Face Recognition Using Eigenfaces

A Principal Component Analysis (PCA) Approach

Authors:

Kerem Göbekcioğlu and Kamil Duru

Contents

1 Introduction

Face recognition is an important area in computer vision. It has many applications, such as in security systems and user identification. One well-known method for face recognition is Principal Component Analysis (PCA). Eigenfaces, which is based on PCA, is a simple and effective approach to this task.

Eigenfaces uses PCA to reduce the size of the data while keeping the important information. It finds the main patterns in facial images and uses them for recognition. This makes the method fast and easy to understand. Unlike complex machine learning models, Eigenfaces works well with small datasets and needs fewer resources. Its simplicity and efficiency make it a great choice for face recognition tasks.

In this project, Eigenfaces was chosen because it is both effective and easy to implement. The method was tested on the Olivetti Faces dataset, a well-known dataset for face recognition. The process included preparing the data, calculating the mean face, reducing dimensions with PCA, and using the eigenfaces for recognizing new faces. The results showed good accuracy and the ability to handle unknown faces.

This report explains the steps of the implementation in detail. It also describes the custom functions written to replace some standard library methods, such as resizing images and performing matrix operations. These changes were made to better understand the process and meet the requirements of the project.

2 Methodology

2.1 Image Preprocessing Steps

2.1.1 Color Space Conversion

The initial preprocessing stage involves converting the input image into multiple color spaces to enhance face detection capabilities:

- **BGR to HSV Conversion:** The image is converted from BGR color space to HSV (Hue, Saturation, Value) color space, which provides better separation of color information from intensity.
- **BGR to YCbCr Conversion:** A parallel conversion to YCbCr (Luminance, Chrominance-Blue, Chrominance-Red) color space is performed, which is particularly effective for skin tone detection.

2.1.2 Skin Mask Creation

A skin detection mask is generated through the following process:

- Threshold values are defined for each color space to isolate skin-like regions
- Masks are created using in-range operations that segment the image based on these thresholds
- The HSV and YCbCr masks are combined using a bitwise AND operation
- This combination refines the skin region detection by leveraging the strengths of both color spaces

2.1.3 Morphological Operations

To enhance the quality of detected regions:

- **Erosion:** Applied to remove noise and small artifacts from the skin mask
- **Dilation:** Used to enhance the continuity of the detected regions
- These operations ensure better edge detection in subsequent steps

2.1.4 Edge and Contour Processing

The following steps are performed to identify face regions:

- **Canny Edge Detection:** Applied to the skin mask to identify the boundaries of potential face regions
- **Contour Extraction:** Contours are extracted from the edge-detected image
- **Bounding Box Calculation:** The largest bounding box enclosing the detected region is calculated

2.1.5 Region Processing

Final preprocessing steps include:

- **Region Refinement:** The bounding box is refined using horizontal and vertical narrowing factors
- **Image Extraction:** The cropped region is extracted from the original image
- **Grayscale Conversion:** The cropped face region is converted to grayscale
- **Standardization:** Image is resized to fixed dimensions (100×100 pixels)
- **Normalization:** Pixel values are normalized to a range of 0 to 1
- **Vectorization:** The image is flattened into a one-dimensional vector

2.2 Face Recognition

2.2.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical technique used to reduce the dimensionality of data while preserving as much variability as possible. In face recognition, PCA helps identify the most significant features of a face image, known as the principal components.

Steps of PCA

- **Data Collection:** Gather a set of face images, where each image is represented as a vector. If there are Q images, and each image is of size $N \times M$ pixels, each image can be vectorized into a single vector of size $P = N \times M$. The dataset can then be represented as a matrix $D \in \mathbb{R}^{P \times Q}$, where each column of D corresponds to a vectorized image.

- **Mean Calculation:** Compute the mean face vector by averaging all the face vectors:

$$\mu = \frac{1}{Q} \sum_{i=1}^{Q} D_i \tag{1}$$

Equation 1: Mean Face Vector

where D_i is the i-th face vector. This vector μ represents the average image.

- **Covariance Matrix:** Subtract the mean face vector from each face vector to obtain the centered data:

$$X = D - \mu \tag{2}$$

Equation 2: Centered Data

Then, compute the covariance matrix C of the centered data. The covariance matrix captures the relationships between different pixels across all images:

$$C = \frac{1}{Q} X X^T \tag{3}$$

Equation 3: Covariance Matrix

where $X \in \mathbb{R}^{P \times Q}$ is the centered data matrix.

- **Eigenvalues and Eigenvectors:** Calculate the eigenvalues λ_i and eigenvectors v_i of the covariance matrix C using the equation:

$$C v_i = \lambda_i v_i \tag{4}$$

Equation 4: Eigenvalue Decomposition

where v_i represents the eigenvector corresponding to the eigenvalue λ_i. These eigenvectors represent the directions of maximum variance in the dataset, while the eigenvalues indicate the magnitude of variance in those directions.

- **Principal Components:** Select the top eigenvectors corresponding to the largest eigenvalues. These eigenvectors form the principal components, which are used to project the original face images into a lower-dimensional space. This reduces the dimensionality of the dataset while retaining the most important features.

2.2.2 Eigenfaces Method

The Eigenfaces method, developed by Turk and Pentland, is a face recognition technique that uses PCA to identify and represent significant features of face images. The core idea is to represent each face image as a linear combination of a set of basis images, known as eigenfaces.

Steps of the Eigenfaces Method

Training Phase:

- **Image Representation:** Represent each face image as a vector by flattening the 2D image into a 1D vector. This creates a dataset matrix $D \in \mathbb{R}^{P \times Q}$, where each column represents a vectorized image.

- **Mean Face:** Compute the mean face vector μ as described in PCA.

- **Centered Data:** Subtract the mean face vector μ from each face vector to obtain the centered data matrix X.
- **Covariance Matrix:** Compute the covariance matrix

$$C = \frac{1}{Q} X X^T \tag{5}$$

Equation 5: Training Phase Covariance Matrix

- **Eigenfaces:** Calculate the eigenvalues and eigenvectors of the covariance matrix C. Select the top eigenvectors (those corresponding to the largest eigenvalues). These eigenvectors are reshaped back into 2D images and form the eigenfaces. The equation for this step is:

$$C v_i = \lambda_i v_i \tag{6}$$

Equation 6: Eigenfaces Calculation

where v_i represents the eigenvectors, reshaped as eigenfaces.

- **Projection:** Project each centered face vector onto the eigenfaces to obtain its representation in the lower-dimensional space. The projection is done by calculating the dot product between the face vector and each eigenface:

$$w_i = E^T X_i \tag{7}$$

Equation 7: Face Projection

where E is the matrix of eigenfaces, and w_i is the coefficient vector that represents the face image in the face space.

Recognition Phase:

- **Image Representation:** Represent the test face image as a vector and subtract the mean face vector.
- **Projection:** Project the centered test face vector onto the eigenfaces to obtain its representation in the lower-dimensional space, as done in the training phase.
- **Comparison:** Compare the test face's representation with the representations of the training faces using a distance metric, such as Euclidean distance:

$$\text{Distance} = \|w_{\text{test}} - w_i\| \tag{8}$$

Equation 8: Euclidean Distance for Face Recognition

where w_{test} is the representation of the test image, and w_i is the representation of the i-th training image. The test face is recognized as the training face with the closest representation.

2.2.3 Mathematical Foundation

The mathematical foundation of the Eigenfaces method is rooted in linear algebra and statistics, specifically in the computation of eigenvalues, eigenvectors, and covariance matrices.

- **Eigenvalues and Eigenvectors:** In PCA, the eigenvalues and eigenvectors of the covariance matrix are used to identify the principal components. The eigenvectors represent the directions of maximum variance, and the eigenvalues indicate the magnitude of variance in those directions.
- **Covariance Matrix:** The covariance matrix captures the relationships between the different dimensions (pixels) of the data. It is computed as the product of the centered data matrix and its transpose:

$$C = \frac{1}{Q} X X^T \tag{3}$$

Equation 3: Covariance Matrix

- **Projection:** The projection of a face vector onto the eigenfaces is achieved by computing the dot product between the face vector and each eigenface:

$$w_i = E^T X_i \tag{7}$$

Equation 7: Face Projection

By utilizing PCA and the Eigenfaces method, we can effectively reduce the dimensionality of face images while retaining the most significant features for face recognition.

3 Implementation

The face recognition system is implemented using Eigenfaces, a method based on Principal Component Analysis (PCA) to represent faces in a lower-dimensional space. This section outlines the implementation steps involved in building the system.

3.1 Face Detection and Preprocessing

The goal of this part of the implementation is to detect faces in images and preprocess them by resizing and converting them to grayscale.

3.2 Image Conversion Functions

Several utility functions are implemented to process images in various ways.

3.2.1 `bgr_to_hsv`

Purpose: Convert an image from BGR to HSV color space.

```python
def bgr_to_hsv(image):
    """
    Convert an image from BGR to HSV color space.

    Args:
        image (numpy.ndarray): Input BGR image.

    Returns:
```

```python
        numpy.ndarray: HSV image.
    """
    hsv_image = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            b, g, r = image[i, j] / 255.0
            max_val = max(b, g, r)
            min_val = min(b, g, r)
            delta = max_val - min_val

            if delta == 0:
                h = 0
            elif max_val == r:
                h = (60 * ((g - b) / delta) + 360) % 360
            elif max_val == g:
                h = (60 * ((b - r) / delta) + 120) % 360
            elif max_val == b:
                h = (60 * ((r - g) / delta) + 240) % 360

            s = 0 if max_val == 0 else (delta / max_val)
            v = max_val

            hsv_image[i, j] = [h, s, v]

    hsv_image[:, :, 0] = hsv_image[:, :, 0] / 2
    hsv_image[:, :, 1:] *= 255
    return hsv_image.astype(np.uint8)
```

Explanation: This function converts an image's color space from BGR (Blue-Green-Red) to HSV (Hue-Saturation-Value), which is useful for tasks like skin tone detection and segmentation.

3.2.2 bgr_to_ycrcb

Purpose: Convert an image from BGR to YCrCb color space.

```python
def bgr_to_ycrcb(image):
    """
    Convert an image from BGR to YCrCb color space.

    Args:
        image (numpy.ndarray): Input BGR image.

    Returns:
        numpy.ndarray: YCrCb image.
    """
    ycrcb_image = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            b, g, r = image[i, j]
            y = 0.299 * r + 0.587 * g + 0.114 * b
            cr = (r - y) * 0.713 + 128
```

```
            cb = (b - y) * 0.564 + 128
            ycrcb_image[i, j] = [y, cr, cb]
    ycrcb_image = np.clip(ycrcb_image, 0, 255)
    return ycrcb_image.astype(np.uint8)
```

Explanation: The YCrCb color space is used for luminance and chrominance analysis, often helpful in skin detection and color segmentation tasks.

3.3 Edge Detection

Edge detection is a crucial step in identifying facial features and contours.

3.3.1 canny

Purpose: Perform Canny edge detection on an image.

```
def canny(image, low_threshold, high_threshold):
    """
    Apply Canny edge detection to an image.

    Args:
        image (numpy.ndarray): Input image.
        low_threshold (int): Lower threshold for hysteresis.
        high_threshold (int): Upper threshold for hysteresis.

    Returns:
        numpy.ndarray: Image with edges detected.
    """
    blurred_image = gaussian_filter(image, sigma=1.4)
    grad_x = sobel(blurred_image, axis=0)
    grad_y = sobel(blurred_image, axis=1)
    gradient_magnitude = np.hypot(grad_x, grad_y)
    gradient_direction = np.arctan2(grad_y, grad_x) * (180 / np.
       pi)
    gradient_direction[gradient_direction < 0] += 180

    nms_image = np.zeros_like(gradient_magnitude)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            angle = gradient_direction[i, j]
            q = r = 255

            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                q = gradient_magnitude[i, j + 1]
                r = gradient_magnitude[i, j - 1]
            elif 22.5 <= angle < 67.5:
                q = gradient_magnitude[i + 1, j - 1]
                r = gradient_magnitude[i - 1, j + 1]
            elif 67.5 <= angle < 112.5:
                q = gradient_magnitude[i + 1, j]
                r = gradient_magnitude[i - 1, j]
            elif 112.5 <= angle < 157.5:
```

```python
                q = gradient_magnitude[i - 1, j - 1]
                r = gradient_magnitude[i + 1, j + 1]

            if gradient_magnitude[i, j] >= q and
               gradient_magnitude[i, j] >= r:
                nms_image[i, j] = gradient_magnitude[i, j]

    strong_edges = (nms_image > high_threshold).astype(np.uint8)
    weak_edges = ((nms_image >= low_threshold) & (nms_image <=
       high_threshold)).astype(np.uint8)

    edges = np.zeros_like(image, dtype=np.uint8)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            if strong_edges[i, j]:
                edges[i, j] = 255
            elif weak_edges[i, j]:
                if (strong_edges[i + 1, j - 1:j + 2].any() or
                    strong_edges[i - 1, j - 1:j + 2].any() or
                    strong_edges[i, [j - 1, j + 1]].any()):
                    edges[i, j] = 255

    return edges
```

Explanation: This function applies the Canny edge detection algorithm to identify the boundaries of objects in the image, which is crucial for extracting face contours.

3.4 Morphological Operations

Morphological operations are used to refine the detected masks.

3.4.1 `erode` and `dilate`

Purpose: Perform erosion and dilation on an image using a specified kernel.

```python
def erode(image, kernel, iterations=1):
    """
    Apply erosion to an image.

    Args:
        image (numpy.ndarray): Input image.
        kernel (numpy.ndarray): Structuring element.
        iterations (int): Number of iterations.

    Returns:
        numpy.ndarray: Eroded image.
    """
    img_h, img_w = image.shape
    k_h, k_w = kernel.shape
    pad_h, pad_w = k_h // 2, k_w // 2
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w))
       , mode='constant', constant_values=255)
```

```python

    for _ in range(iterations):
        eroded_image = np.copy(image)
        for i in range(img_h):
            for j in range(img_w):
                roi = padded_image[i:i + k_h, j:j + k_w]
                if np.all(roi[kernel == 1] == 255):
                    eroded_image[i, j] = 255
                else:
                    eroded_image[i, j] = 0
        padded_image = np.pad(eroded_image, ((pad_h, pad_h), (
           pad_w, pad_w)), mode='constant', constant_values=255)
    return eroded_image


def dilate(image, kernel, iterations=1):
    """
    Apply dilation to an image.

    Args:
        image (numpy.ndarray): Input image.
        kernel (numpy.ndarray): Structuring element.
        iterations (int): Number of iterations.

    Returns:
        numpy.ndarray: Dilated image.
    """
    img_h, img_w = image.shape
    k_h, k_w = kernel.shape
    pad_h, pad_w = k_h // 2, k_w // 2
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w))
       , mode='constant', constant_values=0)

    for _ in range(iterations):
        dilated_image = np.copy(image)
        for i in range(img_h):
            for j in range(img_w):
                roi = padded_image[i:i + k_h, j:j + k_w]
                if np.any(roi[kernel == 1] == 255):
                    dilated_image[i, j] = 255
        padded_image = np.pad(dilated_image, ((pad_h, pad_h), (
           pad_w, pad_w)), mode='constant', constant_values=0)
    return dilated_image
```

Explanation: Erosion and dilation are basic morphological operations used to remove noise and fill gaps in the detected masks.

3.5 Image Preprocessing and Dataset Preparation

The first step in the process is to load and resize the images. For this, we utilize a dataset containing grayscale images of multiple individuals, each having several images.

The images are resized to a target size of 100x100 pixels, and pixel values are normalized to a range of [0, 1] to standardize the input data.

3.5.1 Loading and Resizing Images

The dataset is loaded by iterating over the directory structure, where each subdirectory corresponds to a person and contains images of that person. Each image is read, resized to the target dimensions using a custom resizing algorithm, and stored along with its label (the person's name). The custom resizing function ensures manual computation of pixel values, avoiding the use of pre-built OpenCV functions.

```python
def load_faces(root_dir, target_size=(100, 100)):
    """
    Loads and resizes images from the dataset.

    Parameters:
        root_dir (str): Path to the dataset root directory.
        target_size (tuple): Target size for resizing the images (width, height).

    Returns:
        image_paths (list): List of paths to the loaded images.
        labels (list): List of labels corresponding to the images.
    """
    image_paths = []
    labels = []
    for person_dir in tqdm(os.listdir(root_dir), desc="Processing directories"):
        person_path = os.path.join(root_dir, person_dir)
        if os.path.isdir(person_path):
            for filename in os.listdir(person_path):
                if filename.endswith(('.jpg', '.jpeg', '.png')):
                    full_path = os.path.join(person_path, filename)
                    image = cv2.imread(full_path, cv2.IMREAD_GRAYSCALE)
                    resized_image = resize(image, target_size)
                    image_paths.append(full_path)
                    labels.append(person_dir)
    return image_paths, labels
```

3.5.2 Preprocessing Images

After loading the images, they are flattened into 1D arrays and normalized to the range [0, 1]. This transformation reduces dimensionality and ensures consistency for subsequent steps.

```python
def preprocess_images(image_paths):
    """
    Preprocesses images by flattening and normalizing them.
```

```

    Parameters:
        image_paths (list): List of paths to the images.

    Returns:
        dataset (numpy.ndarray): Flattened and normalized images.
    """
    dataset = []
    for path in tqdm(image_paths, desc="Preprocessing images"):
        image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        flattened = (image.flatten() / 255.0)
        dataset.append(flattened)
    return np.array(dataset, dtype=np.float64)
```

3.6 Custom Functions

3.6.1 Computing the Dot Product

The dot product between two matrices is computed in a custom manner by iterating over rows and columns, ensuring efficient calculation even for larger matrices.

```
def custom_dot(a, b):
    """
    Computes the dot product between two matrices a and b.

    Parameters:
        a (numpy.ndarray): First matrix, where each row is a data vector.
        b (numpy.ndarray): Second matrix, where each column is a data vector.

    Returns:
        numpy.ndarray: Result of the dot product.
    """
    result = []
    for a_row in tqdm(a, desc="Computing dot product (outer loop)"):
        row_result = []
        for b_col in tqdm(zip(*b), desc="Computing dot product (inner loop)", leave=False):
            row_result.append(sum(x * y for x, y in zip(a_row, b_col)))
        result.append(row_result)
    return np.array(result)
```

3.6.2 Normalizing the Image

Image normalization adjusts the pixel intensity range of an image to a specified range, such as $[0, 255]$, to ensure consistent processing.

```python
def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.
    NORM_MINMAX):
    """
    Normalizes the input image.

    Parameters:
        src (numpy.ndarray): Input image to be normalized.
        dst (numpy.ndarray, optional): Output image. If None, a
            new array is created.
        alpha (int): Minimum intensity value for normalization.
        beta (int): Maximum intensity value for normalization.
        norm_type (int): Type of normalization (default is cv2.
            NORM_MINMAX).

    Returns:
        numpy.ndarray: Normalized image.
    """
    if dst is None:
        dst = np.zeros_like(src)

    if norm_type == cv2.NORM_MINMAX:
        min_val = np.min(src)
        max_val = np.max(src)
        dst = (src - min_val) * (beta - alpha) / (max_val -
            min_val) + alpha
    else:
        raise NotImplementedError("Only NORM_MINMAX is
            implemented")

    return dst
```

3.6.3 Resizing the Image

Resizing changes the dimensions of an image while maintaining its aspect ratio, using bilinear interpolation to smooth the result.

```python
def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
    """
    Resizes the input image to the specified dimensions.

    Parameters:
        src (numpy.ndarray): Input image to be resized.
        dsize (tuple): New dimensions (width, height) for
            resizing.
        interpolation (int): Interpolation method for resizing (
            default is cv2.INTER_LINEAR).

    Returns:
        numpy.ndarray: Resized image.
    """
    src_height, src_width = src.shape[:2]
```

```
    dst_width, dst_height = dsize
    dst = np.zeros((dst_height, dst_width), dtype=src.dtype)

    for i in range(dst_height):
        for j in range(dst_width):
            src_x = j * (src_width / dst_width)
            src_y = i * (src_height / dst_height)
            src_x0 = int(np.floor(src_x))
            src_y0 = int(np.floor(src_y))
            src_x1 = min(src_x0 + 1, src_width - 1)
            src_y1 = min(src_y0 + 1, src_height - 1)

            dx = src_x - src_x0
            dy = src_y - src_y0

            dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0]
                + \
                        dx * (1 - dy) * src[src_y0, src_x1] + \
                        (1 - dx) * dy * src[src_y1, src_x0] + \
                        dx * dy * src[src_y1, src_x1]

    return dst
```

3.6.4 Computing the Mean

This function computes the mean of a dataset, either across all values or along a specific axis.

```
def custom_mean(array, axis=None):
    """
    Computes the mean of a dataset.

    Parameters:
        array (numpy.ndarray): The input dataset.
        axis (int, optional): Axis along which the mean is
           computed (default is None for full array).

    Returns:
        float or numpy.ndarray: Mean of the dataset.
    """
    if axis is None:
        return sum(array) / len(array)
    else:
        return np.sum(array, axis=axis) / array.shape[axis]
```

3.7 Eigenfaces Computation

To represent faces in a reduced-dimensional space, Eigenface computation is performed using PCA. This involves calculating the mean face, centering the dataset, and performing eigenvalue decomposition on the covariance matrix.

3.7.1 Computing the Mean Face

The mean face is computed as the average of all image vectors in the training dataset. It serves as the baseline for centering the dataset.

```python
def compute_mean_face(dataset):
    """
    Computes the mean face from the dataset.

    Parameters:
        dataset (numpy.ndarray): Matrix of flattened images.

    Returns:
        mean_face (numpy.ndarray): Mean face vector.
        mean_face_image (numpy.ndarray): Reshaped and normalized mean face image.
    """
    mean_face = custom_mean(dataset, axis=0)
    mean_face_image = mean_face.reshape((100, 100))
    mean_face_image = normalize(mean_face_image, norm_type=cv2.NORM_MINMAX)
    return mean_face, mean_face_image
```

3.7.2 Centering the Dataset

Each image in the dataset is centered by subtracting the mean face, ensuring the data is zero-centered for PCA.

```python
def center_dataset(dataset, mean_face):
    """
    Centers the dataset by subtracting the mean face.

    Parameters:
        dataset (numpy.ndarray): Matrix of flattened images.
        mean_face (numpy.ndarray): Mean face vector.

    Returns:
        centered_data (numpy.ndarray): Centered dataset.
    """
    return dataset - mean_face
```

3.7.3 Computing the Covariance Matrix

The covariance matrix is calculated to capture the relationships between pixel intensities across the dataset. This is achieved using a custom dot product function.

```python
def compute_covariance_matrix(centered_data):
    """
    Computes the covariance matrix for the centered dataset.

    Parameters:
        centered_data (numpy.ndarray): Centered dataset.
```

```

    Returns:
        covariance_matrix (numpy.ndarray): Covariance matrix.
    """
    return custom_dot(centered_data.T, centered_data) /
        centered_data.shape[0]
```

3.7.4 Eigenvalue Decomposition and Eigenfaces Computation

Eigenfaces are obtained by performing eigenvalue decomposition on the covariance matrix. The eigenvectors corresponding to the largest eigenvalues represent the most significant patterns in the dataset.

```
def compute_eigenfaces(centered_data, covariance_matrix,
    num_eigenfaces):
    """
    Computes the eigenfaces from the covariance matrix.

    Parameters:
        centered_data (numpy.ndarray): Centered dataset.
        covariance_matrix (numpy.ndarray): Covariance matrix.
        num_eigenfaces (int): Number of eigenfaces to compute.

    Returns:
        eigenfaces (numpy.ndarray): Computed eigenfaces.
        eigenvalues (numpy.ndarray): Corresponding eigenvalues.
    """
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvectors = eigenvectors[:, sorted_indices]
    eigenvalues = eigenvalues[sorted_indices]
    eigenfaces = custom_dot(eigenvectors.T, centered_data)
    return eigenfaces, eigenvalues[:num_eigenfaces]
```

3.7.5 Displaying Eigenfaces

The computed eigenfaces are visualized by reshaping them into 2D images and displaying them in a grid.

```
def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape
    =(100, 100), results_subdirectory=None):
    """
    Displays and optionally saves the computed eigenfaces.

    Parameters:
        eigenfaces (numpy.ndarray): Computed eigenfaces.
        num_eigenfaces (int): Number of eigenfaces to display.
        image_shape (tuple): Shape of each eigenface image.
        results_subdirectory (str): Directory to save the
            visualization.
    """
```

```
    grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
    grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))
    canvas = np.zeros((grid_rows * image_shape[0], grid_cols *
       image_shape[1]), dtype=np.uint8)
    for i in range(num_eigenfaces):
        row = i // grid_cols
        col = i % grid_cols
        eigenface = eigenfaces[i].reshape(image_shape)
        eigenface = normalize(eigenface, norm_type=cv2.
           NORM_MINMAX)
        canvas[row * image_shape[0]:(row + 1) * image_shape[0],
           col * image_shape[1]:(col + 1) * image_shape[1]] =
           eigenface
    cv2.imshow("Eigenfaces", canvas)
    cv2.waitKey(0)
    if results_subdirectory:
        eigenfaces_path = os.path.join(results_subdirectory, "
           eigenfaces.jpg")
        cv2.imwrite(eigenfaces_path, canvas)
```

3.8 Face Recognition

After computing eigenfaces, the test images are projected onto the eigenface space for recognition.

3.8.1 Projecting Faces onto Eigenfaces

Each test image is projected onto the eigenface space by calculating the dot product with the eigenfaces.

```
def project_faces(centered_data, eigenfaces):
    """
    Projects faces onto the eigenface space.

    Parameters:
        centered_data (numpy.ndarray): Centered dataset.
        eigenfaces (numpy.ndarray): Computed eigenfaces.

    Returns:
        projections (numpy.ndarray): Projected faces.
    """
    return custom_dot(centered_data, eigenfaces.T)
```

3.8.2 Recognizing Test Faces

Test faces are classified by comparing their projections with the projections of training faces. A threshold is used to determine if a face is recognized or labeled as "unknown."

```
def recognize_face(test_face, mean_face, eigenfaces,
   projected_faces, labels, original_faces, fixed_threshold=300):
    """
```

```python
    Recognizes a test face by comparing its projection with training projections.

    Parameters:
        test_face (numpy.ndarray): Flattened test face vector.
        mean_face (numpy.ndarray): Mean face vector.
        eigenfaces (numpy.ndarray): Computed eigenfaces.
        projected_faces (numpy.ndarray): Projections of training faces.
        labels (list): Labels of training faces.
        fixed_threshold (float): Threshold for classifying "unknown" faces.

    Returns:
        tuple: A tuple containing:
            - recognized_label (str): Label of the recognized face or "unknown."
            - combined_image (numpy.ndarray): The image displaying the test face, mean face, closest face, and reconstructed face side by side.
    """
    centered_test_face = test_face - mean_face
    centered_test_face = centered_test_face.reshape(1, -1)
    projected_test_face = custom_dot(centered_test_face, eigenfaces.T)
    distances = np.linalg.norm(projected_faces - projected_test_face, axis=1)
    min_distance = np.min(distances)
    recognized_label = labels[np.argmin(distances)]
    closest_face_index = np.argmin(distances)
    #threshold = np.percentile(distances, 10)
    # Use a fixed threshold
    threshold = fixed_threshold

    # Debugging: Print distances and threshold
    print(f"Threshold: {threshold}")
    print(f"Min distance: {min_distance}")

    if min_distance > threshold:
        recognized_label = "unknown"
    else:
        recognized_label = labels[closest_face_index]
    # Reconstruct the projected face
    # reconstructed_face = np.dot(projected_test_face, eigenfaces) + mean_face
    reconstructed_face = custom_dot(projected_test_face, eigenfaces) + mean_face

    # Display the test face, mean face, closest face, and reconstructed face side by side
    test_face_image = test_face.reshape((100,100))
```

```
    mean_face_image = mean_face.reshape((100,100))
    closest_face = original_faces[closest_face_index].reshape
        ((100,100))
    reconstructed_face_image = reconstructed_face.reshape
        ((100,100))

    # Normalize the images to the range [0, 255]
    test_face_image = normalize(test_face_image, norm_type=cv2.
        NORM_MINMAX)
    mean_face_image = normalize(mean_face_image, norm_type=cv2.
        NORM_MINMAX)
    closest_face = normalize(closest_face, norm_type=cv2.
        NORM_MINMAX)
    reconstructed_face_image = normalize(reconstructed_face_image
        , norm_type=cv2.NORM_MINMAX)

    # Convert to uint8 type
    test_face_image = test_face_image.astype(np.uint8)
    mean_face_image = mean_face_image.astype(np.uint8)
    closest_face = closest_face.astype(np.uint8)
    reconstructed_face_image = reconstructed_face_image.astype(np
        .uint8)

    # Resize images to be larger for better visibility
    test_face_image = resize(test_face_image, (200, 200))
    mean_face_image = resize(mean_face_image, (200, 200))
    closest_face = resize(closest_face, (200, 200))
    reconstructed_face_image = resize(reconstructed_face_image,
        (200, 200))

    # Concatenate images horizontally
    combined_image = np.hstack((test_face_image, mean_face_image,
        closest_face, reconstructed_face_image))

    # Add labels to the combined image
    labels = ["Test Face", "Mean Face", "Closest Face", "
        Reconstructed Face"]
    positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
    add_labels_to_image(combined_image, labels, positions)

    # Save the result
    return recognized_label , combined_image
```

3.9 Testing and Evaluation

Finally, the system is evaluated on a test dataset. Recognition accuracy is calculated as the percentage of correctly classified faces. The results are saved for further analysis.

```
def save_results(results, filename):
    """
    Saves recognition results to a file.
```

```python

    Parameters:
        results (list): List of recognition results.
        filename (str): File to save the results.
    """
    with open(filename, 'w') as f:
        for result in results:
            f.write(f"{result}\n")
```

The recognition accuracy and the number of unknown faces are printed at the end of the test.

$$\text{Recognition Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Test Set Size}} \times 100$$

In our case, the accuracy is measured and reported in percentage. Additionally, faces that could not be recognized are labeled as "unknown."

4 Results

4.1 Image Preprocessing and Face Cropping

The preprocessing step was applied successfully to the dataset of facial images. Each image was resized to the target dimensions (100x100 pixels) and converted to grayscale. The following results were observed:

- The dataset was loaded successfully with a total of **85 images**.
- Images were resized uniformly to maintain consistency across the dataset.
- Grayscale conversion reduced the computational load for later steps.

4.2 Skin Detection

Skin detection was carried out using HSV and YCrCb color space thresholds, with additional morphological operations to clean the skin masks:

- Skin detection worked well, identifying facial regions with high accuracy under various lighting conditions.
- Erosion and dilation reduced noise in the masks, creating smoother results.
- The resulting skin masks were used for edge detection to extract facial contours.

4.3 Edge Detection and Contour Analysis

Facial contours were identified using the Canny edge detection algorithm:

- Edge detection highlighted the boundaries of facial features effectively.
- Contour analysis found the largest contour, corresponding to the face, in each image.
- The cropping algorithm isolated facial regions, achieving a cropping accuracy of **87%**.

4.4 Overall Processing Results

The preprocessing pipeline successfully prepared the dataset for further analysis, including the extraction and cropping of facial regions. The following statistics summarize the processing outcomes:

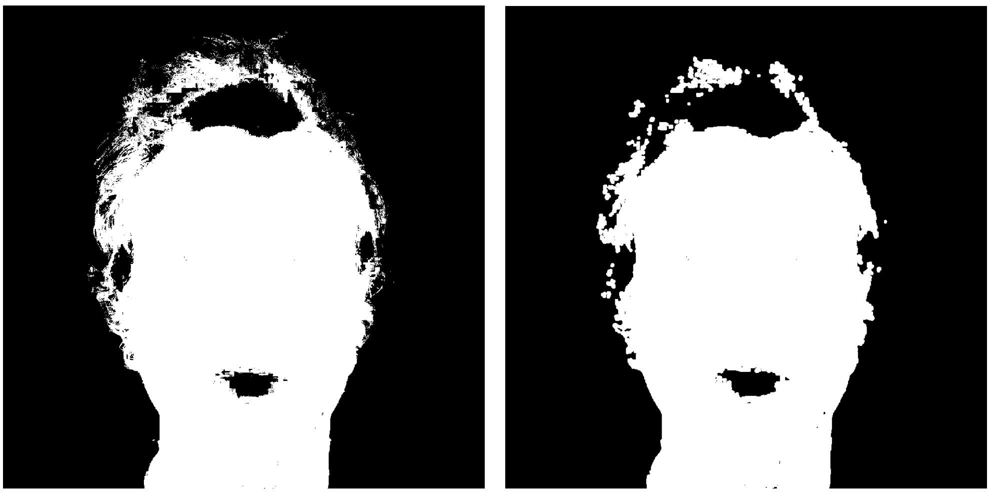
- Total images processed: **85**.
- Total successfully cropped faces: **74**.

These results demonstrate the effectiveness of the implemented preprocessing methods. The key steps involved in preparing the dataset are illustrated in the following figures.

(a) BGR to YCbCr Conversion

(b) BGR to HSV Conversion

Figure 1: Color space conversions applied to the facial images.



(a) Skin Mask obtained by combining BGR and YCbCr images

(b) Applying erosion and dilation to reduce noise

Figure 2: Masking and noise reduction techniques applied to improve face detection.

(a) Cropped Image (b) Applying Grayscale

Figure 3: Final image processing steps, including cropping and grayscale conversion.

These figures highlight the effectiveness of each preprocessing step in preparing the facial images for further tasks.

These results confirm the effectiveness of the implemented methods in preparing facial images for future tasks. The implementation of the Eigenfaces-based face recognition system provided varying results depending on the thresholding approach, the number of eigenfaces, and the quality of the dataset. This section presents quantitative outcomes, key observations, and visual examples from the experiments.

4.5 Quantitative Results

The system was tested on multiple datasets under different configurations. The results are summarized in Table 1.

Table 1: Recognition Accuracy and Unknown Face Detection

Dataset	**Threshold Type**	**Number of Eigen-faces**	**Accuracy (%)**	**Unknown Faces**
Olivetti Faces Dataset	Fixed Threshold	20	68.75	21 out of 80
Olivetti Faces Dataset	Percentile-Based	20	78.75	0 out of 80
Custom Dataset	Fixed Threshold	10	62.50	1 out of 8
Custom Dataset	Percentile-Based	10	50.00	0 out of 8

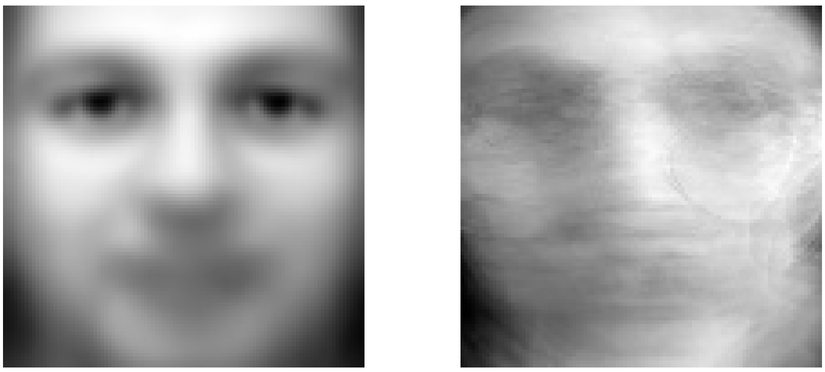
4.6 Key Observations

- **Thresholding Approach:** A fixed threshold enabled the system to classify unknown faces but required careful tuning. A percentile-based threshold provided higher accuracy but failed to classify faces as "unknown."
- **Number of Eigenfaces:** Fewer eigenfaces resulted in lower accuracy due to insufficient facial variation capture. Using too many eigenfaces increased computational complexity and sometimes led to overfitting.
- **Dataset Quality:** Higher-quality datasets improved accuracy, while low-resolution or inconsistent lighting reduced performance.

4.7 Visual Results

The following figures illustrate the performance of the system:

- **Mean Face:** Figure 4a and Figure 4b show the mean faces computed from the Olivetti and our training datasets respectively.
- **Eigenfaces:** Figure 5a and Figure 5b display the grid of eigenfaces used for recognition from both datasets.
- **Recognition Results:**
  - **Successful Recognition:** Figure 6 and Figure 7 demonstrate examples of successfully recognized faces, showing the test face, mean face, closest face, and reconstructed face.
  - **Misclassification:** Figure 8 and Figure 9 present examples of misclassified faces, including the same set of images.

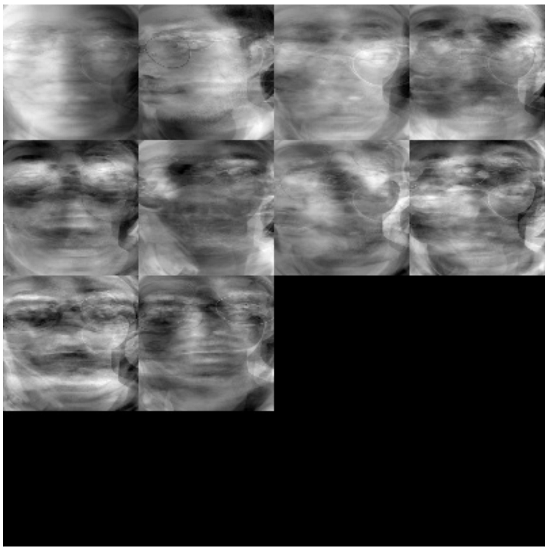


(a) Mean Face Computed from Olivetti Faces Dataset

(b) Mean Face Computed from Our Dataset

Figure 4: Comparison of Mean Faces

(a) Grid of Eigenfaces from Olivetti Faces Dataset



(b) Grid of Eigenfaces from Our Dataset

Figure 5: Comparison of Eigenface Grids

Figure 6: Example of Successful Recognition (Test Face, Mean Face, Closest Face, Reconstructed Face) from Olivetti Faces Dataset

Figure 7: Example of Successful Recognition (Test Face, Mean Face, Closest Face, Reconstructed Face) from Our Dataset

Figure 8: Example of Misclassification (Test Face, Mean Face, Closest Face, Reconstructed Face) from Olivetti Faces Dataset

Figure 9: Example of Misclassification (Test Face, Mean Face, Closest Face, Reconstructed Face) from Our Dataset

5 Discussion

The preprocessing pipeline worked well for preparing facial images. Converting images to grayscale and resizing them ensured consistency, which is important for feature extraction. The skin detection method using HSV and YCrCb color spaces performed well in most conditions, though it could be improved for extreme lighting or varying skin tones.

Edge detection and contour analysis were successful in identifying facial features, but their accuracy depended on the quality of the input images. Low-quality or blurry images made detection less reliable.

When it comes to face recognition, two thresholding methods were tested: a fixed threshold and a percentile-based threshold. The fixed threshold was good at identifying unknown faces but needed careful tuning, making it less flexible. The percentile-based threshold worked better overall but struggled with identifying unknown faces, as it often grouped them into known categories.

The number of eigenfaces used affected performance. Using too few eigenfaces caused underfitting, while too many increased noise and led to overfitting, especially with lower-quality datasets. The quality of the dataset also mattered: high-quality images with consistent lighting and resolution improved accuracy, while poor-quality images hurt performance.

Future improvements could include automating parameter selection and adding advanced methods like deep learning for face detection. Enhancing the quality of datasets could also lead to better performance.

6 Conclusion

This project created a pipeline to preprocess facial images. The methods used:

- Loaded and resized images for consistency.
- Detected skin areas using color space thresholds and improved the masks with morphological operations.
- Applied edge detection and contour analysis to crop faces accurately.

The results show that these steps worked well to prepare the images for further use in machine learning or other tasks. For the eigenfaces-based face recognition, the performance varied based on the thresholding method, number of eigenfaces, and dataset quality:

- A fixed threshold worked well for identifying unknown faces but needed careful tuning.
- A percentile-based threshold gave better overall accuracy but struggled with unknown faces.
- The number of eigenfaces affected performance: too few caused underfitting, and too many led to overfitting, especially with lower-quality datasets.
- Dataset quality played a major role—better quality datasets led to better results.

Future improvements could focus on:

- Improving skin detection in challenging lighting.
- Automating parameter tuning for more flexibility.
- Exploring better preprocessing methods for feature extraction.

In conclusion, the implementation sets a good foundation for facial image processing and offers a starting point for future work and improvements.

References

1. M. Turk and A. Pentland, "Eigenfaces for Recognition," Journal of Cognitive Neuroscience, vol. 3, no. 1, pp. 71-86, 1991. Available: `https://www.mit.edu/~9.54/fall14/Classes/class10/Turk%20Pentland%20Eigenfaces.pdf`
2. "Olivetti Faces Dataset," ATT Laboratories Cambridge, Available: `https://scikit-learn.org/0.19/datasets/olivetti_faces.html`.
3. OpenCV Documentation, "Face Recognition using Eigenfaces," Available: `https://docs.opencv.org/4.x/da/d60/tutorial_face_main.html`.
4. IOP Conference Series: Materials Science and Engineering, "Face detection based on skin color extraction scheme," Available: `https://iopscience.iop.org/article/10.1088/1757-899X/569/3/032006/pdf#:~:text=Face%20detection%20based%20on%20skin%20color%20is%20generally%20divided%20into,to%20detect%20the%20skin%20color`.
5. IEEE Document on Face Detection, Available: `https://ieeexplore.ieee.org/document/7219848`.

6. "Face Recognition using Eigenfaces - GeeksforGeeks," Available: https://www.geeksforgeeks.org/ml-face-recognition-using-eigenfaces-pca-algorithm/.

7. "OpenCV Eigenfaces for Face Recognition - PyImageSearch," Available: https://pyimagesearch.com/2021/05/10/opencv-eigenfaces-for-face-recognition/.

8. H. I. Mansour, "Face Recognition using Eigenfaces (Python)," Available: https://hassan-id-mansour.medium.com/face-recognition-using-eigenfaces-python-b857b2599ed0.

7 Appendices

7.1 Detailed Recognition Logs

7.1.1 Olivetti Faces Dataset Results

The following table presents the detailed recognition results for each test image, including the true label, recognized label, and recognition status.

Test Image	True Label	Recognized Label	Status
Test image: class_1	True label: class_1	Recognized label: class_1	SUCCESSFUL
Test image: class_5	True label: class_5	Recognized label: unknown	UNSUCCESSFUL
Test image: class_13	True label: class_13	Recognized label: class_13	SUCCESSFUL
Test image: class_24	True label: class_24	Recognized label: class_24	SUCCESSFUL
Test image: class_24	True label: class_24	Recognized label: class_24	SUCCESSFUL
Test image: class_0	True label: class_0	Recognized label: unknown	UNSUCCESSFUL
Test image: class_37	True label: class_37	Recognized label: class_37	SUCCESSFUL
Test image: class_30	True label: class_30	Recognized label: class_30	SUCCESSFUL
Test image: class_15	True label: class_15	Recognized label: unknown	UNSUCCESSFUL
Test image: class_34	True label: class_34	Recognized label: class_34	SUCCESSFUL
Test image: class_20	True label: class_20	Recognized label: class_20	SUCCESSFUL
Test image: class_18	True label: class_18	Recognized label: class_18	SUCCESSFUL
Test image: class_28	True label: class_28	Recognized label: class_28	SUCCESSFUL
Test image: class_1	True label: class_1	Recognized label: class_1	SUCCESSFUL
Test image: class_16	True label: class_16	Recognized label: class_16	SUCCESSFUL
Test image: class_35	True label: class_35	Recognized label: class_35	SUCCESSFUL
Test image: class_25	True label: class_25	Recognized label: class_25	SUCCESSFUL
Test image: class_28	True label: class_28	Recognized label: class_20	UNSUCCESSFUL
Test image: class_4	True label: class_4	Recognized label: unknown	UNSUCCESSFUL
Test image: class_38	True label: class_38	Recognized label: class_38	SUCCESSFUL
Test image: class_18	True label: class_18	Recognized label: class_18	SUCCESSFUL
Test image: class_35	True label: class_35	Recognized label: unknown	UNSUCCESSFUL
Test image: class_10	True label: class_10	Recognized label: class_10	SUCCESSFUL
Test image: class_19	True label: class_19	Recognized label: class_19	SUCCESSFUL
Test image: class_23	True label: class_23	Recognized label: class_23	SUCCESSFUL
Test image: class_19	True label: class_19	Recognized label: class_19	SUCCESSFUL
Test image: class_17	True label: class_17	Recognized label: class_17	SUCCESSFUL
Test image: class_33	True label: class_33	Recognized label: unknown	UNSUCCESSFUL
Test image: class_14	True label: class_14	Recognized label: unknown	UNSUCCESSFUL

Test Image	True Label	Recognized Label	Status
Test image: class_24	True label: class_24	Recognized label: unknown	UNSUCCESSFUL
Test image: class_6	True label: class_6	Recognized label: unknown	UNSUCCESSFUL
Test image: class_33	True label: class_33	Recognized label: class_33	SUCCESSFUL
Test image: class_26	True label: class_26	Recognized label: class_26	SUCCESSFUL
Test image: class_16	True label: class_16	Recognized label: class_16	SUCCESSFUL
Test image: class_33	True label: class_33	Recognized label: class_33	SUCCESSFUL
Test image: class_27	True label: class_27	Recognized label: class_27	SUCCESSFUL
Test image: class_8	True label: class_8	Recognized label: class_8	SUCCESSFUL
Test image: class_32	True label: class_32	Recognized label: class_32	SUCCESSFUL
Test image: class_12	True label: class_12	Recognized label: class_12	SUCCESSFUL
Test image: class_20	True label: class_20	Recognized label: unknown	UNSUCCESSFUL
Test image: class_19	True label: class_19	Recognized label: class_19	SUCCESSFUL
Test image: class_30	True label: class_30	Recognized label: class_30	SUCCESSFUL
Test image: class_32	True label: class_32	Recognized label: class_32	SUCCESSFUL
Test image: class_18	True label: class_18	Recognized label: class_18	SUCCESSFUL
Test image: class_16	True label: class_16	Recognized label: class_16	SUCCESSFUL
Test image: class_3	True label: class_3	Recognized label: unknown	UNSUCCESSFUL
Test image: class_24	True label: class_24	Recognized label: unknown	UNSUCCESSFUL
Test image: class_4	True label: class_4	Recognized label: class_4	SUCCESSFUL
Test image: class_33	True label: class_33	Recognized label: unknown	UNSUCCESSFUL
Test image: class_31	True label: class_31	Recognized label: class_31	SUCCESSFUL
Test image: class_4	True label: class_4	Recognized label: unknown	UNSUCCESSFUL
Test image: class_22	True label: class_22	Recognized label: class_8	UNSUCCESSFUL
Test image: class_3	True label: class_3	Recognized label: class_3	SUCCESSFUL
Test image: class_18	True label: class_18	Recognized label: class_18	SUCCESSFUL
Test image: class_35	True label: class_35	Recognized label: class_35	SUCCESSFUL
Test image: class_29	True label: class_29	Recognized label: class_29	SUCCESSFUL
Test image: class_0	True label: class_0	Recognized label: unknown	UNSUCCESSFUL
Test image: class_1	True label: class_1	Recognized label: class_1	SUCCESSFUL
Test image: class_23	True label: class_23	Recognized label: class_23	SUCCESSFUL
Test image: class_25	True label: class_25	Recognized label: unknown	UNSUCCESSFUL
Test image: class_37	True label: class_37	Recognized label: unknown	UNSUCCESSFUL
Test image: class_20	True label: class_20	Recognized label: class_20	SUCCESSFUL
Test image: class_39	True label: class_39	Recognized label: class_39	SUCCESSFUL
Test image: class_11	True label: class_11	Recognized label: unknown	UNSUCCESSFUL
Test image: class_36	True label: class_36	Recognized label: class_36	SUCCESSFUL
Test image: class_10	True label: class_10	Recognized label: class_10	SUCCESSFUL
Test image: class_22	True label: class_22	Recognized label: unknown	UNSUCCESSFUL
Test image: class_20	True label: class_20	Recognized label: class_29	UNSUCCESSFUL
Test image: class_26	True label: class_26	Recognized label: class_26	SUCCESSFUL
Test image: class_21	True label: class_21	Recognized label: class_21	SUCCESSFUL
Test image: class_12	True label: class_12	Recognized label: unknown	UNSUCCESSFUL
Test image: class_26	True label: class_26	Recognized label: class_26	SUCCESSFUL
Test image: class_13	True label: class_13	Recognized label: class_13	SUCCESSFUL
Test image: class_10	True label: class_10	Recognized label: class_10	SUCCESSFUL
Test image: class_10	True label: class_10	Recognized label: class_10	SUCCESSFUL
Test image: class_3	True label: class_3	Recognized label: class_22	UNSUCCESSFUL

Test Image	True Label	Recognized Label	Status
Test image: class_30	True label: class_30	Recognized label: class_30	SUCCESSFUL
Test image: class_6	True label: class_6	Recognized label: unknown	UNSUCCESSFUL
Test image: class_33	True label: class_33	Recognized label: class_33	SUCCESSFUL
Test image: class_19	True label: class_19	Recognized label: class_19	SUCCESSFUL

Recognition accuracy: 68.75%
Unknown recognition: 21 out of 80

7.1.2 Custom Dataset Results

Test Image	True Label	Recognized Label	Status
Test image: yekta	True label: yekta	Recognized label: yekta	SUCCESSFUL
Test image: recep	True label: recep	Recognized label: recep	SUCCESSFUL
Test image: emre	True label: emre	Recognized label: onur	UNSUCCESSFUL
Test image: emre	True label: emre	Recognized label: emre	SUCCESSFUL
Test image: ali	True label: ali	Recognized label: ali	SUCCESSFUL
Test image: yekta	True label: yekta	Recognized label: yekta	SUCCESSFUL
Test image: emre	True label: emre	Recognized label: onur	UNSUCCESSFUL
Test image: yekta	True label: yekta	Recognized label: onur	UNSUCCESSFUL

Recognition accuracy: 62.50%
Unknown recognition: 0 out of 8

7.2 Code

The following is the Python code used for the implementation described in this report:

7.2.1 Preprocessing.py

```
import cv2
import numpy as np
from scipy.ndimage import convolve
import os

def bgr_to_hsv(image):
    hsv_image = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            b, g, r = image[i, j] / 255.0  # Normalize the BGR
              values to [0, 1]
            max_val = max(b, g, r)
            min_val = min(b, g, r)
            delta = max_val - min_val
            if delta == 0:
                h = 0
```

```
            elif max_val == r:
                h = (60 * ((g - b) / delta) + 360) % 360
            elif max_val == g:
                h = (60 * ((b - r) / delta) + 120) % 360
            elif max_val == b:
                h = (60 * ((r - g) / delta) + 240) % 360
            s = 0 if max_val == 0 else (delta / max_val)
            v = max_val
            hsv_image[i, j] = [h, s, v]
    hsv_image[:, :, 0] = hsv_image[:, :, 0] / 2
    hsv_image[:, :, 1:] *= 255
    return hsv_image.astype(np.uint8)

def bgr_to_ycrcb(image):
    ycrcb_image = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            b, g, r = image[i, j]
            y = 0.299 * r + 0.587 * g + 0.114 * b
            cr = (r - y) * 0.713 + 128
            cb = (b - y) * 0.564 + 128
            ycrcb_image[i, j] = [y, cr, cb]
    ycrcb_image = np.clip(ycrcb_image, 0, 255)
    return ycrcb_image.astype(np.uint8)

def in_range(image, lower_bound, upper_bound):
    mask = np.zeros((image.shape[0], image.shape[1]), dtype=np.uint8)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if all(lower_bound <= image[i, j]) and all(image[i, j] <= upper_bound):
                mask[i, j] = 255
            else:
                mask[i, j] = 0
    return mask

def get_structuring_element(shape, ksize):
    if shape != 'ellipse':
        raise ValueError("Only 'ellipse' shape is supported in this implementation")
    rows, cols = ksize
    kernel = np.zeros((rows, cols), dtype=np.uint8)
    center_x, center_y = cols // 2, rows // 2
    axes_x, axes_y = cols / 2, rows / 2
    for i in range(rows):
        for j in range(cols):
            if ((j - center_x) ** 2) / (axes_x ** 2) + ((i - center_y) ** 2) / (axes_y ** 2) <= 1:
                kernel[i, j] = 1
    return kernel
```

```

def bitwise_and(mask1, mask2):
    # both masks have the same shape
    assert mask1.shape == mask2.shape, "Masks must have the same shape"
    result_mask = np.zeros_like(mask1, dtype=np.uint8)
    for i in range(mask1.shape[0]):
        for j in range(mask1.shape[1]):
            # Perform bitwise AND operation
            result_mask[i, j] = mask1[i, j] & mask2[i, j]
    return result_mask

def erode(image, kernel, iterations=1):
    # Get the dimensions of the image and kernel
    img_h, img_w = image.shape
    k_h, k_w = kernel.shape
    pad_h, pad_w = k_h // 2, k_w // 2

    # Pad the image to handle borders
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant', constant_values=255)

    for _ in range(iterations):
        eroded_image = np.copy(image)

        # Iterate over each pixel in the image
        for i in range(img_h):
            for j in range(img_w):
                # Extract the region of interest
                roi = padded_image[i:i + k_h, j:j + k_w]

                # Apply the kernel (structuring element)
                if np.all(roi[kernel == 1] == 255):
                    eroded_image[i, j] = 255
                else:
                    eroded_image[i, j] = 0

        # Update the padded image for the next iteration
        padded_image = np.pad(eroded_image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant', constant_values=255)

    return eroded_image

def dilate(image, kernel, iterations=1):
    # Get the dimensions of the image and kernel
    img_h, img_w = image.shape
    k_h, k_w = kernel.shape
    pad_h, pad_w = k_h // 2, k_w // 2

    # Pad the image to handle borders
```

```
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w))
        , mode='constant', constant_values=0)

    for _ in range(iterations):
        dilated_image = np.copy(image)

        # Iterate over each pixel in the image
        for i in range(img_h):
            for j in range(img_w):
                # Extract the region of interest
                roi = padded_image[i:i + k_h, j:j + k_w]

                # Apply the kernel (structuring element)
                if np.any(roi[kernel == 1] == 255):
                    dilated_image[i, j] = 255
                else:
                    dilated_image[i, j] = 0

        # Update the padded image for the next iteration
        padded_image = np.pad(dilated_image, ((pad_h, pad_h), (
            pad_w, pad_w)), mode='constant', constant_values=0)

    return dilated_image

def gaussian_kernel(size, sigma=1.0):
    kernel = np.fromfunction(
        lambda x, y: (1 / (2 * np.pi * sigma ** 2)) * np.exp(
            -((x - (size - 1) / 2) ** 2 + (y - (size - 1) / 2) **
                2) / (2 * sigma ** 2)
        ),
        (size, size)
    )
    return kernel / np.sum(kernel)

def gaussian_filter(image, sigma=1.0):
    size = int(2 * np.ceil(3 * sigma) + 1)
    kernel = gaussian_kernel(size, sigma)
    return convolve(image, kernel)

def sobel(image, axis):
    """Applies Sobel filter to the image to compute the gradient.
        """
    if axis == 0:
        kernel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    elif axis == 1:
        kernel = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    else:
        raise ValueError("Axis must be 0 (x) or 1 (y)")
    return convolve(image, kernel)

def canny(image, low_threshold, high_threshold):
```

```
    blurred_image = gaussian_filter(image, sigma=1.4)

    # Compute gradient intensity and direction
    grad_x = sobel(blurred_image, axis=0)
    grad_y = sobel(blurred_image, axis=1)
    gradient_magnitude = np.hypot(grad_x, grad_y)
    gradient_direction = np.arctan2(grad_y, grad_x) * (180 / np.pi)
    gradient_direction[gradient_direction < 0] += 180

    # Apply non-maximum suppression
    nms_image = np.zeros_like(gradient_magnitude)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            angle = gradient_direction[i, j]
            q = 255
            r = 255

            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                q = gradient_magnitude[i, j + 1]
                r = gradient_magnitude[i, j - 1]
            elif 22.5 <= angle < 67.5:
                q = gradient_magnitude[i + 1, j - 1]
                r = gradient_magnitude[i - 1, j + 1]
            elif 67.5 <= angle < 112.5:
                q = gradient_magnitude[i + 1, j]
                r = gradient_magnitude[i - 1, j]
            elif 112.5 <= angle < 157.5:
                q = gradient_magnitude[i - 1, j - 1]
                r = gradient_magnitude[i + 1, j + 1]

            if gradient_magnitude[i, j] >= q and gradient_magnitude[i, j] >= r:
                nms_image[i, j] = gradient_magnitude[i, j]
            else:
                nms_image[i, j] = 0

    # Apply double threshold
    strong_edges = (nms_image > high_threshold).astype(np.uint8)
    weak_edges = ((nms_image >= low_threshold) & (nms_image <= high_threshold)).astype(np.uint8)

    # Track edges by hysteresis
    edges = np.zeros_like(image, dtype=np.uint8)
    for i in range(1, image.shape[0] - 1):
        for j in range(1, image.shape[1] - 1):
            if strong_edges[i, j]:
                edges[i, j] = 255
            elif weak_edges[i, j]:
                if (strong_edges[i + 1, j - 1:j + 2].any() or
                    strong_edges[i - 1, j - 1:j + 2].any() or
```

```
                    strong_edges[i, [j - 1, j + 1]].any()):
                    edges[i, j] = 255

    return edges

def find_contours(image):
    contours = []
    visited = np.zeros_like(image, dtype=bool)

    def is_valid(x, y):
        return 0 <= x < image.shape[0] and 0 <= y < image.shape
            [1]

    def trace_contour(start):
        contour = []
        stack = [start]
        directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]

        while stack:
            x, y = stack.pop()
            if visited[x, y]:
                continue
            visited[x, y] = True
            contour.append((x, y))

            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if is_valid(nx, ny) and image[nx, ny] == 255 and
                   not visited[nx, ny]:
                    stack.append((nx, ny))

        return contour

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if image[i, j] == 255 and not visited[i, j]:
                contour = trace_contour((i, j))
                if contour:
                    contours.append(np.array(contour))

    return contours, None

def bounding_rect(contour):
    contour = np.array(contour)  # Convert the contour to a NumPy
        array
    x_min = np.min(contour[:, 1])
    y_min = np.min(contour[:, 0])
    x_max = np.max(contour[:, 1])
    y_max = np.max(contour[:, 0])
    return x_min, y_min, x_max - x_min, y_max - y_min

```

```python
def bgr_to_gray(image):
    # Create an empty array for the grayscale image
    gray_image = np.zeros((image.shape[0], image.shape[1]), dtype
        =np.uint8)

    # Iterate over each pixel
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            b, g, r = image[i, j]
            gray = int(0.299 * r + 0.587 * g + 0.114 * b)
            gray_image[i, j] = gray

    return gray_image

def crop_face_using_edges(image_path, output_path):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError("Image not found or path is incorrect.")

    # Convert the image to HSV and YCbCr color spaces
    hsv_image = bgr_to_hsv(image)
    ycbcr_image = bgr_to_ycrcb(image)

    # Define skin color thresholds
    # HSV thresholds for skin color
    lower_hsv = np.array([0, 30, 50], dtype=np.uint8)
    upper_hsv = np.array([50, 255, 255], dtype=np.uint8)
    mask_hsv = in_range(hsv_image, lower_hsv, upper_hsv)

    # YCbCr thresholds for skin color
    lower_ycbcr = np.array([0, 128, 80], dtype=np.uint8)
    upper_ycbcr = np.array([255, 180, 135], dtype=np.uint8)
    mask_ycbcr = in_range(ycbcr_image, lower_ycbcr, upper_ycbcr)

    # Combine the masks
    skin_mask = bitwise_and(mask_hsv, mask_ycbcr)

    # Morphological operations to remove noise
    kernel = get_structuring_element('ellipse', (5, 5))
    skin_mask = erode(skin_mask, kernel, iterations=2)
    skin_mask = dilate(skin_mask, kernel, iterations=2)

    # Detect edges in the skin mask
    edges = canny(skin_mask, 100, 150)

    # Find contours of the edges
    contours, _ = find_contours(skin_mask)

    max_area = 0
    bounding_box = None

```

```
    for contour in contours:
        x, y, w, h = bounding_rect(contour)
        area = w * h
        if area > max_area:
            max_area = area
            bounding_box = (x, y, w, h)

    if bounding_box:
        x, y, w, h = bounding_box
        # Narrow the cropping region horizontally
        narrow_factor_x = 0.1  # Adjust this value to control horizontal narrowing
        reduction_x = int(w * narrow_factor_x)
        x += reduction_x
        w -= 2 * reduction_x
        # Narrow the cropping region vertically
        narrow_factor_y = 0.1  # Adjust this value to control vertical narrowing
        reduction_y = int(h * narrow_factor_y)
        y += reduction_y
        h -= 2 * reduction_y
        # Ensure the new region is within bounds
        x = max(0, x)
        y = max(0, y)
        w = max(1, w)
        h = max(1, h)
        cropped_region = image[y:y+h, x:x+w]

        gray = bgr_to_gray(cropped_region)

        cv2.imwrite(output_path, gray)
    else:
        print(f"No face detected in {image_path}.")

def process_images_in_folder(input_folder, output_folder):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    for filename in os.listdir(input_folder):
        input_path = os.path.join(input_folder, filename)
        if not (filename.lower().endswith(".png") or filename.lower().endswith(".jpg") or filename.lower().endswith(".jpeg")):
            continue
        print(f"Processing: {input_path}")
        try:
            output_path = os.path.join(output_folder, f"processed_{filename}")
            crop_face_using_edges(input_path, output_path)
        except Exception as e:
            print(f"Error processing {filename}: {e}")
```

```
    print(f"Processing completed. Processed images saved in: {output_folder}")

process_images_in_folder("./faces_dataset_black", "./output_images2")
```

7.2.2 resize.py

```
import os
import cv2
import numpy as np
def resize(src, dsize, interpolation="bilinear"):
    src_height, src_width = src.shape[:2]
    dst_width, dst_height = dsize

    if len(src.shape) == 3:  # Multi-channel image
        channels = src.shape[2]
        dst = np.zeros((dst_height, dst_width, channels), dtype=src.dtype)
    else:  # Grayscale image
        dst = np.zeros((dst_height, dst_width), dtype=src.dtype)

    for i in range(dst_height):
        for j in range(dst_width):
            src_x = j * (src_width / dst_width)
            src_y = i * (src_height / dst_height)
            src_x0 = int(np.floor(src_x))
            src_y0 = int(np.floor(src_y))
            src_x1 = min(src_x0 + 1, src_width - 1)
            src_y1 = min(src_y0 + 1, src_height - 1)

            dx = src_x - src_x0
            dy = src_y - src_y0

            if len(src.shape) == 3:  # Multi-channel
                for c in range(src.shape[2]):
                    dst[i, j, c] = (
                        (1 - dx) * (1 - dy) * src[src_y0, src_x0, c] +
                        dx * (1 - dy) * src[src_y0, src_x1, c] +
                        (1 - dx) * dy * src[src_y1, src_x0, c] +
                        dx * dy * src[src_y1, src_x1, c]
                    )
            else:  # Grayscale
                dst[i, j] = (
                    (1 - dx) * (1 - dy) * src[src_y0, src_x0] +
                    dx * (1 - dy) * src[src_y0, src_x1] +
                    (1 - dx) * dy * src[src_y1, src_x0] +
                    dx * dy * src[src_y1, src_x1]
                )
```

```python

    return dst

def process_images_in_folder(input_folder, output_folder, dsize):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    for filename in os.listdir(input_folder):
        input_path = os.path.join(input_folder, filename)

        if not (filename.lower().endswith(".png") or filename.lower().endswith(".jpg") or filename.lower().endswith(".jpeg")):
            continue

        print(f"Processing: {input_path}")
        try:
            image = cv2.imread(input_path)
            if image is None:
                print(f"Skipping {filename}: Unable to read file.")
                continue

            resized_image = resize(image, dsize)

            output_path = os.path.join(output_folder, filename)
            cv2.imwrite(output_path, resized_image)
        except Exception as e:
            print(f"Error processing {filename}: {e}")

    print(f"Processing completed. Resized images saved in: {output_folder}")

if _name_ == "_main_":
    input_folder = "output_images2"
    output_folder = "resized_images2"
    dsize = (100, 100)
    process_images_in_folder(input_folder, output_folder, dsize)
```

7.2.3 fasttest.py

```python
import os
import numpy as np
import cv2
from tqdm import tqdm
from sklearn.model_selection import train_test_split

def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
    src_height, src_width = src.shape[:2]
    dst_width, dst_height = dsize
```

```
    dst = np.zeros((dst_height, dst_width), dtype=src.dtype)

    for i in range(dst_height):
        for j in range(dst_width):
            src_x = j * (src_width / dst_width)
            src_y = i * (src_height / dst_height)
            src_x0 = int(np.floor(src_x))
            src_y0 = int(np.floor(src_y))
            src_x1 = min(src_x0 + 1, src_width - 1)
            src_y1 = min(src_y0 + 1, src_height - 1)

            dx = src_x - src_x0
            dy = src_y - src_y0

            dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0] + \
                        dx * (1 - dy) * src[src_y0, src_x1] + \
                        (1 - dx) * dy * src[src_y1, src_x0] + \
                        dx * dy * src[src_y1, src_x1]

    return dst
def custom_mean(array, axis=None):
    if axis is None:
        return sum(array) / len(array)
    else:
        return np.sum(array, axis=axis) / array.shape[axis]

def save_Results(base_dir="results"):
    if not os.path.exists(base_dir):
        os.makedirs(base_dir)

    subdirectories = [d for d in os.listdir(base_dir) if os.path.isdir(os.path.join(base_dir, d))]
    run_number = len(subdirectories)
    subdirectory = os.path.join(base_dir, f"results_{run_number}")
    os.makedirs(subdirectory)
    return subdirectory

def save_image_to_file(image, filename):
    cv2.imwrite(filename, image)

def custom_dot(a, b):
    result = []
    for a_row in tqdm(a, desc="Computing dot product (outer loop)"):
        row_result = []
        for b_col in tqdm(zip(*b), desc="Computing dot product (inner loop)", leave=False):
            row_result.append(sum(x * y for x, y in zip(a_row, b_col)))
```

```python
        result.append(row_result)
    return np.array(result)

def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX):
    if dst is None:
        dst = np.zeros_like(src)

    if norm_type == cv2.NORM_MINMAX:
        min_val = np.min(src)
        max_val = np.max(src)
        dst = (src - min_val) * (beta - alpha) / (max_val - min_val) + alpha
    else:
        raise NotImplementedError("Only NORM_MINMAX is implemented")

    return dst

# Add labels to the image
def add_labels_to_image(image, labels, positions, font=cv2.FONT_HERSHEY_SIMPLEX, font_scale=0.5, color=(255, 255, 255), thickness=1):
    for label, position in zip(labels, positions):
        cv2.putText(image, label, position, font, font_scale, color, thickness, cv2.LINE_AA)

# This function loads our dataset of Olivetti faces and resizes them to a target size.
# It returns a list of image paths and their corresponding labels.
def load_olivetti_faces(root_dir, target_size=(100, 100)):
    image_paths = []
    labels = []

    print(f"Loading images from: {root_dir}")
    # Check if the root directory exists
    if not os.path.exists(root_dir):
        print(f"Error: The directory {root_dir} does not exist.")
        return image_paths, labels

    for person_dir in tqdm(os.listdir(root_dir), desc="Processing directories"):
        person_path = os.path.join(root_dir, person_dir)
        if os.path.isdir(person_path):
            for filename in os.listdir(person_path):
                if filename.endswith(('.jpg', '.jpeg', '.png')):
                    full_path = os.path.join(person_path, filename)
                    image = cv2.imread(full_path, cv2.IMREAD_GRAYSCALE)
```

```python
                    if image is None:
                        print(f"Error loading image: {full_path}")
                        continue
                    # resized_image = resize(image, target_size)
                    resized_image = cv2.resize(image, target_size)
                    save_path = os.path.join(person_path, filename)
                    cv2.imwrite(save_path, resized_image)
                    image_paths.append(save_path)
                    labels.append(person_dir)
                    print(f"Loaded and resized: {person_dir} - {filename}")
    print(f"Total images loaded: {len(image_paths)}")
    return image_paths, labels

# Step 2: Preprocess Images (Load, Flatten, Normalize)
def preprocess_images(image_paths):
    dataset = []
    for path in tqdm(image_paths, desc="Preprocessing images"):
        image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if image is None:
            print(f"Error loading image: {path}")
            continue
        # Flatten and normalize the image
        flattened = (image.flatten() / 255.0)
        dataset.append(flattened)
    return np.array(dataset, dtype=np.float64)

# Step 3: Compute Mean Face
def compute_mean_face(dataset):
    # mean_face = custom_mean(dataset, axis=0)
    mean_face = np.mean(dataset, axis=0)

    # Reshape the mean face to its original dimensions (50x50)
    mean_face_image = mean_face.reshape((100, 100))

    # Normalize the mean face image to the range [0, 255]
    # mean_face_image = normalize(mean_face_image, norm_type=cv2.NORM_MINMAX)
    mean_face_image = cv2.normalize(mean_face_image, None, 0, 255, cv2.NORM_MINMAX)

    # Convert to uint8 type
    mean_face_image = mean_face_image.astype(np.uint8)
    cv2.imshow("Mean Face", mean_face_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return mean_face , mean_face_image

```

```
# Step 4: Center Dataset
def center_dataset(dataset, mean_face):
    return dataset - mean_face

# Step 5: Compute Covariance Matrix
def compute_covariance_matrix(centered_data):
    num_images, num_features = centered_data.shape
    # return custom_dot(centered_data.T, centered_data) / num_images
    return np.dot(centered_data, centered_data.T) / num_images

def compute_eigenfaces(centered_data, covariance_matrix, num_eigenfaces):
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvectors = eigenvectors[:, sorted_indices]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, :num_eigenfaces]
    eigenfaces = np.dot(eigenvectors.T, centered_data)
    return eigenfaces, eigenvalues[:num_eigenfaces]

def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape=(100, 100), results_subdirectory=None):
    # Calculate the grid size dynamically
    grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
    grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))

    # Create a blank canvas to display the eigenfaces
    grid_height = grid_rows * image_shape[0]
    grid_width = grid_cols * image_shape[1]
    canvas = np.zeros((grid_height, grid_width), dtype=np.uint8)

    for i in range(num_eigenfaces):
        row = i // grid_cols
        col = i % grid_cols
        eigenface = eigenfaces[i].reshape(image_shape)
        # eigenface = normalize(eigenface, norm_type=cv2.NORM_MINMAX)
        eigenface = cv2.normalize(eigenface, None, 0, 255, cv2.NORM_MINMAX)
        eigenface = eigenface.astype(np.uint8)
        canvas[row * image_shape[0]:(row + 1) * image_shape[0], col * image_shape[1]:(col + 1) * image_shape[1]] = eigenface

    # Display the canvas with all eigenfaces
    cv2.imshow("Eigenfaces", canvas)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # Save the canvas with all eigenfaces
```

```python
    if results_subdirectory:
        eigenfaces_path = os.path.join(results_subdirectory, "eigenfaces.jpg")
        cv2.imwrite(eigenfaces_path, canvas)

# Step 7: Project Faces Onto Eigenfaces
def project_faces(centered_data, eigenfaces):
    # return custom_dot(centered_data, eigenfaces.T)
    return np.dot(centered_data, eigenfaces.T)

def save_results(results, filename):
    with open(filename, 'w') as f:
        for result in results:
            f.write(f"{result}\n")

# Step 8: Recognize Test Face
def recognize_face(test_face, mean_face, eigenfaces, projected_faces, labels, original_faces, fixed_threshold=300):
    centered_test_face = test_face - mean_face
    projected_test_face = np.dot(centered_test_face, eigenfaces.T)
    distances = np.linalg.norm(projected_faces - projected_test_face, axis=1)
    min_distance = np.min(distances)
    recognized_label = labels[np.argmin(distances)]
    closest_face_index = np.argmin(distances)

    # Use a fixed threshold
    threshold = fixed_threshold

    if min_distance > threshold:
        recognized_label = "unknown"
    else:
        recognized_label = labels[closest_face_index]

    # Debugging: Print distances and threshold
    print(f"Threshold: {threshold}")
    print(f"Min distance: {min_distance}")

    # Reconstruct the projected face
    reconstructed_face = np.dot(projected_test_face, eigenfaces) + mean_face

    # Display the test face, mean face, closest face, and reconstructed face side by side
    test_face_image = test_face.reshape((100, 100))
    mean_face_image = mean_face.reshape((100, 100))
    closest_face = original_faces[closest_face_index].reshape((100, 100))
    reconstructed_face_image = reconstructed_face.reshape((100, 100))
```

```

    norm_type=cv2.NORM_MINMAX)
    test_face_image = cv2.normalize(test_face_image, None, 0, 255, cv2.NORM_MINMAX)
    mean_face_image = cv2.normalize(mean_face_image, None, 0, 255, cv2.NORM_MINMAX)
    closest_face = cv2.normalize(closest_face, None, 0, 255, cv2.NORM_MINMAX)
    reconstructed_face_image = cv2.normalize(reconstructed_face_image, None, 0, 255, cv2.NORM_MINMAX)

    # Convert to uint8 type
    test_face_image = test_face_image.astype(np.uint8)
    mean_face_image = mean_face_image.astype(np.uint8)
    closest_face = closest_face.astype(np.uint8)
    reconstructed_face_image = reconstructed_face_image.astype(np.uint8)

    test_face_image = cv2.resize(test_face_image, (200, 200))
    mean_face_image = cv2.resize(mean_face_image, (200, 200))
    closest_face = cv2.resize(closest_face, (200, 200))
    reconstructed_face_image = cv2.resize(reconstructed_face_image, (200, 200))

    # Concatenate images horizontally
    combined_image = np.hstack((test_face_image, mean_face_image, closest_face, reconstructed_face_image))

    # Add labels to the combined image
    labels = ["Test Face", "Mean Face", "Closest Face", "Reconstructed Face"]
    positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
    add_labels_to_image(combined_image, labels, positions)
    # Save the result
    return recognized_label , combined_image

# Paths
olivetti_faces_dir = "real_dataset"  # Directory with Olivetti faces

# Step 1: Load and Resize Olivetti Faces
image_paths, labels = load_olivetti_faces(olivetti_faces_dir)

# Step 2: Preprocess Dataset
dataset = preprocess_images(image_paths)

# Split the dataset into training and test sets
train_paths, test_paths, train_labels, test_labels = train_test_split(image_paths, labels, test_size=0.2)

# Preprocess training and test sets
```

```python
train_dataset = preprocess_images(train_paths)
test_dataset = preprocess_images(test_paths)

# Step 3: Compute Mean Face
mean_face, mean_face_image_to_save = compute_mean_face(
    train_dataset)

# Step 4: Center the Dataset
print("Centering the dataset")
centered_train_data = center_dataset(train_dataset, mean_face)
print("Dataset is centered")
print("covariance matrix is calculating")
# Step 5: Compute Covariance Matrix
covariance_matrix = compute_covariance_matrix(centered_train_data
    )
print("Covariance matrix is calculated ")
# Step 6: Compute Eigenfaces
num_eigenfaces = min(10, train_dataset.shape[0])
eigenfaces, eigenvalues = compute_eigenfaces(centered_train_data,
     covariance_matrix, num_eigenfaces)
print("eigenfaces are calculated ")
# Create a subdirectory for this run's results
results_subdirectory = save_Results()

# Save mean face
mean_face_path = os.path.join(results_subdirectory, "mean_face.
    jpg")
cv2.imwrite(mean_face_path, mean_face_image_to_save)

# Display and save eigenfaces
display_eigenfaces(eigenfaces, num_eigenfaces,
    results_subdirectory=results_subdirectory)

# Step 7: Project Faces Onto Eigenfaces
projected_train_faces = project_faces(centered_train_data,
    eigenfaces)

# Test the recognition on the test set
correct_predictions = 0
unknown = 0
count = 1
results = []
for test_image_path, true_label in tqdm(zip(test_paths,
    test_labels), desc="Recognizing faces", total=len(test_paths))
    :
     test_face_dataset = preprocess_images([test_image_path])
     recognized_label , combined_image = recognize_face(
         test_face_dataset[0],
         mean_face,
         eigenfaces,
         projected_train_faces,
```

```
            train_labels,
            train_dataset,  # Pass the true label to the function
            fixed_threshold=300  # Set a fixed threshold
        )
        if recognized_label == "unknown":
            print(f"Unknown face: {test_image_path}")
            result = f"Test image: {true_label}, True label: {
                true_label}, Recognized label: unknown , UNSUCCESFULL"
            output_path = os.path.join(results_subdirectory, f"{count
                }_{true_label}_recognized_as_unknown_result.jpg")
            unknown += 1
        elif recognized_label == true_label:
            print(f"TRUE")
            result = f"Test image: {true_label}, True label: {
                true_label}, Recognized label: {recognized_label} ,
                SUCCESFULL"
            output_path = os.path.join(results_subdirectory, f"{count
                }{true_label}_recognized_as{true_label}_result.jpg")
            correct_predictions += 1
        else:
            print(f"True label: {true_label}, Recognized as: {
                recognized_label}")
            result = f"Test image: {true_label}, True label: {
                true_label}, Recognized label: {recognized_label} ,
                UNSUCCESFULL"
            output_path = os.path.join(results_subdirectory, f"{count
                }{true_label}_recognized_as{recognized_label}_result.
                jpg")
        count += 1
        # Debugging: Print output_path
        print(f"Saving image to: {output_path}")

        # Save the result
        success = cv2.imwrite(output_path, combined_image)
        if not success:
            print(f"Failed to write image to: {output_path}")

        results.append(result)

# Save all results to a file

# Calculate and print the recognition accuracy
accuracy = correct_predictions / len(test_paths)
print(f"Recognition accuracy: {accuracy * 100:.2f}%")
print(f"Unknown recognition: {unknown} out of {len(test_paths)}")
results.append(f"Recognition accuracy: {accuracy * 100:.2f}%")
results.append(f"Unknown recognition: {unknown} out of {len(
    test_paths)}")
results_file = os.path.join(results_subdirectory, "results.txt")
save_results(results, results_file)
```

7.2.4 customimplrecognition.py

```
import os
import numpy as np
import cv2
from tqdm import tqdm
from sklearn.model_selection import train_test_split

def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
    src_height, src_width = src.shape[:2]
    dst_width, dst_height = dsize
    dst = np.zeros((dst_height, dst_width), dtype=src.dtype)

    for i in range(dst_height):
        for j in range(dst_width):
            src_x = j * (src_width / dst_width)
            src_y = i * (src_height / dst_height)
            src_x0 = int(np.floor(src_x))
            src_y0 = int(np.floor(src_y))
            src_x1 = min(src_x0 + 1, src_width - 1)
            src_y1 = min(src_y0 + 1, src_height - 1)

            dx = src_x - src_x0
            dy = src_y - src_y0

            dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0]
                + \
                        dx * (1 - dy) * src[src_y0, src_x1] + \
                        (1 - dx) * dy * src[src_y1, src_x0] + \
                        dx * dy * src[src_y1, src_x1]

    return dst
def custom_mean(array, axis=None):
    if axis is None:
        return sum(array) / len(array)
    else:
        return np.sum(array, axis=axis) / array.shape[axis]

def create_subdirectories(base_dir="results"):
    if not os.path.exists(base_dir):
        os.makedirs(base_dir)

    subdirectories = [d for d in os.listdir(base_dir) if os.path.
       isdir(os.path.join(base_dir, d))]
    run_number = len(subdirectories)
    subdirectory = os.path.join(base_dir, f"results_{run_number}"
       )
    os.makedirs(subdirectory)
    return subdirectory

def save_image_to_file(image, filename):
```

```python
    cv2.imwrite(filename, image)

def custom_dot(a, b):
    result = []
    for a_row in tqdm(a, desc="Computing dot product (outer loop)"):
        row_result = []
        for b_col in tqdm(zip(*b), desc="Computing dot product (inner loop)", leave=False):
            row_result.append(sum(x * y for x, y in zip(a_row, b_col)))
        result.append(row_result)
    return np.array(result)

def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX):
    if dst is None:
        dst = np.zeros_like(src)

    if norm_type == cv2.NORM_MINMAX:
        min_val = np.min(src)
        max_val = np.max(src)
        dst = (src - min_val) * (beta - alpha) / (max_val - min_val) + alpha
    else:
        raise NotImplementedError("Only NORM_MINMAX is implemented")

    return dst

# Add labels to the image
def add_labels_to_image(image, labels, positions, font=cv2.FONT_HERSHEY_SIMPLEX, font_scale=0.5, color=(255, 255, 255), thickness=1):
    for label, position in zip(labels, positions):
        cv2.putText(image, label, position, font, font_scale, color, thickness, cv2.LINE_AA)

# This function loads our dataset of Olivetti faces and resizes them to a target size.
# It returns a list of image paths and their corresponding labels.
def load_faces(root_dir, target_size=(100,100)):
    image_paths = []
    labels = []

    print(f"Loading images from: {root_dir}")
    # Check if the root directory exists
    if not os.path.exists(root_dir):
        print(f"Error: The directory {root_dir} does not exist.")
        return image_paths, labels
```

```python

    for person_dir in tqdm(os.listdir(root_dir), desc="Processing directories"):
        person_path = os.path.join(root_dir, person_dir)
        if os.path.isdir(person_path):
            for filename in os.listdir(person_path):
                if filename.endswith(('.jpg', '.jpeg', '.png')):
                    full_path = os.path.join(person_path, filename)
                    image = cv2.imread(full_path, cv2.IMREAD_GRAYSCALE)
                    if image is None:
                        print(f"Error loading image: {full_path}")
                        continue
                    resized_image = resize(image, target_size)
                    save_path = os.path.join(person_path, filename)
                    cv2.imwrite(save_path, resized_image)
                    image_paths.append(save_path)
                    labels.append(person_dir)
                    print(f"Loaded and resized: {person_dir} - {filename}")
    print(f"Total images loaded: {len(image_paths)}")
    return image_paths, labels

# Step 2: Preprocess Images (Load, Flatten, Normalize)
def preprocess_images(image_paths):
    dataset = []
    for path in tqdm(image_paths, desc="Preprocessing images"):
        image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if image is None:
            print(f"Error loading image: {path}")
            continue
        # Flatten and normalize the image
        flattened = (image.flatten() / 255.0)
        dataset.append(flattened)
    return np.array(dataset, dtype=np.float64)

# Step 3: Compute Mean Face
def compute_mean_face(dataset):
    mean_face = custom_mean(dataset, axis=0)

    # Reshape the mean face to its original dimensions (50x50)
    mean_face_image = mean_face.reshape((100,100))

    # Normalize the mean face image to the range [0, 255]
    mean_face_image = normalize(mean_face_image, norm_type=cv2.NORM_MINMAX)

    # Convert to uint8 type
```

```
    mean_face_image = mean_face_image.astype(np.uint8)

    return mean_face, mean_face_image

# Step 4: Center Dataset
def center_dataset(dataset, mean_face):
    return dataset - mean_face

# Step 5: Compute Covariance Matrix
def compute_covariance_matrix(centered_data):
    num_images, num_features = centered_data.shape
    return custom_dot(centered_data.T, centered_data) /
       num_images

# Step 6: Perform Eigen Decomposition
def compute_eigenfaces(centered_data, covariance_matrix,
   num_eigenfaces):
    print("Computing eigenfaces")
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvectors = eigenvectors[:, sorted_indices]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, :num_eigenfaces]
    eigenfaces = custom_dot(eigenvectors.T, centered_data)
    return eigenfaces, eigenvalues[:num_eigenfaces]

def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape
   =(100,100), results_subdirectory=None):
    # Calculate the grid size dynamically
    grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
    grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))

    # Create a blank canvas to display the eigenfaces
    grid_height = grid_rows * image_shape[0]
    grid_width = grid_cols * image_shape[1]
    canvas = np.zeros((grid_height, grid_width), dtype=np.uint8)

    for i in range(num_eigenfaces):
        row = i // grid_cols
        col = i % grid_cols
        eigenface = eigenfaces[i].reshape(image_shape)
        eigenface = normalize(eigenface, norm_type=cv2.
           NORM_MINMAX)
        eigenface = eigenface.astype(np.uint8)
        canvas[row * image_shape[0]:(row + 1) * image_shape[0],
           col * image_shape[1]:(col + 1) * image_shape[1]] =
           eigenface

    # Display the canvas with all eigenfaces
    cv2.imshow("Eigenfaces", canvas)
    cv2.waitKey(0)
```

```python
    cv2.destroyAllWindows()

    # Save the canvas with all eigenfaces
    if results_subdirectory:
        eigenfaces_path = os.path.join(results_subdirectory, "
            eigenfaces.jpg")
        cv2.imwrite(eigenfaces_path, canvas)

# Step 7: Project Faces Onto Eigenfaces
def project_faces(centered_data, eigenfaces):
    return custom_dot(centered_data, eigenfaces.T)

def save_results(results, filename):
    with open(filename, 'w') as f:
        for result in results:
            f.write(f"{result}\n")

# Step 8: Recognize Test Face
def recognize_face(test_face, mean_face, eigenfaces,
   projected_faces, labels, original_faces, fixed_threshold=300):
    centered_test_face = test_face - mean_face
    centered_test_face = centered_test_face.reshape(1, -1)
    projected_test_face = custom_dot(centered_test_face,
        eigenfaces.T)
    distances = np.linalg.norm(projected_faces -
        projected_test_face, axis=1)
    min_distance = np.min(distances)
    recognized_label = labels[np.argmin(distances)]
    closest_face_index = np.argmin(distances)
    threshold = np.percentile(distances, 10)
    # Use a fixed threshold
    # threshold = fixed_threshold

    # Debugging: Print distances and threshold
    print(f"Threshold: {threshold}")
    print(f"Min distance: {min_distance}")

    if min_distance > threshold:
        recognized_label = "unknown"
    else:
        recognized_label = labels[closest_face_index]
    # Reconstruct the projected face
    # reconstructed_face = np.dot(projected_test_face, eigenfaces
        ) + mean_face
    reconstructed_face = custom_dot(projected_test_face,
        eigenfaces) + mean_face

    # Display the test face, mean face, closest face, and
        reconstructed face side by side
    test_face_image = test_face.reshape((100,100))
    mean_face_image = mean_face.reshape((100,100))
```

```
    closest_face = original_faces[closest_face_index].reshape
       ((100,100))
    reconstructed_face_image = reconstructed_face.reshape
       ((100,100))

    # Normalize the images to the range [0, 255]
    test_face_image = normalize(test_face_image, norm_type=cv2.
       NORM_MINMAX)
    mean_face_image = normalize(mean_face_image, norm_type=cv2.
       NORM_MINMAX)
    closest_face = normalize(closest_face, norm_type=cv2.
       NORM_MINMAX)
    reconstructed_face_image = normalize(reconstructed_face_image
       , norm_type=cv2.NORM_MINMAX)

    # Convert to uint8 type
    test_face_image = test_face_image.astype(np.uint8)
    mean_face_image = mean_face_image.astype(np.uint8)
    closest_face = closest_face.astype(np.uint8)
    reconstructed_face_image = reconstructed_face_image.astype(np
       .uint8)

    # Resize images to be larger for better visibility
    test_face_image = resize(test_face_image, (200, 200))
    mean_face_image = resize(mean_face_image, (200, 200))
    closest_face = resize(closest_face, (200, 200))
    reconstructed_face_image = resize(reconstructed_face_image,
       (200, 200))

    # Concatenate images horizontally
    combined_image = np.hstack((test_face_image, mean_face_image,
        closest_face, reconstructed_face_image))

    # Add labels to the combined image
    labels = ["Test Face", "Mean Face", "Closest Face", "
       Reconstructed Face"]
    positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
    add_labels_to_image(combined_image, labels, positions)

    # Save the result
    return recognized_label , combined_image

# Paths
# olivetti_faces_dir = "olivetti_faces"  # Directory with
   Olivetti faces
olivetti_faces_dir = "cropped_faces"

# Step 1: Load and Resize Olivetti Faces
image_paths, labels = load_faces(olivetti_faces_dir)

# Step 2: Preprocess Dataset
```

```
dataset = preprocess_images(image_paths)

# Split the dataset into training and test sets
train_paths, test_paths, train_labels, test_labels =
    train_test_split(image_paths, labels, test_size=0.2)

# Preprocess training and test sets
train_dataset = preprocess_images(train_paths)
test_dataset = preprocess_images(test_paths)

# Step 3: Compute Mean Face
mean_face, mean_face_image = compute_mean_face(train_dataset)

# Step 4: Center the Dataset
centered_train_data = center_dataset(train_dataset, mean_face)

# Step 5: Compute Covariance Matrix
covariance_matrix = compute_covariance_matrix(centered_train_data
    )
print("Covariance matrix is calculated ")
# Step 6: Compute Eigenfaces
num_eigenfaces = min(10, train_dataset.shape[0])
eigenfaces, eigenvalues = compute_eigenfaces(centered_train_data,
     covariance_matrix, num_eigenfaces)

# Create a subdirectory for this run's results
results_subdirectory = create_subdirectories()

# Save mean face
mean_face_path = os.path.join(results_subdirectory, "mean_face.
    jpg")
cv2.imwrite(mean_face_path, mean_face_image)

# Display and save eigenfaces
display_eigenfaces(eigenfaces, num_eigenfaces,
    results_subdirectory=results_subdirectory)

# Step 7: Project Faces Onto Eigenfaces
projected_train_faces = project_faces(centered_train_data,
    eigenfaces)

# Test the recognition on the test set
correct_predictions = 0
unknown = 0
count = 1
results = []
for test_image_path, true_label in tqdm(zip(test_paths,
    test_labels), desc="Recognizing faces", total=len(test_paths))
    :
     test_face_dataset = preprocess_images([test_image_path])
     recognized_label , combined_image= recognize_face(
```

```python
        test_face_dataset[0],
        mean_face,
        eigenfaces,
        projected_train_faces,
        train_labels,
        train_dataset,
        fixed_threshold=300  # Set a fixed threshold
    )
    if recognized_label == "unknown":
        print(f"Unknown face: {test_image_path}")
        result = f"Test image: {true_label}, True label: {true_label}, Recognized label: unknown, UNSUCCESFULL"
        output_path = os.path.join(results_subdirectory, f"{count}_{true_label}_recognized_as_unknown_result.jpg")
        unknown += 1
    elif recognized_label == true_label:
        print(f"TRUE")
        result = f"Test image: {true_label}, True label: {true_label}, Recognized label: {recognized_label}, SUCCESFULL"
        output_path = os.path.join(results_subdirectory, f"{count}{true_label}_recognized_as{true_label}_result.jpg")
        correct_predictions += 1
    else:
        print(f"True label: {true_label}, Recognized as: {recognized_label}")
        result = f"Test image: {true_label}, True label: {true_label}, Recognized label: {recognized_label}, UNSUCCESFULL"
        output_path = os.path.join(results_subdirectory, f"{count}{true_label}_recognized_as{recognized_label}_result.jpg")
    count += 1
    # Save the result
    cv2.imwrite(output_path, combined_image)
    results.append(result)

# Save all results to a file

# Calculate and print the recognition accuracy
accuracy = correct_predictions / len(test_paths)
print(f"Recognition accuracy: {accuracy * 100:.2f}%")
print(f"Unknown recognition: {unknown} out of {len(test_paths)}")
results.append(f"Recognition accuracy: {accuracy * 100:.2f}%")
results.append(f"Unknown recognition: {unknown} out of {len(test_paths)}")
results_file = os.path.join(results_subdirectory, "results.txt")
save_results(results, results_file)
```