDA_proj <- Y_pc %*% fda_results$P_fda

          # Loop over the FDA dimensions specified in the grid
          for (n_fda in param_grid$fda_dims) {
            # Skip FDA dimension if it exceeds the number of PCA components used
            if (n_fda > n_pc) next

            FDA_dims <- n_fda  # Use the loop variable directly.
            Y_used <- FDA_proj[, 1:FDA_dims, drop = FALSE]
            E_used <- fda_results$E_fda[1:FDA_dims, drop = FALSE]

            cat("Using", n_pc, "PCA components and", FDA_dims, "FDA components.\n")

            Y_train <- Y_used[train_indices, , drop = FALSE]
            Y_test <- Y_used[test_indices, , drop = FALSE]
            image_ids_train <- image_ids[train_indices]
            image_ids_test <- image_ids[test_indices]

            results <- run_knn_classification(Y_train, Y_test, image_ids_train, image_ids_test,
                                              metrics, param_grid, fold, n_pc, E_used, FDA_dims, results)
          }
        } else {
          # If only PCA is computed, use the PCA subspace directly
          Y_used <- Y_pc
          E_used <- E[1:n_pc, drop = FALSE]
          n_used <- n_pc

          Y_train <- Y_used[train_indices, , drop = FALSE]
          Y_test <- Y_used[test_indices, , drop = FALSE]
          image_ids_train <- image_ids[train_indices]
          image_ids_test <- image_ids[test_indices]

          results <- run_knn_classification(Y_train, Y_test, image_ids_train, image_ids_test,
                                            metrics, param_grid, fold, n_pc, E_used, FDA_dims = NA, results)

        }
      }
    } else {
      # If no PCA is computed, use the original data directly.
      Y_train <- Y[train_indices, , drop = FALSE]
      Y_test <- Y[test_indices, , drop = FALSE]
      image_ids_train <- image_ids[train_indices]
      image_ids_test <- image_ids[test_indices]

      results <- run_knn_classification(Y_train, Y_test, image_ids_train, image_ids_test,
                                        metrics, param_grid, fold, n_pc = NA, E_pc = NULL, FDA_dims = NA, results)
    }
  }
```

```
  return(results)
}
```

4.3. Get Best Parameters

The function `get_best_params` is used to handle the grid search for optimal classifier parameters, either for finding the best threshold for each distance metric or for determining the best values of k and `n_pc` (and, if applicable, FDA dimensions) by fold. This function accepts several parameters:

- `best_threshold` : Boolean indicating if we are looking for the best threshold or not.
- `M` : Matrix with the raw RGB image representation. One image by row.
- `image_ids` : ids of the images in the dataset.
- `param_grid` : Grid of parameters to try.
- `num_folds` : Number of train/test splits to generate.
- `num_people_to_exclude` : Number of people whose images will be excluded from the test set and included in the test.
- `compute_pca` : Boolean indicating whether we compute the PCA and project the images, or we just use the raw image representation.
- `apply_fda` : Boolean indicating whether to apply FDA on the PCA projection.

If `best_threshold` is TRUE, the function returns a data structure that includes a data frame containing the best threshold per fold and per metric, another data frame with the average best threshold by metric, and a simpler, easily accessible vector of the best thresholds per metric. In this mode, the function identifies, for each fold and each metric, the threshold that yields the highest balanced accuracy, and then aggregates these results across folds.

On the other hand, if `best_threshold` is FALSE, the function focuses on finding the optimal k (number of neighbors) and `n_pc` (number of principal components) by fold. It returns a data frame that contains the best k and `n_pc` values for each fold, a summary data frame that provides the average best k, average best `n_pc` , and average accuracy scores per metric, and additional information about which metric was the best (i.e., the one that performed best in the most folds) along with the average k and `n_pc` for that metric. Importantly, when `apply_fda` is set to TRUE, the function adds the same information about FDA dimensions as the information gathered for `n_pc` .

```r
get_best_params <- function(best_threshold = TRUE, M, image_ids, metrics, param_grid,
                            num_folds = 5, num_people_to_exclude = 1,
                            compute_pca = TRUE, apply_fda = FALSE) {

  # Compute the parameters using either PCA only or PCA followed by FDA (if apply_fda is TRUE)
  results <- find_best_params(M = M, image_ids = image_ids, metrics = metrics,
                              param_grid = param_grid, num_folds = num_folds,
                              num_people_to_exclude = num_people_to_exclude,
                              compute_pca = compute_pca, apply_fda = apply_fda)

  if (best_threshold) {
    # Step 1: Get the best threshold per fold and metric
    best_thresholds_per_fold <- results %>%
      group_by(Fold, Metric) %>%
      slice_max(Balanced_Accuracy, n = 1, with_ties = FALSE) %>%  # Select best threshold per fold and metric
      select(Fold, Metric, Threshold, Balanced_Accuracy) %>%
      ungroup()

    # Step 2: Compute the average threshold per metric
    average_best_thresholds <- best_thresholds_per_fold %>%
      group_by(Metric) %>%
      summarise(Average_Threshold = mean(Threshold, na.rm = TRUE)) %>%
      ungroup()

    # Convert to a named vector for easy access
    best_thresholds_vector <- average_best_thresholds %>%
      pull(Average_Threshold, name = Metric)

    return(list(best_thresholds_per_fold = best_thresholds_per_fold,
                average_best_thresholds = average_best_thresholds,
                best_thresholds = best_thresholds_vector))
  } else {
    # Step 1: Get the best row per fold based on Balanced Accuracy
    best_per_fold <- results %>%
      group_by(Fold) %>%
      slice_max(Balanced_Accuracy, with_ties = FALSE) %>%  # Select row with highest Balanced Accuracy per fold
      ungroup()

    print(best_per_fold)

    # Step 2: Compute the average k, n_pcs, and (if applicable) FDA_dims by Metric
    average_k_npcs_by_metric <- best_per_fold %>%
      group_by(Metric) %>%
      summarise(
        Occurrences = n(),
        Average_k = round(mean(k, na.rm = TRUE)),
        Average_n_pcs = round(mean(n_pcs, na.rm = TRUE)),
        Average_Known_Accuracy = mean(Known_Accuracy, na.rm = TRUE),
        Average_Unknown_Accuracy = mean(Unknown_Accuracy, na.rm = TRUE),
        Average_Balanced_Accuracy = mean(Balanced_Accuracy, na.rm = TRUE),
        Average_fda_dims = if (apply_fda) round(mean(FDA_dims, na.rm = TRUE)) else NA
      ) %>%
      ungroup()

    print(average_k_npcs_by_metric)

    # Step 3: Find the best metric (here we choose the metric with the highest occurrence count)
    best_metric <- average_k_npcs_by_metric %>%
      slice_max(Occurrences, with_ties = FALSE) %>%
      pull(Metric)

    # Step 4: Compute the average k and n_pcs ONLY for the best metric
    final_avg_k <- average_k_npcs_by_metric %>%
      filter(Metric == best_metric) %>%
      pull(Average_k) %>%
      as.integer()
    cat("final_avg_k: ", final_avg_k, "\n")

    final_avg_n_pcs <- average_k_npcs_by_metric %>%
      filter(Metric == best_metric) %>%
      pull(Average_n_pcs) %>%
      as.integer()
    cat("final_avg_n_pcs: ", final_avg_n_pcs, "\n")

    final_avg_fda_dims <- if (apply_fda) {
      average_k_npcs_by_metric %>%
        filter(Metric == best_metric) %>%
        pull(Average_fda_dims) %>%
        as.integer()
    } else {
      NA
    }
    cat("final_avg_fda_dims: ", final_avg_fda_dims, "\n")

    return(list(best_per_fold = best_per_fold,
                average_k_npcs_by_metric = average_k_npcs_by_metric,
                best_metric = best_metric,
                final_avg_k = final_avg_k,
                final_avg_n_pcs = final_avg_n_pcs,
                final_avg_fda_dims = final_avg_fda_dims))
  }
}
```

5. PCA-KNN

5.1. Threshold selection

As it's explained before, the rest of parameters will be fixed on this stage. The ranges of thresholds for each metric were selected by trying the distances manually between images of the same person and between images of different person. We took the min and the max distance among different trials, and the ranges are made so those min and max values found are included.

```
print("You will not see the text output.")
```

```
## [1] "You will not see the text output."
```

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best threshold...\n")
  param_grid_thresholds <- list(
    n_pcs = c(30),  # Fixed n_pcs
    k_values = c(5),  # Fixed k
    modified_sse = seq(3e-5, 2.5e-4, by = 5e-6), # Threshold
    mahalanobis = seq(-6000, -1500, by = 100),# Threshold
    weighted_angle = seq(-0.07, -0.009, by = 0.005)# Threshold
  )

best_thresholds_pca_results <- get_best_params(best_threshold=TRUE, M,image_ids=image_ids, metrics=c('modified_sse', 'mahala
nobis', 'weighted_angle'), param_grid=param_grid_thresholds, num_folds = 15, num_people_to_exclude=2, compute_pca=TRUE)

saveRDS(best_thresholds_pca_results, "best_thresholds_pca_results.rds")
}
```

```
best_thresholds_pca_results <- readRDS("best_thresholds_pca_results.rds")

kable(best_thresholds_pca_results$best_thresholds_per_fold)
```

Fold	Metric	Threshold	Balanced_Accuracy
1	mahalanobis	-3.60e+03	0.6707317
1	modified_sse	3.50e-05	0.8170732
1	weighted_angle	-6.50e-02	0.6849593
2	mahalanobis	-2.90e+03	0.7439024
2	modified_sse	1.15e-04	0.8780488
2	weighted_angle	-5.00e-02	0.9024390
3	mahalanobis	-3.50e+03	0.7073171
3	modified_sse	8.50e-05	0.9146341
3	weighted_angle	-5.00e-02	0.9634146
4	mahalanobis	-3.20e+03	0.7073171
4	modified_sse	1.20e-04	0.8780488
4	weighted_angle	-4.50e-02	0.9268293
5	mahalanobis	-3.30e+03	0.6585366
5	modified_sse	1.00e-04	0.8658537
5	weighted_angle	-4.50e-02	0.9512195
6	mahalanobis	-3.50e+03	0.6585366
6	modified_sse	1.05e-04	0.8170732
6	weighted_angle	-5.50e-02	0.8069106
7	mahalanobis	-3.10e+03	0.7439024
7	modified_sse	8.00e-05	0.8414634
7	weighted_angle	-5.50e-02	0.8902439
8	mahalanobis	-2.70e+03	0.8048780
8	modified_sse	6.50e-05	0.8658537
8	weighted_angle	-5.00e-02	0.8902439
9	mahalanobis	-2.90e+03	0.7804878
9	modified_sse	1.10e-04	0.9024390
9	weighted_angle	-5.00e-02	0.9268293
10	mahalanobis	-3.00e+03	0.7804878
10	modified_sse	1.30e-04	0.9024390
10	weighted_angle	-6.00e-02	0.8658537
11	mahalanobis	-2.60e+03	0.8428571
11	modified_sse	8.50e-05	0.9000000
11	weighted_angle	-6.50e-02	0.7571429
12	mahalanobis	-2.60e+03	0.8048780
12	modified_sse	1.30e-04	0.8536585

Fold	Metric	Threshold	Balanced_Accuracy
12	weighted_angle	-4.50e-02	0.8902439
13	mahalanobis	-3.10e+03	0.7682927
13	modified_sse	9.00e-05	0.9146341
13	weighted_angle	-6.50e-02	0.8363821
14	mahalanobis	-2.50e+03	0.8857143
14	modified_sse	1.00e-04	0.9428571
14	weighted_angle	-5.50e-02	0.9428571
15	mahalanobis	-3.70e+03	0.7073171
15	modified_sse	5.00e-05	0.9146341
15	weighted_angle	-5.50e-02	0.9512195

```
kable(best_thresholds_pca_results$average_best_thresholds)
```

Metric	Average_Threshold
mahalanobis	-3.08e+03
modified_sse	9.33e-05
weighted_angle	-5.40e-02

```
best_thresholds_pca_results$best_thresholds
```

```
##    mahalanobis   modified_sse weighted_angle
##  -3.080000e+03   9.333333e-05  -5.400000e-02
```

5.2 Metric, k, n_pcs selection

In the previous step, we found the best threshold for each metric. So now we have fixed those values. We will iterate over the first 30 number of components. We chose that number because at any trial we did during the implementation we didn't find better results above that number, so in order to save computational time, we limit the number of components to try. The same argument applies to the decision of trying k up to 10.

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best params...\n")
  best_thresholds_pca_results <- readRDS("best_thresholds_pca_results.rds")
  best_thresholds_pca <- best_thresholds_pca_results$best_thresholds

  param_grid_others <- list(
    n_pcs = seq(10, 30, by=1),
    #n_pcs = c(4),
    k_values = seq(1, 10, by=1),
    #k_values = c(1),
    modified_sse = c(best_thresholds_pca["modified_sse"]),
    mahalanobis = c(best_thresholds_pca["mahalanobis"]),
    weighted_angle = c(best_thresholds_pca["weighted_angle"])
  )

  best_params_pca <- get_best_params(best_threshold=FALSE, M,image_ids=image_ids, metrics=c('modified_sse', 'mahalanobis',
'weighted_angle'), param_grid=param_grid_others, num_folds = 15, num_people_to_exclude=2, compute_pca=TRUE)

  saveRDS(best_params_pca, "best_params_pca.rds")
}
```

```
best_params_pca <- readRDS("best_params_pca.rds")

kable(best_params_pca$best_per_fold)
```

Fold	n_pcs	Metric	k	Threshold	Known_Accuracy	Unknown_Accuracy	Balanced_Accuracy	FDA_dims
1	16	modified_sse	1	9.33e-05	0.9024390	1	0.9512195	NA
2	12	modified_sse	1	9.33e-05	0.9756098	1	0.9878049	NA
3	21	weighted_angle	1	-5.40e-02	0.9756098	1	0.9878049	NA
4	24	weighted_angle	1	-5.40e-02	0.9512195	1	0.9756098	NA
5	25	weighted_angle	1	-5.40e-02	0.9512195	1	0.9756098	NA
6	15	modified_sse	1	9.33e-05	0.9268293	1	0.9634146	NA
7	26	weighted_angle	1	-5.40e-02	0.9756098	1	0.9878049	NA
8	18	modified_sse	1	9.33e-05	0.9512195	1	0.9756098	NA
9	22	modified_sse	1	9.33e-05	0.9512195	1	0.9756098	NA
10	14	modified_sse	1	9.33e-05	0.9512195	1	0.9756098	NA
11	18	modified_sse	1	9.33e-05	1.0000000	1	1.0000000	NA
12	24	weighted_angle	3	-5.40e-02	0.9512195	1	0.9756098	NA
13	26	weighted_angle	3	-5.40e-02	1.0000000	1	1.0000000	NA
14	29	weighted_angle	1	-5.40e-02	0.9428571	1	0.9714286	NA
15	26	weighted_angle	1	-5.40e-02	0.9512195	1	0.9756098	NA

```
kable(best_params_pca$average_k_npcs_by_metric)
```

Metric	Occurrences	Average_k	Average_n_pcs	Average_Known_Accuracy	Average_Unknown_Accuracy	Average_Balanced_Accuracy	Average_f
modified_sse	7	1	16	0.9512195	1	0.9756098	NA
weighted_angle	8	2	25	0.9623693	1	0.9811847	NA

```
best_params_pca$best_metric
```

```
## [1] "weighted_angle"
```

```
best_params_pca$final_avg_k
```

```
## [1] 2
```

```
best_params_pca$final_avg_n_pcs
```

```
## [1] 25
```

6. Compare PCA vs Original Image Representation (Find Best Parameters Original Image Representation)

6.1. Threshold selection

Same procedure as in PCA-KNN is implemented.

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best parameters...\n")
  param_grid_thresholds <- list(
    n_pcs = c(30),  # Fixed n_pcs
    k_values = c(5),  # Fixed k
    modified_sse = seq(from = 2.0e-07, to = 1.1e-05, length.out = 30), # Threshold
    mahalanobis = seq(-6000, -1500, by = 100),
    weighted_angle = seq(-0.07, -0.009, by = 0.005)
  )

  best_thresholds_no_pca_results <- get_best_params(best_threshold=TRUE, M,image_ids=image_ids, metrics=c('modified_sse', 'm
ahalanobis', 'weighted_angle'), param_grid=param_grid_thresholds, num_folds = 5, num_people_to_exclude=2, compute_pca=FALSE)

  saveRDS(best_thresholds_no_pca_results, "best_thresholds_no_pca_results.rds")
}
```

```
best_thresholds_no_pca_results <- readRDS("best_thresholds_no_pca_results.rds")

kable(best_thresholds_no_pca_results$best_thresholds_per_fold)
```

Fold	Metric	Threshold	Balanced_Accuracy
1	modified_sse	6.5e-06	0.7682927
2	modified_sse	8.4e-06	0.9024390
3	modified_sse	4.7e-06	0.9024390
4	modified_sse	5.4e-06	0.8780488
5	modified_sse	5.4e-06	0.8536585

```
kable(best_thresholds_no_pca_results$average_best_thresholds)
```

Metric	Average_Threshold
modified_sse	6.1e-06

```
best_thresholds_no_pca_results$best_thresholds
```

```
## modified_sse
## 6.084138e-06
```

6.2. Metric, k, n_pcs selection

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best parameters...\n")
  best_thresholds_no_pca_results <- readRDS("best_thresholds_no_pca_results.rds")
  best_thresholds_no_pca <- best_thresholds_no_pca_results$best_thresholds

  param_grid_others <- list(
    n_pcs = seq(10, 30, by=1),
    #n_pcs = c(4),
    k_values = seq(1, 10, by=1),
    #k_values = c(1),
    modified_sse = c(best_thresholds_no_pca["modified_sse"]),
    mahalanobis = c(best_thresholds_no_pca["mahalanobis"]),
    weighted_angle = c(best_thresholds_no_pca["weighted_angle"])
  )

  best_params_no_pca <- get_best_params(best_threshold=FALSE, M,image_ids=image_ids, metrics=c('modified_sse', 'mahalanobi
s', 'weighted_angle'), param_grid=param_grid_others, num_folds = 5, num_people_to_exclude=2, compute_pca=FALSE)

  saveRDS(best_params_no_pca, "best_params_no_pca.rds")
}
```

```
best_params_no_pca <- readRDS("best_params_no_pca.rds")
kable(best_params_no_pca$best_per_fold)
```

Fold	n_pcs	Metric	k	Threshold	Known_Accuracy	Unknown_Accuracy	Balanced_Accuracy	FDA_dims
1	NA	modified_sse	1	6.1e-06	0.8292683	1	0.9146341	NA
2	NA	modified_sse	1	6.1e-06	0.8780488	1	0.9390244	NA
3	NA	modified_sse	1	6.1e-06	0.9512195	1	0.9756098	NA
4	NA	modified_sse	1	6.1e-06	0.8780488	1	0.9390244	NA
5	NA	modified_sse	1	6.1e-06	0.9024390	1	0.9512195	NA

```
kable(best_params_no_pca$average_k_npcs_by_metric)
```

Metric	Occurrences	Average_k	Average_n_pcs	Average_Known_Accuracy	Average_Unknown_Accuracy	Average_Balanced_Accuracy	Average_fda
modified_sse	5	1	NaN	0.8878049	1	0.9439024	NA

```
best_params_no_pca$best_metric
```

```
## [1] "modified_sse"
```

```
best_params_no_pca$final_avg_k
```

```
## [1] 1
```

7. PCA-FDA-KNN

7.1. Threshold selection

The same procedure as in PCA-KNN is implemented. In this case, we also fix `fda_dims` to save computational time. The chosen fixed value is 24 because in this case we have 25 classes, so setting a higher value will just add up noise.

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best threshold...\n")
  param_grid_thresholds <- list(
    n_pcs = c(30),  # Fixed n_pcs
    k_values = c(5),  # Fixed k
    fda_dims = c(24), #seq(10,24, by = 1)
    modified_sse = seq(3e-5, 2.5e-4, by = 5e-6), # Threshold
    mahalanobis = seq(-6000, -1500, by = 100),# Not used
    weighted_angle = seq(-0.07, -0.009, by = 0.005)# Not used
  )

best_thresholds_fda_results <- get_best_params(best_threshold=TRUE, M,image_ids=image_ids, metrics=c('modified_sse', 'mahala
nobis', 'weighted_angle'), param_grid=param_grid_thresholds, num_folds = 15, num_people_to_exclude=2, compute_pca=TRUE, appl
y_fda = TRUE)

saveRDS(best_thresholds_fda_results, "best_thresholds_fda_results.rds")
}
```

```
best_thresholds_fda_results <- readRDS("best_thresholds_fda_results.rds")

kable(best_thresholds_fda_results$best_thresholds_per_fold)
```

Fold	Metric	Threshold	Balanced_Accuracy
1	modified_sse	0.000065	0.8170732
2	modified_sse	0.000120	0.9146341
3	modified_sse	0.000180	0.9390244
4	modified_sse	0.000200	0.8902439

Fold	Metric	Threshold	Balanced_Accuracy
5	modified_sse	0.000190	0.8658537
6	modified_sse	0.000250	0.8536585
7	modified_sse	0.000215	0.8780488
8	modified_sse	0.000110	0.9146341
9	modified_sse	0.000080	0.8902439
10	modified_sse	0.000235	0.9024390
11	modified_sse	0.000130	0.8714286
12	modified_sse	0.000155	0.8292683
13	modified_sse	0.000140	0.8902439
14	modified_sse	0.000225	0.9571429
15	modified_sse	0.000245	0.9390244

```
kable(best_thresholds_fda_results$average_best_thresholds)
```

Metric	Average_Threshold
modified_sse	0.0001693

```
best_thresholds_fda_results$best_thresholds
```

```
## modified_sse
## 0.0001693333
```

7.2. Metric, k, n_pcs selection

In this next grid search the values of the `fda_dims` start in 10 instead of starting with a lower value to save some computational time. Also, `n_pcs` is capped to 24 (instead of 30) to reduce computational time, but if it is desired, the grid can be widen to obtain an even more precise result. Actually, the best option obviously will be to compute all the combinations in only one grid, but the need to separate it in two is also obvious, because it's too computationally expensive.

```
if(COMPUTE_GRID_SEARCH){
  cat("Finding best params...\n")
  if(COMPUTE_GRID_SEARCH){
    cat("Finding best params...\n")
    best_thresholds_fda_results <- readRDS("best_thresholds_fda_results.rds")
    best_thresholds_fda <- best_thresholds_fda_results$best_thresholds

    param_grid_others <- list(
      n_pcs = seq(10, 24, by=1),
      k_values = seq(1, 10, by=1),
      fda_dims = seq(10,24, by = 1),
      modified_sse = c(best_thresholds_fda["modified_sse"]),
      mahalanobis = c(best_thresholds_fda["mahalanobis"]),
      weighted_angle = c(best_thresholds_fda["weighted_angle"])
    )

    best_params_fda <- get_best_params(best_threshold=FALSE, M,image_ids=image_ids, metrics=c('modified_sse', 'mahalanobis',
'weighted_angle'), param_grid=param_grid_others, num_folds = 15, num_people_to_exclude=2, compute_pca=TRUE, apply_fda = TRU
E)

    saveRDS(best_params_fda, "best_params_fda.rds")
  }
}
```

```
best_params_fda <- readRDS("best_params_fda.rds")

kable(best_params_fda$best_per_fold)
```

Fold	n_pcs	Metric	k	Threshold	Known_Accuracy	Unknown_Accuracy	Balanced_Accuracy	FDA_dims
1	23	modified_sse	1	0.0001693	0.9756098	1	0.9878049	12
2	14	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10
3	17	modified_sse	1	0.0001693	1.0000000	1	1.0000000	12
4	22	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10
5	19	modified_sse	1	0.0001693	1.0000000	1	1.0000000	14
6	22	modified_sse	1	0.0001693	1.0000000	1	1.0000000	16
7	23	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10
8	15	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10
9	20	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10
10	22	modified_sse	1	0.0001693	1.0000000	1	1.0000000	12
11	13	modified_sse	1	0.0001693	1.0000000	1	1.0000000	12
12	23	modified_sse	1	0.0001693	0.9512195	1	0.9756098	10
13	23	modified_sse	1	0.0001693	1.0000000	1	1.0000000	13
14	22	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10

Fold	n_pcs	Metric	k	Threshold	Known_Accuracy	Unknown_Accuracy	Balanced_Accuracy	FDA_dims
15	20	modified_sse	1	0.0001693	1.0000000	1	1.0000000	10

```
kable(best_params_fda$average_k_npcs_by_metric)
```

Metric	Occurrences	Average_k	Average_n_pcs	Average_Known_Accuracy	Average_Unknown_Accuracy	Average_Balanced_Accuracy	Average_fda
modified_sse	15	1	20	0.995122	1	0.997561	

```
best_params_fda$final_avg_k
```

```
## [1] 1
```

```
best_params_fda$final_avg_n_pcs
```

```
## [1] 20
```

```
best_params_fda$final_avg_fda_dims
```

```
## [1] 11
```

8. Conclusions

The results show that applying PCA before KNN improves balanced accuracy (0.981 vs. 0.943) while significantly reducing computation time. Even if we expected a larger performance gap, the similarity in results suggests that KNN can still work well in high-dimensional spaces when properly tuned. Notice that the difference in balanced accuracy is not that big, but we can appreciate a bigger difference if we take a look to the known accuracy, suggesting that the model that is not using PCA, is finding more problems in correctly classifying the known faces. The PCA approach retains enough variance to achieve strong recognition while making training and inference more efficient.

We should choose the PCA-based KNN model because it achieves better accuracy while drastically reducing computation time and memory usage. It is more scalable, less prone to overfitting, and provides a more efficient representation of facial features without sacrificing recognition performance.

The grid search results under the FDA approach indicate that the optimal configuration to use is k = 1, 20 principal components, and 11 FDA dimensions. Specifically, the summary data reveals an average known accuracy of 0.995, an average unknown accuracy of 1, and an average balanced accuracy of 0.998. This demonstrates that applying FDA on top of PCA further refines our model's ability to distinguish between classes, leading to near-perfect recognition performance. Overall, these findings confirm that the PCA-based KNN model, especially when enhanced with FDA, improves recognition accuracy—particularly for known faces.