

# Face Recognition Using Eigenfaces

## A Principal Component Analysis (PCA) Approach

**Authors:**

Kerem Göbekcioğlu and Kamil Duru

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Methodology</b>	<b>3</b>
2.1	Image Preprocessing Steps . . . . .	3
2.1.1	Color Space Conversion . . . . .	3
2.1.2	Skin Mask Creation . . . . .	3
2.1.3	Morphological Operations . . . . .	4
2.1.4	Edge and Contour Processing . . . . .	4
2.1.5	Region Processing . . . . .	4
2.2	Face Recognition . . . . .	4
2.2.1	Principal Component Analysis (PCA) . . . . .	4
2.2.2	Eigenfaces Method . . . . .	5
2.2.3	Mathematical Foundation . . . . .	6
<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Face Detection and Preprocessing . . . . .	7
3.2	Image Conversion Functions . . . . .	7
3.2.1	<code>bgr_to_hsv</code> . . . . .	7
3.2.2	<code>bgr_to_ycrcb</code> . . . . .	8
3.3	Edge Detection . . . . .	9
3.3.1	<code>canny</code> . . . . .	9
3.4	Morphological Operations . . . . .	10
3.4.1	<code>erode</code> and <code>dilate</code> . . . . .	10
3.5	Image Preprocessing and Dataset Preparation . . . . .	11
3.5.1	Loading and Resizing Images . . . . .	12
3.5.2	Preprocessing Images . . . . .	12
3.6	Custom Functions . . . . .	13
3.6.1	Computing the Dot Product . . . . .	13
3.6.2	Normalizing the Image . . . . .	13
3.6.3	Resizing the Image . . . . .	14
3.6.4	Computing the Mean . . . . .	15
3.7	Eigenfaces Computation . . . . .	15
3.7.1	Computing the Mean Face . . . . .	16
3.7.2	Centering the Dataset . . . . .	16
3.7.3	Computing the Covariance Matrix . . . . .	16
3.7.4	Eigenvalue Decomposition and Eigenfaces Computation . . . . .	17
3.7.5	Displaying Eigenfaces . . . . .	17
3.8	Face Recognition . . . . .	18
3.8.1	Projecting Faces onto Eigenfaces . . . . .	18
3.8.2	Recognizing Test Faces . . . . .	18
3.9	Testing and Evaluation . . . . .	20
<b>4</b>	<b>Results</b>	<b>21</b>
4.1	Image Preprocessing and Face Cropping . . . . .	21
4.2	Skin Detection . . . . .	21
4.3	Edge Detection and Contour Analysis . . . . .	21
4.4	Overall Processing Results . . . . .	22

4.5	Quantitative Results . . . . .	23
4.6	Key Observations . . . . .	24
4.7	Visual Results . . . . .	24
<b>5</b>	<b>Discussion</b>	<b>26</b>
<b>6</b>	<b>Conclusion</b>	<b>26</b>
<b>7</b>	<b>Appendices</b>	<b>28</b>
7.1	Detailed Recognition Logs . . . . .	28
7.1.1	Olivetti Faces Dataset Results . . . . .	28
7.1.2	Custom Dataset Results . . . . .	30
7.2	Code . . . . .	30
7.2.1	Preprocessing.py . . . . .	30
7.2.2	resize.py . . . . .	38
7.2.3	fasttest.py . . . . .	39
7.2.4	customimplrecognition.py . . . . .	48

# 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 ( $100 \times 100$  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:

$$\mu = \frac{1}{Q} \sum_{i=1}^Q D_i \quad (1)$$

*Equation 1: Mean Face Vector*

where  $D_i$  is the  $i$ -th face vector. This vector  $\mu$  represents the average image.

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

$$X = D - \mu \quad (2)$$

*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:

$$C = \frac{1}{Q} X X^T \quad (3)$$

*Equation 3: Covariance Matrix*

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

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

$$C v_i = \lambda_i v_i \quad (4)$$

*Equation 4: Eigenvalue Decomposition*

where  $v_i$  represents the eigenvector corresponding to the eigenvalue  $\lambda_i$ . 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  $\mu$  as described in PCA.

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

$$C = \frac{1}{Q} X X^T \quad (5)$$

*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:

$$C v_i = \lambda_i v_i \quad (6)$$

*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:

$$w_i = E^T X_i \quad (7)$$

*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:

$$\text{Distance} = \|w_{\text{test}} - w_i\| \quad (8)$$

*Equation 8: Euclidean Distance for Face Recognition*

where  $w_{\text{test}}$  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:

$$C = \frac{1}{Q}XX^T \quad (3)$$

*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:

$$w_i = E^T X_i \quad (7)$$

*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.

```

1 def bgr_to_hsv(image):
2     """
3     Convert an image from BGR to HSV color space.
4
5     Args:
6         image (numpy.ndarray): Input BGR image.
7
8     Returns:
```



```

9         numpy.ndarray: HSV image.
10     """
11     hsv_image = np.zeros_like(image, dtype=np.float32)
12     for i in range(image.shape[0]):
13         for j in range(image.shape[1]):
14             b, g, r = image[i, j] / 255.0
15             max_val = max(b, g, r)
16             min_val = min(b, g, r)
17             delta = max_val - min_val
18
19             if delta == 0:
20                 h = 0
21             elif max_val == r:
22                 h = (60 * ((g - b) / delta) + 360) % 360
23             elif max_val == g:
24                 h = (60 * ((b - r) / delta) + 120) % 360
25             elif max_val == b:
26                 h = (60 * ((r - g) / delta) + 240) % 360
27
28             s = 0 if max_val == 0 else (delta / max_val)
29             v = max_val
30
31             hsv_image[i, j] = [h, s, v]
32
33     hsv_image[:, :, 0] = hsv_image[:, :, 0] / 2
34     hsv_image[:, :, 1:] *= 255
35     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\_ycrb

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

```

1 def bgr_to_ycrcb(image):
2     """
3     Convert an image from BGR to YCrCb color space.
4
5     Args:
6         image (numpy.ndarray): Input BGR image.
7
8     Returns:
9         numpy.ndarray: YCrCb image.
10    """
11    ycrb_image = np.zeros_like(image, dtype=np.float32)
12    for i in range(image.shape[0]):
13        for j in range(image.shape[1]):
14            b, g, r = image[i, j]
15            y = 0.299 * r + 0.587 * g + 0.114 * b
16            cr = (r - y) * 0.713 + 128

```

```

17         cb = (b - y) * 0.564 + 128
18         ycrcb_image[i, j] = [y, cr, cb]
19     ycrcb_image = np.clip(ycrcb_image, 0, 255)
20     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.

```

1 def canny(image, low_threshold, high_threshold):
2     """
3     Apply Canny edge detection to an image.
4
5     Args:
6         image (numpy.ndarray): Input image.
7         low_threshold (int): Lower threshold for hysteresis.
8         high_threshold (int): Upper threshold for hysteresis.
9
10    Returns:
11        numpy.ndarray: Image with edges detected.
12    """
13    blurred_image = gaussian_filter(image, sigma=1.4)
14    grad_x = sobel(blurred_image, axis=0)
15    grad_y = sobel(blurred_image, axis=1)
16    gradient_magnitude = np.hypot(grad_x, grad_y)
17    gradient_direction = np.arctan2(grad_y, grad_x) * (180 / np.
18        pi)
19    gradient_direction[gradient_direction < 0] += 180
20
21    nms_image = np.zeros_like(gradient_magnitude)
22    for i in range(1, image.shape[0] - 1):
23        for j in range(1, image.shape[1] - 1):
24            angle = gradient_direction[i, j]
25            q = r = 255
26
27            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
28                q = gradient_magnitude[i, j + 1]
29                r = gradient_magnitude[i, j - 1]
30            elif 22.5 <= angle < 67.5:
31                q = gradient_magnitude[i + 1, j - 1]
32                r = gradient_magnitude[i - 1, j + 1]
33            elif 67.5 <= angle < 112.5:
34                q = gradient_magnitude[i + 1, j]
35                r = gradient_magnitude[i - 1, j]
36            elif 112.5 <= angle < 157.5:

```

```

36         q = gradient_magnitude[i - 1, j - 1]
37         r = gradient_magnitude[i + 1, j + 1]
38
39         if gradient_magnitude[i, j] >= q and
40            gradient_magnitude[i, j] >= r:
41             nms_image[i, j] = gradient_magnitude[i, j]
42
43 strong_edges = (nms_image > high_threshold).astype(np.uint8)
44 weak_edges = ((nms_image >= low_threshold) & (nms_image <=
45             high_threshold)).astype(np.uint8)
46
47 edges = np.zeros_like(image, dtype=np.uint8)
48 for i in range(1, image.shape[0] - 1):
49     for j in range(1, image.shape[1] - 1):
50         if strong_edges[i, j]:
51             edges[i, j] = 255
52         elif weak_edges[i, j]:
53             if (strong_edges[i + 1, j - 1:j + 2].any() or
54                 strong_edges[i - 1, j - 1:j + 2].any() or
55                 strong_edges[i, [j - 1, j + 1]].any()):
56                 edges[i, j] = 255
57
58 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.

```

1 def erode(image, kernel, iterations=1):
2     """
3     Apply erosion to an image.
4
5     Args:
6         image (numpy.ndarray): Input image.
7         kernel (numpy.ndarray): Structuring element.
8         iterations (int): Number of iterations.
9
10    Returns:
11        numpy.ndarray: Eroded image.
12    """
13    img_h, img_w = image.shape
14    k_h, k_w = kernel.shape
15    pad_h, pad_w = k_h // 2, k_w // 2
16    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)),
17                           , mode='constant', constant_values=255)

```

```

17
18     for _ in range(iterations):
19         eroded_image = np.copy(image)
20         for i in range(img_h):
21             for j in range(img_w):
22                 roi = padded_image[i:i + k_h, j:j + k_w]
23                 if np.all(roi[kernel == 1] == 255):
24                     eroded_image[i, j] = 255
25                 else:
26                     eroded_image[i, j] = 0
27         padded_image = np.pad(eroded_image, ((pad_h, pad_h), (
28             pad_w, pad_w)), mode='constant', constant_values=255)
29
30
31 def dilate(image, kernel, iterations=1):
32     """
33     Apply dilation to an image.
34
35     Args:
36         image (numpy.ndarray): Input image.
37         kernel (numpy.ndarray): Structuring element.
38         iterations (int): Number of iterations.
39
40     Returns:
41         numpy.ndarray: Dilated image.
42     """
43     img_h, img_w = image.shape
44     k_h, k_w = kernel.shape
45     pad_h, pad_w = k_h // 2, k_w // 2
46     padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)),
47                             , mode='constant', constant_values=0)
48
49     for _ in range(iterations):
50         dilated_image = np.copy(image)
51         for i in range(img_h):
52             for j in range(img_w):
53                 roi = padded_image[i:i + k_h, j:j + k_w]
54                 if np.any(roi[kernel == 1] == 255):
55                     dilated_image[i, j] = 255
56         padded_image = np.pad(dilated_image, ((pad_h, pad_h), (
57             pad_w, pad_w)), mode='constant', constant_values=0)
58     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.

```
1 def load_faces(root_dir, target_size=(100, 100)):  
2     """  
3     Loads and resizes images from the dataset.  
4  
5     Parameters:  
6         root_dir (str): Path to the dataset root directory.  
7         target_size (tuple): Target size for resizing the images  
8             (width, height).  
9  
10    Returns:  
11        image_paths (list): List of paths to the loaded images.  
12        labels (list): List of labels corresponding to the images  
13  
14    """  
15    image_paths = []  
16    labels = []  
17    for person_dir in tqdm(os.listdir(root_dir), desc="Processing  
18    directories"):  
19        person_path = os.path.join(root_dir, person_dir)  
20        if os.path.isdir(person_path):  
21            for filename in os.listdir(person_path):  
22                if filename.endswith(('.jpg', '.jpeg', '.png')):  
23                    full_path = os.path.join(person_path,  
24                    filename)  
25                    image = cv2.imread(full_path, cv2.  
26                    IMREAD_GRAYSCALE)  
27                    resized_image = resize(image, target_size)  
28                    image_paths.append(full_path)  
29                    labels.append(person_dir)  
30    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.

```
1 def preprocess_images(image_paths):  
2     """  
3     Preprocesses images by flattening and normalizing them.
```

```

4
5     Parameters:
6         image_paths (list): List of paths to the images.
7
8     Returns:
9         dataset (numpy.ndarray): Flattened and normalized images.
10    """
11    dataset = []
12    for path in tqdm(image_paths, desc="Preprocessing images"):
13        image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
14        flattened = (image.flatten() / 255.0)
15        dataset.append(flattened)
16    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.

```

1 def custom_dot(a, b):
2     """
3     Computes the dot product between two matrices a and b.
4
5     Parameters:
6         a (numpy.ndarray): First matrix, where each row is a data
7                             vector.
8         b (numpy.ndarray): Second matrix, where each column is a
9                             data vector.
10
11     Returns:
12         numpy.ndarray: Result of the dot product.
13    """
14    result = []
15    for a_row in tqdm(a, desc="Computing dot product (outer loop)"):
16        row_result = []
17        for b_col in tqdm(zip(*b), desc="Computing dot product (inner loop)", leave=False):
18            row_result.append(sum(x * y for x, y in zip(a_row, b_col)))
19        result.append(row_result)
20    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.

```

1 def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.
  NORM_MINMAX):
2     """
3     Normalizes the input image.
4
5     Parameters:
6         src (numpy.ndarray): Input image to be normalized.
7         dst (numpy.ndarray, optional): Output image. If None, a
          new array is created.
8         alpha (int): Minimum intensity value for normalization.
9         beta (int): Maximum intensity value for normalization.
10        norm_type (int): Type of normalization (default is cv2.
          NORM_MINMAX).
11
12    Returns:
13        numpy.ndarray: Normalized image.
14    """
15    if dst is None:
16        dst = np.zeros_like(src)
17
18    if norm_type == cv2.NORM_MINMAX:
19        min_val = np.min(src)
20        max_val = np.max(src)
21        dst = (src - min_val) * (beta - alpha) / (max_val -
          min_val) + alpha
22    else:
23        raise NotImplementedError("Only NORM_MINMAX is
          implemented")
24
25    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.

```

1 def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
2     """
3     Resizes the input image to the specified dimensions.
4
5     Parameters:
6         src (numpy.ndarray): Input image to be resized.
7         dsize (tuple): New dimensions (width, height) for
          resizing.
8         interpolation (int): Interpolation method for resizing (
          default is cv2.INTER_LINEAR).
9
10    Returns:
11        numpy.ndarray: Resized image.
12    """
13    src_height, src_width = src.shape[:2]

```

```

14     dst_width, dst_height = dsize
15     dst = np.zeros((dst_height, dst_width), dtype=src.dtype)
16
17     for i in range(dst_height):
18         for j in range(dst_width):
19             src_x = j * (src_width / dst_width)
20             src_y = i * (src_height / dst_height)
21             src_x0 = int(np.floor(src_x))
22             src_y0 = int(np.floor(src_y))
23             src_x1 = min(src_x0 + 1, src_width - 1)
24             src_y1 = min(src_y0 + 1, src_height - 1)
25
26             dx = src_x - src_x0
27             dy = src_y - src_y0
28
29             dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0]
30                 + \
31                     dx * (1 - dy) * src[src_y0, src_x1] + \
32                     (1 - dx) * dy * src[src_y1, src_x0] + \
33                     dx * dy * src[src_y1, src_x1]
34
35     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.

```

1 def custom_mean(array, axis=None):
2     """
3     Computes the mean of a dataset.
4
5     Parameters:
6         array (numpy.ndarray): The input dataset.
7         axis (int, optional): Axis along which the mean is
8                               computed (default is None for full array).
9
10    Returns:
11        float or numpy.ndarray: Mean of the dataset.
12    """
13    if axis is None:
14        return sum(array) / len(array)
15    else:
16        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.

```
1 def compute_mean_face(dataset):
2     """
3     Computes the mean face from the dataset.
4
5     Parameters:
6         dataset (numpy.ndarray): Matrix of flattened images.
7
8     Returns:
9         mean_face (numpy.ndarray): Mean face vector.
10        mean_face_image (numpy.ndarray): Reshaped and normalized
11        mean face image.
12    """
13    mean_face = custom_mean(dataset, axis=0)
14    mean_face_image = mean_face.reshape((100, 100))
15    mean_face_image = normalize(mean_face_image, norm_type=cv2.
16                                NORM_MINMAX)
```

### 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.

```
1 def center_dataset(dataset, mean_face):
2     """
3     Centers the dataset by subtracting the mean face.
4
5     Parameters:
6         dataset (numpy.ndarray): Matrix of flattened images.
7         mean_face (numpy.ndarray): Mean face vector.
8
9     Returns:
10        centered_data (numpy.ndarray): Centered dataset.
11    """
12    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.

```
1 def compute_covariance_matrix(centered_data):
2     """
3     Computes the covariance matrix for the centered dataset.
4
5     Parameters:
6         centered_data (numpy.ndarray): Centered dataset.
```

```

7
8     Returns:
9         covariance_matrix (numpy.ndarray): Covariance matrix.
10    """
11    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.

```

1 def compute_eigenfaces(centered_data, covariance_matrix,
2   num_eigenfaces):
3     """
4     Computes the eigenfaces from the covariance matrix.
5
6     Parameters:
7         centered_data (numpy.ndarray): Centered dataset.
8         covariance_matrix (numpy.ndarray): Covariance matrix.
9         num_eigenfaces (int): Number of eigenfaces to compute.
10
11     Returns:
12         eigenfaces (numpy.ndarray): Computed eigenfaces.
13         eigenvalues (numpy.ndarray): Corresponding eigenvalues.
14     """
15     eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
16     sorted_indices = np.argsort(eigenvalues)[::-1]
17     eigenvectors = eigenvectors[:, sorted_indices]
18     eigenvalues = eigenvalues[sorted_indices]
19     eigenfaces = custom_dot(eigenvectors.T, centered_data)
20     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.

```

1 def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape
2   =(100, 100), results_subdirectory=None):
3     """
4     Displays and optionally saves the computed eigenfaces.
5
6     Parameters:
7         eigenfaces (numpy.ndarray): Computed eigenfaces.
8         num_eigenfaces (int): Number of eigenfaces to display.
9         image_shape (tuple): Shape of each eigenface image.
10        results_subdirectory (str): Directory to save the
11        visualization.
12    """

```

```

11 grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
12 grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))
13 canvas = np.zeros((grid_rows * image_shape[0], grid_cols *
14                    image_shape[1]), dtype=np.uint8)
15 for i in range(num_eigenfaces):
16     row = i // grid_cols
17     col = i % grid_cols
18     eigenface = eigenfaces[i].reshape(image_shape)
19     eigenface = normalize(eigenface, norm_type=cv2.
20                           NORM_MINMAX)
21     canvas[row * image_shape[0]:(row + 1) * image_shape[0],
22           col * image_shape[1]:(col + 1) * image_shape[1]] =
23         eigenface
24 cv2.imshow("Eigenfaces", canvas)
25 cv2.waitKey(0)
26 if results_subdirectory:
27     eigenfaces_path = os.path.join(results_subdirectory, "
28                                     eigenfaces.jpg")
29     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.

```

1 def project_faces(centered_data, eigenfaces):
2     """
3     Projects faces onto the eigenface space.
4
5     Parameters:
6         centered_data (numpy.ndarray): Centered dataset.
7         eigenfaces (numpy.ndarray): Computed eigenfaces.
8
9     Returns:
10        projections (numpy.ndarray): Projected faces.
11    """
12    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."

```

1 def recognize_face(test_face, mean_face, eigenfaces,
2                   projected_faces, labels, original_faces, fixed_threshold=300):

```

```

3 Recognizes a test face by comparing its projection with
  training projections.
4
5 Parameters:
6     test_face (numpy.ndarray): Flattened test face vector.
7     mean_face (numpy.ndarray): Mean face vector.
8     eigenfaces (numpy.ndarray): Computed eigenfaces.
9     projected_faces (numpy.ndarray): Projections of training
   faces.
10    labels (list): Labels of training faces.
11    fixed_threshold (float): Threshold for classifying "
   unknown" faces.
12
13 Returns:
14     tuple: A tuple containing:
15         - recognized_label (str): Label of the recognized
16           face or "unknown."
17         - combined_image (numpy.ndarray): The image
18           displaying the test face, mean face, closest face,
19           and reconstructed face side by side.
20
21 """
22 centered_test_face = test_face - mean_face
23 centered_test_face = centered_test_face.reshape(1, -1)
24 projected_test_face = custom_dot(centered_test_face,
   eigenfaces.T)
25 distances = np.linalg.norm(projected_faces -
   projected_test_face, axis=1)
26 min_distance = np.min(distances)
27 recognized_label = labels[np.argmin(distances)]
28 closest_face_index = np.argmin(distances)
29 #threshold = np.percentile(distances, 10)
30 # Use a fixed threshold
31 threshold = fixed_threshold
32
33 # Debugging: Print distances and threshold
34 print(f"Threshold: {threshold}")
35 print(f"Min distance: {min_distance}")
36
37 if min_distance > threshold:
38     recognized_label = "unknown"
39 else:
40     recognized_label = labels[closest_face_index]
41 # Reconstruct the projected face
42 # reconstructed_face = np.dot(projected_test_face, eigenfaces
   ) + mean_face
43 reconstructed_face = custom_dot(projected_test_face,
   eigenfaces) + mean_face
44
45 # Display the test face, mean face, closest face, and
   reconstructed face side by side
46 test_face_image = test_face.reshape((100,100))

```

```

43 mean_face_image = mean_face.reshape((100,100))
44 closest_face = original_faces[closest_face_index].reshape
   ((100,100))
45 reconstructed_face_image = reconstructed_face.reshape
   ((100,100))
46
47 # Normalize the images to the range [0, 255]
48 test_face_image = normalize(test_face_image, norm_type=cv2.
   NORM_MINMAX)
49 mean_face_image = normalize(mean_face_image, norm_type=cv2.
   NORM_MINMAX)
50 closest_face = normalize(closest_face, norm_type=cv2.
   NORM_MINMAX)
51 reconstructed_face_image = normalize(reconstructed_face_image
   , norm_type=cv2.NORM_MINMAX)
52
53 # Convert to uint8 type
54 test_face_image = test_face_image.astype(np.uint8)
55 mean_face_image = mean_face_image.astype(np.uint8)
56 closest_face = closest_face.astype(np.uint8)
57 reconstructed_face_image = reconstructed_face_image.astype(np
   .uint8)
58
59 # Resize images to be larger for better visibility
60 test_face_image = resize(test_face_image, (200, 200))
61 mean_face_image = resize(mean_face_image, (200, 200))
62 closest_face = resize(closest_face, (200, 200))
63 reconstructed_face_image = resize(reconstructed_face_image,
   (200, 200))
64
65 # Concatenate images horizontally
66 combined_image = np.hstack((test_face_image, mean_face_image,
   closest_face, reconstructed_face_image))
67
68 # Add labels to the combined image
69 labels = ["Test Face", "Mean Face", "Closest Face", "
   Reconstructed Face"]
70 positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
71 add_labels_to_image(combined_image, labels, positions)
72
73 # Save the result
74 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.

```

1 def save_results(results, filename):
2     """
3     Saves recognition results to a file.

```

```

4
5     Parameters:
6         results (list): List of recognition results.
7         filename (str): File to save the results.
8     """
9     with open(filename, 'w') as f:
10         for result in results:
11             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.

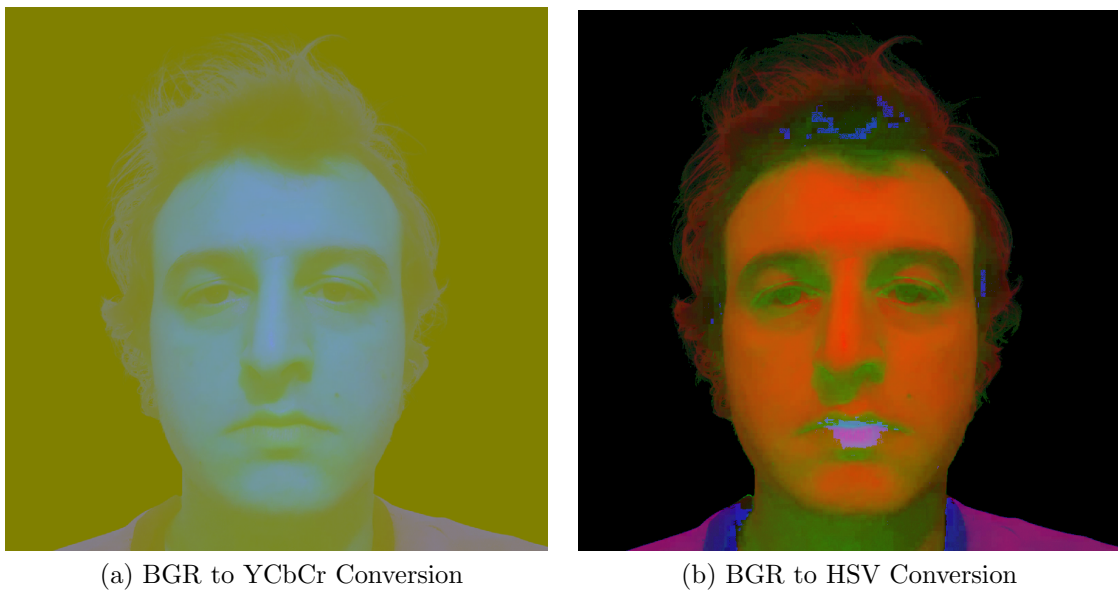


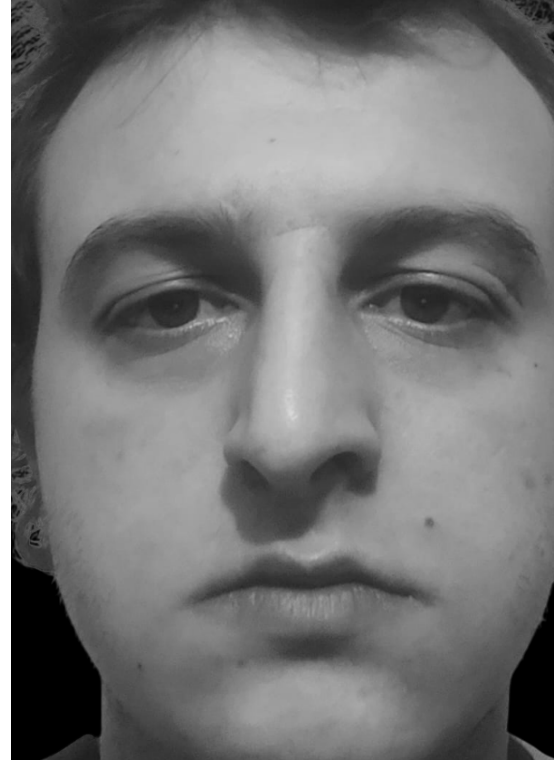
Figure 1: Color space conversions applied to the facial images.



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



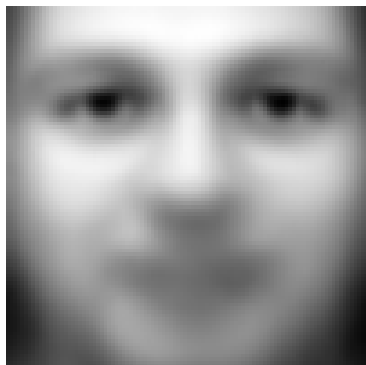
## 4.6 Key Observations

- **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.

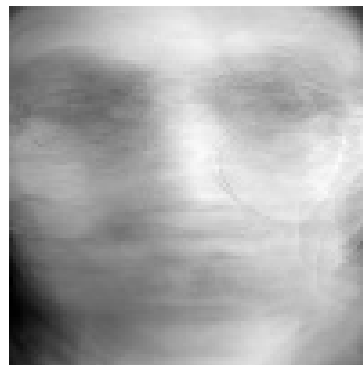
## 4.7 Visual Results

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

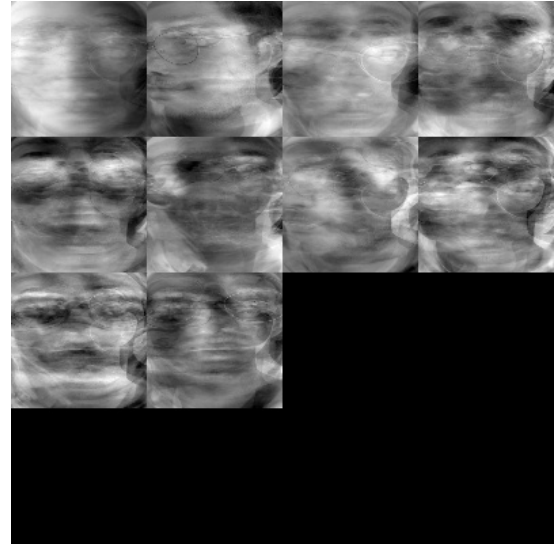


(b) Mean Face Computed from Our Dataset

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

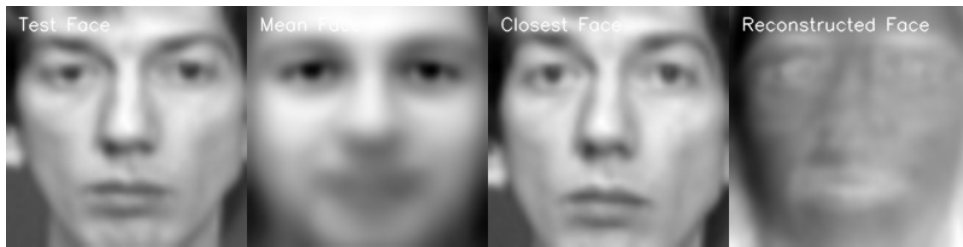


Figure 6: Example of Successful Recognition (Test Face, Mean Face, Closest Face, Reconstructed Face) from Olivetti Faces Dataset



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

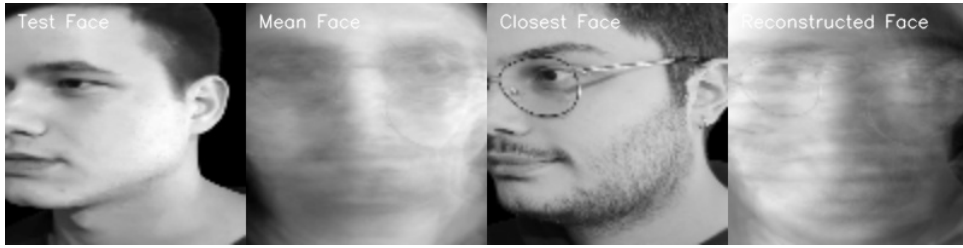


Figure 9: Example of Misclassification (Test Face, Mean Face, Closest Face, Reconstructed Face) from Our Dataset

## 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," AT&T Laboratories Cambridge, Available: [https://scikit-learn.org/0.19/datasets/olivetti\\_faces.html](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](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

```

1 import cv2
2 import numpy as np
3 from scipy.ndimage import convolve
4 import os
5
6 def bgr_to_hsv(image):
7     hsv_image = np.zeros_like(image, dtype=np.float32)
8     for i in range(image.shape[0]):
9         for j in range(image.shape[1]):
10             b, g, r = image[i, j] / 255.0 # Normalize the BGR
11                 values to [0, 1]
12             max_val = max(b, g, r)
13             min_val = min(b, g, r)
14             delta = max_val - min_val
15             if delta == 0:
16                 h = 0

```

```

16         elif max_val == r:
17             h = (60 * ((g - b) / delta) + 360) % 360
18         elif max_val == g:
19             h = (60 * ((b - r) / delta) + 120) % 360
20         elif max_val == b:
21             h = (60 * ((r - g) / delta) + 240) % 360
22         s = 0 if max_val == 0 else (delta / max_val)
23         v = max_val
24         hsv_image[i, j] = [h, s, v]
25     hsv_image[:, :, 0] = hsv_image[:, :, 0] / 2
26     hsv_image[:, :, 1:] *= 255
27     return hsv_image.astype(np.uint8)
28
29 def bgr_to_ycr_cb(image):
30     ycr_cb_image = np.zeros_like(image, dtype=np.float32)
31     for i in range(image.shape[0]):
32         for j in range(image.shape[1]):
33             b, g, r = image[i, j]
34             y = 0.299 * r + 0.587 * g + 0.114 * b
35             cr = (r - y) * 0.713 + 128
36             cb = (b - y) * 0.564 + 128
37             ycr_cb_image[i, j] = [y, cr, cb]
38     ycr_cb_image = np.clip(ycr_cb_image, 0, 255)
39     return ycr_cb_image.astype(np.uint8)
40
41 def in_range(image, lower_bound, upper_bound):
42     mask = np.zeros((image.shape[0], image.shape[1]), dtype=np.
43                     uint8)
44     for i in range(image.shape[0]):
45         for j in range(image.shape[1]):
46             if all(lower_bound <= image[i, j]) and all(image[i, j]
47                 <= upper_bound):
48                 mask[i, j] = 255
49             else:
50                 mask[i, j] = 0
51     return mask
52
53 def get_structuring_element(shape, ksize):
54     if shape != 'ellipse':
55         raise ValueError("Only 'ellipse' shape is supported in
56             this implementation")
57     rows, cols = ksize
58     kernel = np.zeros((rows, cols), dtype=np.uint8)
59     center_x, center_y = cols // 2, rows // 2
60     axes_x, axes_y = cols / 2, rows / 2
61     for i in range(rows):
62         for j in range(cols):
63             if ((j - center_x) ** 2) / (axes_x ** 2) + ((i -
64                 center_y) ** 2) / (axes_y ** 2) <= 1:
65                 kernel[i, j] = 1
66     return kernel

```



```

63
64 def bitwise_and(mask1, mask2):
65     # both masks have the same shape
66     assert mask1.shape == mask2.shape, "Masks must have the same
        shape"
67     result_mask = np.zeros_like(mask1, dtype=np.uint8)
68     for i in range(mask1.shape[0]):
69         for j in range(mask1.shape[1]):
70             # Perform bitwise AND operation
71             result_mask[i, j] = mask1[i, j] & mask2[i, j]
72     return result_mask
73
74 def erode(image, kernel, iterations=1):
75     # Get the dimensions of the image and kernel
76     img_h, img_w = image.shape
77     k_h, k_w = kernel.shape
78     pad_h, pad_w = k_h // 2, k_w // 2
79
80     # Pad the image to handle borders
81     padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)),
        , mode='constant', constant_values=255)
82
83     for _ in range(iterations):
84         eroded_image = np.copy(image)
85
86         # Iterate over each pixel in the image
87         for i in range(img_h):
88             for j in range(img_w):
89                 # Extract the region of interest
90                 roi = padded_image[i:i + k_h, j:j + k_w]
91
92                 # Apply the kernel (structuring element)
93                 if np.all(roi[kernel == 1] == 255):
94                     eroded_image[i, j] = 255
95                 else:
96                     eroded_image[i, j] = 0
97
98                 # Update the padded image for the next iteration
99                 padded_image = np.pad(eroded_image, ((pad_h, pad_h), (
        pad_w, pad_w)), mode='constant', constant_values=255)
100
101     return eroded_image
102
103 def dilate(image, kernel, iterations=1):
104     # Get the dimensions of the image and kernel
105     img_h, img_w = image.shape
106     k_h, k_w = kernel.shape
107     pad_h, pad_w = k_h // 2, k_w // 2
108
109     # Pad the image to handle borders

```

```

110 padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w))
    , mode='constant', constant_values=0)
111
112 for _ in range(iterations):
113     dilated_image = np.copy(image)
114
115     # Iterate over each pixel in the image
116     for i in range(img_h):
117         for j in range(img_w):
118             # Extract the region of interest
119             roi = padded_image[i:i + k_h, j:j + k_w]
120
121             # Apply the kernel (structuring element)
122             if np.any(roi[kernel == 1] == 255):
123                 dilated_image[i, j] = 255
124             else:
125                 dilated_image[i, j] = 0
126
127             # Update the padded image for the next iteration
128             padded_image = np.pad(dilated_image, ((pad_h, pad_h), (
                pad_w, pad_w)), mode='constant', constant_values=0)
129
130     return dilated_image
131
132 def gaussian_kernel(size, sigma=1.0):
133     kernel = np.fromfunction(
134         lambda x, y: (1 / (2 * np.pi * sigma ** 2)) * np.exp(
135             -((x - (size - 1) / 2) ** 2 + (y - (size - 1) / 2) **
136               2) / (2 * sigma ** 2)
137         ),
138         (size, size)
139     )
140     return kernel / np.sum(kernel)
141
142 def gaussian_filter(image, sigma=1.0):
143     size = int(2 * np.ceil(3 * sigma) + 1)
144     kernel = gaussian_kernel(size, sigma)
145     return convolve(image, kernel)
146
147 def sobel(image, axis):
148     """Applies Sobel filter to the image to compute the gradient.
149     """
150     if axis == 0:
151         kernel = np.array([[ -1,  0,  1], [ -2,  0,  2], [ -1,  0,  1]])
152     elif axis == 1:
153         kernel = np.array([[ 1,  2,  1], [ 0,  0,  0], [ -1, -2, -1]])
154     else:
155         raise ValueError("Axis must be 0 (x) or 1 (y)")
156     return convolve(image, kernel)
157
158 def canny(image, low_threshold, high_threshold):

```

```

157 blurred_image = gaussian_filter(image, sigma=1.4)
158
159 # Compute gradient intensity and direction
160 grad_x = sobel(blurred_image, axis=0)
161 grad_y = sobel(blurred_image, axis=1)
162 gradient_magnitude = np.hypot(grad_x, grad_y)
163 gradient_direction = np.arctan2(grad_y, grad_x) * (180 / np.
    pi)
164 gradient_direction[gradient_direction < 0] += 180
165
166 # Apply non-maximum suppression
167 nms_image = np.zeros_like(gradient_magnitude)
168 for i in range(1, image.shape[0] - 1):
169     for j in range(1, image.shape[1] - 1):
170         angle = gradient_direction[i, j]
171         q = 255
172         r = 255
173
174         if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
175             q = gradient_magnitude[i, j + 1]
176             r = gradient_magnitude[i, j - 1]
177         elif 22.5 <= angle < 67.5:
178             q = gradient_magnitude[i + 1, j - 1]
179             r = gradient_magnitude[i - 1, j + 1]
180         elif 67.5 <= angle < 112.5:
181             q = gradient_magnitude[i + 1, j]
182             r = gradient_magnitude[i - 1, j]
183         elif 112.5 <= angle < 157.5:
184             q = gradient_magnitude[i - 1, j - 1]
185             r = gradient_magnitude[i + 1, j + 1]
186
187         if gradient_magnitude[i, j] >= q and
            gradient_magnitude[i, j] >= r:
188             nms_image[i, j] = gradient_magnitude[i, j]
189         else:
190             nms_image[i, j] = 0
191
192 # Apply double threshold
193 strong_edges = (nms_image > high_threshold).astype(np.uint8)
194 weak_edges = ((nms_image >= low_threshold) & (nms_image <=
    high_threshold)).astype(np.uint8)
195
196 # Track edges by hysteresis
197 edges = np.zeros_like(image, dtype=np.uint8)
198 for i in range(1, image.shape[0] - 1):
199     for j in range(1, image.shape[1] - 1):
200         if strong_edges[i, j]:
201             edges[i, j] = 255
202         elif weak_edges[i, j]:
203             if (strong_edges[i + 1, j - 1:j + 2].any() or
204                 strong_edges[i - 1, j - 1:j + 2].any() or

```

```

205         strong_edges[i, [j - 1, j + 1]].any()):
206         edges[i, j] = 255
207
208     return edges
209
210 def find_contours(image):
211     contours = []
212     visited = np.zeros_like(image, dtype=bool)
213
214     def is_valid(x, y):
215         return 0 <= x < image.shape[0] and 0 <= y < image.shape
            [1]
216
217     def trace_contour(start):
218         contour = []
219         stack = [start]
220         directions = [(-1, 0), (0, -1), (1, 0), (0, 1)]
221
222         while stack:
223             x, y = stack.pop()
224             if visited[x, y]:
225                 continue
226             visited[x, y] = True
227             contour.append((x, y))
228
229             for dx, dy in directions:
230                 nx, ny = x + dx, y + dy
231                 if is_valid(nx, ny) and image[nx, ny] == 255 and
                    not visited[nx, ny]:
232                     stack.append((nx, ny))
233
234         return contour
235
236     for i in range(image.shape[0]):
237         for j in range(image.shape[1]):
238             if image[i, j] == 255 and not visited[i, j]:
239                 contour = trace_contour((i, j))
240                 if contour:
241                     contours.append(np.array(contour))
242
243     return contours, None
244
245 def bounding_rect(contour):
246     contour = np.array(contour) # Convert the contour to a NumPy
        array
247     x_min = np.min(contour[:, 1])
248     y_min = np.min(contour[:, 0])
249     x_max = np.max(contour[:, 1])
250     y_max = np.max(contour[:, 0])
251     return x_min, y_min, x_max - x_min, y_max - y_min
252

```

```

253 def bgr_to_gray(image):
254     # Create an empty array for the grayscale image
255     gray_image = np.zeros((image.shape[0], image.shape[1]), dtype
                             =np.uint8)
256
257     # Iterate over each pixel
258     for i in range(image.shape[0]):
259         for j in range(image.shape[1]):
260             b, g, r = image[i, j]
261             gray = int(0.299 * r + 0.587 * g + 0.114 * b)
262             gray_image[i, j] = gray
263
264     return gray_image
265
266 def crop_face_using_edges(image_path, output_path):
267     image = cv2.imread(image_path)
268     if image is None:
269         raise ValueError("Image not found or path is incorrect.")
270
271     # Convert the image to HSV and YCbCr color spaces
272     hsv_image = bgr_to_hsv(image)
273     ycbcr_image = bgr_to_ycrcb(image)
274
275     # Define skin color thresholds
276     # HSV thresholds for skin color
277     lower_hsv = np.array([0, 30, 50], dtype=np.uint8)
278     upper_hsv = np.array([50, 255, 255], dtype=np.uint8)
279     mask_hsv = in_range(hsv_image, lower_hsv, upper_hsv)
280
281     # YCbCr thresholds for skin color
282     lower_ycbcr = np.array([0, 128, 80], dtype=np.uint8)
283     upper_ycbcr = np.array([255, 180, 135], dtype=np.uint8)
284     mask_ycbcr = in_range(ycbcr_image, lower_ycbcr, upper_ycbcr)
285
286     # Combine the masks
287     skin_mask = bitwise_and(mask_hsv, mask_ycbcr)
288
289     # Morphological operations to remove noise
290     kernel = get_structuring_element('ellipse', (5, 5))
291     skin_mask = erode(skin_mask, kernel, iterations=2)
292     skin_mask = dilate(skin_mask, kernel, iterations=2)
293
294     # Detect edges in the skin mask
295     edges = canny(skin_mask, 100, 150)
296
297     # Find contours of the edges
298     contours, _ = find_contours(skin_mask)
299
300     max_area = 0
301     bounding_box = None
302

```

```

303     for contour in contours:
304         x, y, w, h = bounding_rect(contour)
305         area = w * h
306         if area > max_area:
307             max_area = area
308             bounding_box = (x, y, w, h)
309
310     if bounding_box:
311         x, y, w, h = bounding_box
312         # Narrow the cropping region horizontally
313         narrow_factor_x = 0.1 # Adjust this value to control
                                # horizontal narrowing
314         reduction_x = int(w * narrow_factor_x)
315         x += reduction_x
316         w -= 2 * reduction_x
317         # Narrow the cropping region vertically
318         narrow_factor_y = 0.1 # Adjust this value to control
                                # vertical narrowing
319         reduction_y = int(h * narrow_factor_y)
320         y += reduction_y
321         h -= 2 * reduction_y
322         # Ensure the new region is within bounds
323         x = max(0, x)
324         y = max(0, y)
325         w = max(1, w)
326         h = max(1, h)
327         cropped_region = image[y:y+h, x:x+w]
328
329         gray = bgr_to_gray(cropped_region)
330
331         cv2.imwrite(output_path, gray)
332     else:
333         print(f"No face detected in {image_path}.")
334
335 def process_images_in_folder(input_folder, output_folder):
336     if not os.path.exists(output_folder):
337         os.makedirs(output_folder)
338     for filename in os.listdir(input_folder):
339         input_path = os.path.join(input_folder, filename)
340         if not (filename.lower().endswith(".png") or filename.
341                 lower().endswith(".jpg") or filename.lower().endswith(
342                     ".jpeg")):
343             continue
344         print(f"Processing: {input_path}")
345         try:
346             output_path = os.path.join(output_folder, f"
347                                     processed_{filename}")
348             crop_face_using_edges(input_path, output_path)
349         except Exception as e:
350             print(f"Error processing {filename}: {e}")

```

```

348     print(f"Processing completed. Processed images saved in: {
        output_folder}")
349
350 process_images_in_folder("./faces_dataset_black", "./
    output_images2")

```

## 7.2.2 resize.py

```

1  import os
2  import cv2
3  import numpy as np
4  def resize(src, dsize, interpolation="bilinear"):
5      src_height, src_width = src.shape[:2]
6      dst_width, dst_height = dsize
7
8      if len(src.shape) == 3: # Multi-channel image
9          channels = src.shape[2]
10         dst = np.zeros((dst_height, dst_width, channels), dtype=
            src.dtype)
11     else: # Grayscale image
12         dst = np.zeros((dst_height, dst_width), dtype=src.dtype)
13
14     for i in range(dst_height):
15         for j in range(dst_width):
16             src_x = j * (src_width / dst_width)
17             src_y = i * (src_height / dst_height)
18             src_x0 = int(np.floor(src_x))
19             src_y0 = int(np.floor(src_y))
20             src_x1 = min(src_x0 + 1, src_width - 1)
21             src_y1 = min(src_y0 + 1, src_height - 1)
22
23             dx = src_x - src_x0
24             dy = src_y - src_y0
25
26             if len(src.shape) == 3: # Multi-channel
27                 for c in range(src.shape[2]):
28                     dst[i, j, c] = (
29                         (1 - dx) * (1 - dy) * src[src_y0, src_x0,
                            c] +
30                         dx * (1 - dy) * src[src_y0, src_x1, c] +
31                         (1 - dx) * dy * src[src_y1, src_x0, c] +
32                         dx * dy * src[src_y1, src_x1, c]
33                     )
34             else: # Grayscale
35                 dst[i, j] = (
36                     (1 - dx) * (1 - dy) * src[src_y0, src_x0] +
37                     dx * (1 - dy) * src[src_y0, src_x1] +
38                     (1 - dx) * dy * src[src_y1, src_x0] +
39                     dx * dy * src[src_y1, src_x1]
40                 )

```

```

41     return dst
42
43
44 def process_images_in_folder(input_folder, output_folder, dsize):
45     if not os.path.exists(output_folder):
46         os.makedirs(output_folder)
47
48     for filename in os.listdir(input_folder):
49         input_path = os.path.join(input_folder, filename)
50
51         if not (filename.lower().endswith(".png") or filename.
52                 lower().endswith(".jpg") or filename.lower().endswith(
53                     ".jpeg")):
54             continue
55
56         print(f"Processing: {input_path}")
57         try:
58             image = cv2.imread(input_path)
59             if image is None:
60                 print(f"Skipping {filename}: Unable to read file.
61                     ")
62                 continue
63
64             resized_image = resize(image, dsize)
65
66             output_path = os.path.join(output_folder, filename)
67             cv2.imwrite(output_path, resized_image)
68         except Exception as e:
69             print(f"Error processing {filename}: {e}")
70
71         print(f"Processing completed. Resized images saved in: {
72             output_folder}")
73
74 if __name__ == "__main__":
75     input_folder = "output_images2"
76     output_folder = "resized_images2"
77     dsize = (100, 100)
78     process_images_in_folder(input_folder, output_folder, dsize)

```

### 7.2.3 fasttest.py

```

1 import os
2 import numpy as np
3 import cv2
4 from tqdm import tqdm
5 from sklearn.model_selection import train_test_split
6
7 def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
8     src_height, src_width = src.shape[:2]
9     dst_width, dst_height = dsize

```



```

10     dst = np.zeros((dst_height, dst_width), dtype=src.dtype)
11
12     for i in range(dst_height):
13         for j in range(dst_width):
14             src_x = j * (src_width / dst_width)
15             src_y = i * (src_height / dst_height)
16             src_x0 = int(np.floor(src_x))
17             src_y0 = int(np.floor(src_y))
18             src_x1 = min(src_x0 + 1, src_width - 1)
19             src_y1 = min(src_y0 + 1, src_height - 1)
20
21             dx = src_x - src_x0
22             dy = src_y - src_y0
23
24             dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0]
25                 + \
26                     dx * (1 - dy) * src[src_y0, src_x1] + \
27                     (1 - dx) * dy * src[src_y1, src_x0] + \
28                     dx * dy * src[src_y1, src_x1]
29
30     return dst
31
32 def custom_mean(array, axis=None):
33     if axis is None:
34         return sum(array) / len(array)
35     else:
36         return np.sum(array, axis=axis) / array.shape[axis]
37
38 def save_Results(base_dir="results"):
39     if not os.path.exists(base_dir):
40         os.makedirs(base_dir)
41
42     subdirectories = [d for d in os.listdir(base_dir) if os.path.
43                     isdir(os.path.join(base_dir, d))]
44     run_number = len(subdirectories)
45     subdirectory = os.path.join(base_dir, f"results_{run_number}"
46                                )
47     os.makedirs(subdirectory)
48     return subdirectory
49
50 def save_image_to_file(image, filename):
51     cv2.imwrite(filename, image)
52
53 def custom_dot(a, b):
54     result = []
55     for a_row in tqdm(a, desc="Computing dot product (outer loop)
56                        "):
57         row_result = []
58         for b_col in tqdm(zip(*b), desc="Computing dot product (
59                             inner loop)", leave=False):
60             row_result.append(sum(x * y for x, y in zip(a_row,
61                                                         b_col)))

```

```

55         result.append(row_result)
56     return np.array(result)
57
58 def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.
59     NORM_MINMAX):
60     if dst is None:
61         dst = np.zeros_like(src)
62
63     if norm_type == cv2.NORM_MINMAX:
64         min_val = np.min(src)
65         max_val = np.max(src)
66         dst = (src - min_val) * (beta - alpha) / (max_val -
67             min_val) + alpha
68     else:
69         raise NotImplementedError("Only NORM_MINMAX is
70             implemented")
71
72     return dst
73
74 # Add labels to the image
75 def add_labels_to_image(image, labels, positions, font=cv2.
76     FONT_HERSHEY_SIMPLEX, font_scale=0.5, color=(255, 255, 255),
77     thickness=1):
78     for label, position in zip(labels, positions):
79         cv2.putText(image, label, position, font, font_scale,
80             color, thickness, cv2.LINE_AA)
81
82 # This function loads our dataset of Olivetti faces and resizes
83 # them to a target size.
84 # It returns a list of image paths and their corresponding labels
85 .
86 def load_olivetti_faces(root_dir, target_size=(100, 100)):
87     image_paths = []
88     labels = []
89
90     print(f"Loading images from: {root_dir}")
91     # Check if the root directory exists
92     if not os.path.exists(root_dir):
93         print(f"Error: The directory {root_dir} does not exist.")
94         return image_paths, labels
95
96     for person_dir in tqdm(os.listdir(root_dir), desc="Processing
97         directories"):
98         person_path = os.path.join(root_dir, person_dir)
99         if os.path.isdir(person_path):
100             for filename in os.listdir(person_path):
101                 if filename.endswith(('.jpg', '.jpeg', '.png')):
102                     full_path = os.path.join(person_path,
103                         filename)
104                     image = cv2.imread(full_path, cv2.
105                         IMREAD_GRAYSCALE)

```

```

95         if image is None:
96             print(f"Error loading image: {full_path}"
97                 )
98             continue
99         # resized_image = resize(image, target_size)
100         resized_image = cv2.resize(image, target_size
101                                     )
102         save_path = os.path.join(person_path,
103                                   filename)
104         cv2.imwrite(save_path, resized_image)
105         image_paths.append(save_path)
106         labels.append(person_dir)
107         print(f"Loaded and resized: {person_dir} - {
108               filename}")
109     print(f"Total images loaded: {len(image_paths)}")
110     return image_paths, labels
111
112 # Step 2: Preprocess Images (Load, Flatten, Normalize)
113 def preprocess_images(image_paths):
114     dataset = []
115     for path in tqdm(image_paths, desc="Preprocessing images"):
116         image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
117         if image is None:
118             print(f"Error loading image: {path}")
119             continue
120         # Flatten and normalize the image
121         flattened = (image.flatten() / 255.0)
122         dataset.append(flattened)
123     return np.array(dataset, dtype=np.float64)
124
125 # Step 3: Compute Mean Face
126 def compute_mean_face(dataset):
127     # mean_face = custom_mean(dataset, axis=0)
128     mean_face = np.mean(dataset, axis=0)
129
130     # Reshape the mean face to its original dimensions (50x50)
131     mean_face_image = mean_face.reshape((100, 100))
132
133     # Normalize the mean face image to the range [0, 255]
134     # mean_face_image = normalize(mean_face_image, norm_type=cv2.
135     #                             NORM_MINMAX)
136     mean_face_image = cv2.normalize(mean_face_image, None, 0,
137                                     255, cv2.NORM_MINMAX)
138
139     # Convert to uint8 type
140     mean_face_image = mean_face_image.astype(np.uint8)
141     cv2.imshow("Mean Face", mean_face_image)
142     cv2.waitKey(0)
143     cv2.destroyAllWindows()
144     return mean_face, mean_face_image

```

```

140 # Step 4: Center Dataset
141 def center_dataset(dataset, mean_face):
142     return dataset - mean_face
143
144 # Step 5: Compute Covariance Matrix
145 def compute_covariance_matrix(centered_data):
146     num_images, num_features = centered_data.shape
147     # return custom_dot(centered_data.T, centered_data) /
148     #     num_images
149     return np.dot(centered_data, centered_data.T) / num_images
150
151 def compute_eigenfaces(centered_data, covariance_matrix,
152     num_eigenfaces):
153     eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
154     sorted_indices = np.argsort(eigenvalues)[::-1]
155     eigenvectors = eigenvectors[:, sorted_indices]
156     eigenvalues = eigenvalues[sorted_indices]
157     eigenvectors = eigenvectors[:, :num_eigenfaces]
158     eigenfaces = np.dot(eigenvectors.T, centered_data)
159     return eigenfaces, eigenvalues[:num_eigenfaces]
160
161 def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape
162     =(100, 100), results_subdirectory=None):
163     # Calculate the grid size dynamically
164     grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
165     grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))
166
167     # Create a blank canvas to display the eigenfaces
168     grid_height = grid_rows * image_shape[0]
169     grid_width = grid_cols * image_shape[1]
170     canvas = np.zeros((grid_height, grid_width), dtype=np.uint8)
171
172     for i in range(num_eigenfaces):
173         row = i // grid_cols
174         col = i % grid_cols
175         eigenface = eigenfaces[i].reshape(image_shape)
176         # eigenface = normalize(eigenface, norm_type=cv2.
177         #     NORM_MINMAX)
178         eigenface = cv2.normalize(eigenface, None, 0, 255, cv2.
179             NORM_MINMAX)
180         eigenface = eigenface.astype(np.uint8)
181         canvas[row * image_shape[0]:(row + 1) * image_shape[0],
182             col * image_shape[1]:(col + 1) * image_shape[1]] =
183             eigenface
184
185     # Display the canvas with all eigenfaces
186     cv2.imshow("Eigenfaces", canvas)
187     cv2.waitKey(0)
188     cv2.destroyAllWindows()
189
190     # Save the canvas with all eigenfaces

```

```

184     if results_subdirectory:
185         eigenfaces_path = os.path.join(results_subdirectory, "
186             eigenfaces.jpg")
187         cv2.imwrite(eigenfaces_path, canvas)
188
189 # Step 7: Project Faces Onto Eigenfaces
190 def project_faces(centered_data, eigenfaces):
191     # return custom_dot(centered_data, eigenfaces.T)
192     return np.dot(centered_data, eigenfaces.T)
193
194 def save_results(results, filename):
195     with open(filename, 'w') as f:
196         for result in results:
197             f.write(f"{result}\n")
198
199 # Step 8: Recognize Test Face
200 def recognize_face(test_face, mean_face, eigenfaces,
201     projected_faces, labels, original_faces, fixed_threshold=300):
202     centered_test_face = test_face - mean_face
203     projected_test_face = np.dot(centered_test_face, eigenfaces.T
204     )
205     distances = np.linalg.norm(projected_faces -
206         projected_test_face, axis=1)
207     min_distance = np.min(distances)
208     recognized_label = labels[np.argmin(distances)]
209     closest_face_index = np.argmin(distances)
210
211     # Use a fixed threshold
212     threshold = fixed_threshold
213
214     if min_distance > threshold:
215         recognized_label = "unknown"
216     else:
217         recognized_label = labels[closest_face_index]
218
219     # Debugging: Print distances and threshold
220     print(f"Threshold: {threshold}")
221     print(f"Min distance: {min_distance}")
222
223     # Reconstruct the projected face
224     reconstructed_face = np.dot(projected_test_face, eigenfaces)
225     + mean_face
226
227     # Display the test face, mean face, closest face, and
228     reconstructed face side by side
229     test_face_image = test_face.reshape((100, 100))
230     mean_face_image = mean_face.reshape((100, 100))
231     closest_face = original_faces[closest_face_index].reshape
232         ((100, 100))
233     reconstructed_face_image = reconstructed_face.reshape((100,
234         100))

```

```

227 norm_type=cv2.NORM_MINMAX)
228 test_face_image = cv2.normalize(test_face_image, None, 0,
229 255, cv2.NORM_MINMAX)
230 mean_face_image = cv2.normalize(mean_face_image, None, 0,
231 255, cv2.NORM_MINMAX)
232 closest_face = cv2.normalize(closest_face, None, 0, 255, cv2.
233 NORM_MINMAX)
234 reconstructed_face_image = cv2.normalize(
235 reconstructed_face_image, None, 0, 255, cv2.NORM_MINMAX)
236
237 # Convert to uint8 type
238 test_face_image = test_face_image.astype(np.uint8)
239 mean_face_image = mean_face_image.astype(np.uint8)
240 closest_face = closest_face.astype(np.uint8)
241 reconstructed_face_image = reconstructed_face_image.astype(np
242 .uint8)
243
244 test_face_image = cv2.resize(test_face_image, (200, 200))
245 mean_face_image = cv2.resize(mean_face_image, (200, 200))
246 closest_face = cv2.resize(closest_face, (200, 200))
247 reconstructed_face_image = cv2.resize(
248 reconstructed_face_image, (200, 200))
249
250 # Concatenate images horizontally
251 combined_image = np.hstack((test_face_image, mean_face_image,
252 closest_face, reconstructed_face_image))
253
254 # Add labels to the combined image
255 labels = ["Test Face", "Mean Face", "Closest Face", "
256 Reconstructed Face"]
257 positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
258 add_labels_to_image(combined_image, labels, positions)
259 # Save the result
260 return recognized_label , combined_image
261
262 # Paths
263 Olivetti_faces_dir = "real_dataset" # Directory with Olivetti
264 faces
265
266 # Step 1: Load and Resize Olivetti Faces
267 image_paths, labels = load_olivetti_faces(Olivetti_faces_dir)
268
269 # Step 2: Preprocess Dataset
270 dataset = preprocess_images(image_paths)
271
272 # Split the dataset into training and test sets
273 train_paths, test_paths, train_labels, test_labels =
274 train_test_split(image_paths, labels, test_size=0.2)
275
276 # Preprocess training and test sets

```

```

268 train_dataset = preprocess_images(train_paths)
269 test_dataset = preprocess_images(test_paths)
270
271 # Step 3: Compute Mean Face
272 mean_face, mean_face_image_to_save = compute_mean_face(
    train_dataset)
273
274 # Step 4: Center the Dataset
275 print("Centering the dataset")
276 centered_train_data = center_dataset(train_dataset, mean_face)
277 print("Dataset is centered")
278 print("covariance matrix is calculating")
279 # Step 5: Compute Covariance Matrix
280 covariance_matrix = compute_covariance_matrix(centered_train_data
    )
281 print("Covariance matrix is calculated ")
282 # Step 6: Compute Eigenfaces
283 num_eigenfaces = min(10, train_dataset.shape[0])
284 eigenfaces, eigenvalues = compute_eigenfaces(centered_train_data,
    covariance_matrix, num_eigenfaces)
285 print("eigenfaces are calculated ")
286 # Create a subdirectory for this run's results
287 results_subdirectory = save_Results()
288
289 # Save mean face
290 mean_face_path = os.path.join(results_subdirectory, "mean_face.
    jpg")
291 cv2.imwrite(mean_face_path, mean_face_image_to_save)
292
293 # Display and save eigenfaces
294 display_eigenfaces(eigenfaces, num_eigenfaces,
    results_subdirectory=results_subdirectory)
295
296 # Step 7: Project Faces Onto Eigenfaces
297 projected_train_faces = project_faces(centered_train_data,
    eigenfaces)
298
299 # Test the recognition on the test set
300 correct_predictions = 0
301 unknown = 0
302 count = 1
303 results = []
304 for test_image_path, true_label in tqdm(zip(test_paths,
    test_labels), desc="Recognizing faces", total=len(test_paths))
    :
305     test_face_dataset = preprocess_images([test_image_path])
306     recognized_label, combined_image = recognize_face(
307         test_face_dataset[0],
308         mean_face,
309         eigenfaces,
310         projected_train_faces,

```

```

311     train_labels,
312     train_dataset, # Pass the true label to the function
313     fixed_threshold=300 # Set a fixed threshold
314 )
315 if recognized_label == "unknown":
316     print(f"Unknown face: {test_image_path}")
317     result = f"Test image: {true_label}, True label: {
318         true_label}, Recognized label: unknown , UNSUCCESSFUL"
319     output_path = os.path.join(results_subdirectory, f"{count
320         }_{true_label}_recognized_as_unknown_result.jpg")
321     unknown += 1
322 elif recognized_label == true_label:
323     print(f"TRUE")
324     result = f"Test image: {true_label}, True label: {
325         true_label}, Recognized label: {recognized_label} ,
326         SUCCESSFUL"
327     output_path = os.path.join(results_subdirectory, f"{count
328         }_{true_label}_recognized_as{true_label}_result.jpg")
329     correct_predictions += 1
330 else:
331     print(f"True label: {true_label}, Recognized as: {
332         recognized_label}")
333     result = f"Test image: {true_label}, True label: {
334         true_label}, Recognized label: {recognized_label} ,
335         UNSUCCESSFUL"
336     output_path = os.path.join(results_subdirectory, f"{count
337         }_{true_label}_recognized_as{recognized_label}_result.
338         jpg")
339     count += 1
340 # Debugging: Print output_path
341 print(f"Saving image to: {output_path}")
342
343 # Save the result
344 success = cv2.imwrite(output_path, combined_image)
345 if not success:
346     print(f"Failed to write image to: {output_path}")
347
348 results.append(result)
349
350 # Save all results to a file
351
352 # Calculate and print the recognition accuracy
353 accuracy = correct_predictions / len(test_paths)
354 print(f"Recognition accuracy: {accuracy * 100:.2f}%")
355 print(f"Unknown recognition: {unknown} out of {len(test_paths)}")
356 results.append(f"Recognition accuracy: {accuracy * 100:.2f}%")
357 results.append(f"Unknown recognition: {unknown} out of {len(
358     test_paths)}")
359 results_file = os.path.join(results_subdirectory, "results.txt")
360 save_results(results, results_file)

```



## 7.2.4 customimplrecognition.py

```
1 import os
2 import numpy as np
3 import cv2
4 from tqdm import tqdm
5 from sklearn.model_selection import train_test_split
6
7 def resize(src, dsize, interpolation=cv2.INTER_LINEAR):
8     src_height, src_width = src.shape[:2]
9     dst_width, dst_height = dsize
10    dst = np.zeros((dst_height, dst_width), dtype=src.dtype)
11
12    for i in range(dst_height):
13        for j in range(dst_width):
14            src_x = j * (src_width / dst_width)
15            src_y = i * (src_height / dst_height)
16            src_x0 = int(np.floor(src_x))
17            src_y0 = int(np.floor(src_y))
18            src_x1 = min(src_x0 + 1, src_width - 1)
19            src_y1 = min(src_y0 + 1, src_height - 1)
20
21            dx = src_x - src_x0
22            dy = src_y - src_y0
23
24            dst[i, j] = (1 - dx) * (1 - dy) * src[src_y0, src_x0]
25                        + \
26                        dx * (1 - dy) * src[src_y0, src_x1] + \
27                        (1 - dx) * dy * src[src_y1, src_x0] + \
28                        dx * dy * src[src_y1, src_x1]
29
30    return dst
31
32 def custom_mean(array, axis=None):
33     if axis is None:
34         return sum(array) / len(array)
35     else:
36         return np.sum(array, axis=axis) / array.shape[axis]
37
38 def create_subdirectories(base_dir="results"):
39     if not os.path.exists(base_dir):
40         os.makedirs(base_dir)
41
42     subdirectories = [d for d in os.listdir(base_dir) if os.path.
43                     isdir(os.path.join(base_dir, d))]
44     run_number = len(subdirectories)
45     subdirectory = os.path.join(base_dir, f"results_{run_number}"
46                                )
47     os.makedirs(subdirectory)
48     return subdirectory
49
50 def save_image_to_file(image, filename):
```

```

47     cv2.imwrite(filename, image)
48
49 def custom_dot(a, b):
50     result = []
51     for a_row in tqdm(a, desc="Computing dot product (outer loop)"):
52         row_result = []
53         for b_col in tqdm(zip(*b), desc="Computing dot product (inner loop)", leave=False):
54             row_result.append(sum(x * y for x, y in zip(a_row, b_col)))
55         result.append(row_result)
56     return np.array(result)
57
58 def normalize(src, dst=None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX):
59     if dst is None:
60         dst = np.zeros_like(src)
61
62     if norm_type == cv2.NORM_MINMAX:
63         min_val = np.min(src)
64         max_val = np.max(src)
65         dst = (src - min_val) * (beta - alpha) / (max_val - min_val) + alpha
66     else:
67         raise NotImplementedError("Only NORM_MINMAX is implemented")
68
69     return dst
70
71 # Add labels to the image
72 def add_labels_to_image(image, labels, positions, font=cv2.FONT_HERSHEY_SIMPLEX, font_scale=0.5, color=(255, 255, 255), thickness=1):
73     for label, position in zip(labels, positions):
74         cv2.putText(image, label, position, font, font_scale, color, thickness, cv2.LINE_AA)
75
76 # This function loads our dataset of Olivetti faces and resizes them to a target size.
77 # It returns a list of image paths and their corresponding labels
78
79 def load_faces(root_dir, target_size=(100,100)):
80     image_paths = []
81     labels = []
82
83     print(f>Loading images from: {root_dir}")
84     # Check if the root directory exists
85     if not os.path.exists(root_dir):
86         print(f>Error: The directory {root_dir} does not exist.")
87     return image_paths, labels

```

```

87
88     for person_dir in tqdm(os.listdir(root_dir), desc="Processing
89         directories"):
90         person_path = os.path.join(root_dir, person_dir)
91         if os.path.isdir(person_path):
92             for filename in os.listdir(person_path):
93                 if filename.endswith(('.jpg', '.jpeg', '.png')):
94                     full_path = os.path.join(person_path,
95                         filename)
96                     image = cv2.imread(full_path, cv2.
97                         IMREAD_GRAYSCALE)
98                     if image is None:
99                         print(f"Error loading image: {full_path}"
100                             )
101                         continue
102                     resized_image = resize(image, target_size)
103                     save_path = os.path.join(person_path,
104                         filename)
105                     cv2.imwrite(save_path, resized_image)
106                     image_paths.append(save_path)
107                     labels.append(person_dir)
108                     print(f"Loaded and resized: {person_dir} - {
109                         filename}")
110
111     print(f"Total images loaded: {len(image_paths)}")
112     return image_paths, labels
113
114 # Step 2: Preprocess Images (Load, Flatten, Normalize)
115 def preprocess_images(image_paths):
116     dataset = []
117     for path in tqdm(image_paths, desc="Preprocessing images"):
118         image = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
119         if image is None:
120             print(f"Error loading image: {path}")
121             continue
122         # Flatten and normalize the image
123         flattened = (image.flatten() / 255.0)
124         dataset.append(flattened)
125     return np.array(dataset, dtype=np.float64)
126
127 # Step 3: Compute Mean Face
128 def compute_mean_face(dataset):
129     mean_face = custom_mean(dataset, axis=0)
130
131     # Reshape the mean face to its original dimensions (50x50)
132     mean_face_image = mean_face.reshape((100,100))
133
134     # Normalize the mean face image to the range [0, 255]
135     mean_face_image = normalize(mean_face_image, norm_type=cv2.
136         NORM_MINMAX)
137
138     # Convert to uint8 type

```

```

131     mean_face_image = mean_face_image.astype(np.uint8)
132
133     return mean_face, mean_face_image
134
135 # Step 4: Center Dataset
136 def center_dataset(dataset, mean_face):
137     return dataset - mean_face
138
139 # Step 5: Compute Covariance Matrix
140 def compute_covariance_matrix(centered_data):
141     num_images, num_features = centered_data.shape
142     return custom_dot(centered_data.T, centered_data) /
        num_images
143
144 # Step 6: Perform Eigen Decomposition
145 def compute_eigenfaces(centered_data, covariance_matrix,
        num_eigenfaces):
146     print("Computing eigenfaces")
147     eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
148     sorted_indices = np.argsort(eigenvalues)[::-1]
149     eigenvectors = eigenvectors[:, sorted_indices]
150     eigenvalues = eigenvalues[sorted_indices]
151     eigenvectors = eigenvectors[:, :num_eigenfaces]
152     eigenfaces = custom_dot(eigenvectors.T, centered_data)
153     return eigenfaces, eigenvalues[:num_eigenfaces]
154
155 def display_eigenfaces(eigenfaces, num_eigenfaces, image_shape
        =(100,100), results_subdirectory=None):
156     # Calculate the grid size dynamically
157     grid_cols = int(np.ceil(np.sqrt(num_eigenfaces)))
158     grid_rows = int(np.ceil(np.sqrt(num_eigenfaces)))
159
160     # Create a blank canvas to display the eigenfaces
161     grid_height = grid_rows * image_shape[0]
162     grid_width = grid_cols * image_shape[1]
163     canvas = np.zeros((grid_height, grid_width), dtype=np.uint8)
164
165     for i in range(num_eigenfaces):
166         row = i // grid_cols
167         col = i % grid_cols
168         eigenface = eigenfaces[i].reshape(image_shape)
169         eigenface = normalize(eigenface, norm_type=cv2.
            NORM_MINMAX)
170         eigenface = eigenface.astype(np.uint8)
171         canvas[row * image_shape[0]:(row + 1) * image_shape[0],
            col * image_shape[1]:(col + 1) * image_shape[1]] =
            eigenface
172
173     # Display the canvas with all eigenfaces
174     cv2.imshow("Eigenfaces", canvas)
175     cv2.waitKey(0)

```

```

176     cv2.destroyAllWindows()
177
178     # Save the canvas with all eigenfaces
179     if results_subdirectory:
180         eigenfaces_path = os.path.join(results_subdirectory, "
181             eigenfaces.jpg")
182         cv2.imwrite(eigenfaces_path, canvas)
183
184 # Step 7: Project Faces Onto Eigenfaces
185 def project_faces(centered_data, eigenfaces):
186     return custom_dot(centered_data, eigenfaces.T)
187
188 def save_results(results, filename):
189     with open(filename, 'w') as f:
190         for result in results:
191             f.write(f"{result}\n")
192
193 # Step 8: Recognize Test Face
194 def recognize_face(test_face, mean_face, eigenfaces,
195     projected_faces, labels, original_faces, fixed_threshold=300):
196     centered_test_face = test_face - mean_face
197     centered_test_face = centered_test_face.reshape(1, -1)
198     projected_test_face = custom_dot(centered_test_face,
199         eigenfaces.T)
200     distances = np.linalg.norm(projected_faces -
201         projected_test_face, axis=1)
202     min_distance = np.min(distances)
203     recognized_label = labels[np.argmin(distances)]
204     closest_face_index = np.argmin(distances)
205     threshold = np.percentile(distances, 10)
206     # Use a fixed threshold
207     # threshold = fixed_threshold
208
209     # Debugging: Print distances and threshold
210     print(f"Threshold: {threshold}")
211     print(f"Min distance: {min_distance}")
212
213     if min_distance > threshold:
214         recognized_label = "unknown"
215     else:
216         recognized_label = labels[closest_face_index]
217     # Reconstruct the projected face
218     # reconstructed_face = np.dot(projected_test_face, eigenfaces
219         ) + mean_face
220     reconstructed_face = custom_dot(projected_test_face,
221         eigenfaces) + mean_face
222
223     # Display the test face, mean face, closest face, and
224     # reconstructed face side by side
225     test_face_image = test_face.reshape((100,100))
226     mean_face_image = mean_face.reshape((100,100))

```

```

220     closest_face = original_faces[closest_face_index].reshape
        ((100,100))
221     reconstructed_face_image = reconstructed_face.reshape
        ((100,100))
222
223     # Normalize the images to the range [0, 255]
224     test_face_image = normalize(test_face_image, norm_type=cv2.
        NORM_MINMAX)
225     mean_face_image = normalize(mean_face_image, norm_type=cv2.
        NORM_MINMAX)
226     closest_face = normalize(closest_face, norm_type=cv2.
        NORM_MINMAX)
227     reconstructed_face_image = normalize(reconstructed_face_image
        , norm_type=cv2.NORM_MINMAX)
228
229     # Convert to uint8 type
230     test_face_image = test_face_image.astype(np.uint8)
231     mean_face_image = mean_face_image.astype(np.uint8)
232     closest_face = closest_face.astype(np.uint8)
233     reconstructed_face_image = reconstructed_face_image.astype(np
        .uint8)
234
235     # Resize images to be larger for better visibility
236     test_face_image = resize(test_face_image, (200, 200))
237     mean_face_image = resize(mean_face_image, (200, 200))
238     closest_face = resize(closest_face, (200, 200))
239     reconstructed_face_image = resize(reconstructed_face_image,
        (200, 200))
240
241     # Concatenate images horizontally
242     combined_image = np.hstack((test_face_image, mean_face_image,
        closest_face, reconstructed_face_image))
243
244     # Add labels to the combined image
245     labels = ["Test Face", "Mean Face", "Closest Face", "
        Reconstructed Face"]
246     positions = [(10, 20), (210, 20), (410, 20), (610, 20)]
247     add_labels_to_image(combined_image, labels, positions)
248
249     # Save the result
250     return recognized_label , combined_image
251
252 # Paths
253 # olivetti_faces_dir = "olivetti_faces" # Directory with
    Olivetti faces
254 olivetti_faces_dir = "cropped_faces"
255
256 # Step 1: Load and Resize Olivetti Faces
257 image_paths, labels = load_faces(olivetti_faces_dir)
258
259 # Step 2: Preprocess Dataset

```

```

260 dataset = preprocess_images(image_paths)
261
262 # Split the dataset into training and test sets
263 train_paths, test_paths, train_labels, test_labels =
    train_test_split(image_paths, labels, test_size=0.2)
264
265 # Preprocess training and test sets
266 train_dataset = preprocess_images(train_paths)
267 test_dataset = preprocess_images(test_paths)
268
269 # Step 3: Compute Mean Face
270 mean_face, mean_face_image = compute_mean_face(train_dataset)
271
272 # Step 4: Center the Dataset
273 centered_train_data = center_dataset(train_dataset, mean_face)
274
275 # Step 5: Compute Covariance Matrix
276 covariance_matrix = compute_covariance_matrix(centered_train_data
    )
277 print("Covariance matrix is calculated ")
278 # Step 6: Compute Eigenfaces
279 num_eigenfaces = min(10, train_dataset.shape[0])
280 eigenfaces, eigenvalues = compute_eigenfaces(centered_train_data,
    covariance_matrix, num_eigenfaces)
281
282 # Create a subdirectory for this run's results
283 results_subdirectory = create_subdirectories()
284
285 # Save mean face
286 mean_face_path = os.path.join(results_subdirectory, "mean_face.
    jpg")
287 cv2.imwrite(mean_face_path, mean_face_image)
288
289 # Display and save eigenfaces
290 display_eigenfaces(eigenfaces, num_eigenfaces,
    results_subdirectory=results_subdirectory)
291
292 # Step 7: Project Faces Onto Eigenfaces
293 projected_train_faces = project_faces(centered_train_data,
    eigenfaces)
294
295 # Test the recognition on the test set
296 correct_predictions = 0
297 unknown = 0
298 count = 1
299 results = []
300 for test_image_path, true_label in tqdm(zip(test_paths,
    test_labels), desc="Recognizing faces", total=len(test_paths))
    :
301     test_face_dataset = preprocess_images([test_image_path])
302     recognized_label, combined_image = recognize_face(

```

```

303     test_face_dataset[0],
304     mean_face,
305     eigenfaces,
306     projected_train_faces,
307     train_labels,
308     train_dataset,
309     fixed_threshold=300 # Set a fixed threshold
310 )
311 if recognized_label == "unknown":
312     print(f"Unknown face: {test_image_path}")
313     result = f"Test image: {true_label}, True label: {
314         true_label}, Recognized label: unknown , UNSUCCESSFUL"
315     output_path = os.path.join(results_subdirectory, f"{count
316         }_{true_label}_recognized_as_unknown_result.jpg")
317     unknown += 1
318 elif recognized_label == true_label:
319     print(f"TRUE")
320     result = f"Test image: {true_label}, True label: {
321         true_label}, Recognized label: {recognized_label} ,
322         SUCCESSFUL"
323     output_path = os.path.join(results_subdirectory, f"{count
324         }_{true_label}_recognized_as{true_label}_result.jpg")
325     correct_predictions += 1
326 else:
327     print(f"True label: {true_label}, Recognized as: {
328         recognized_label}")
329     result = f"Test image: {true_label}, True label: {
330         true_label}, Recognized label: {recognized_label} ,
331         UNSUCCESSFUL"
332     output_path = os.path.join(results_subdirectory, f"{count
333         }_{true_label}_recognized_as{recognized_label}_result.
334         jpg")
335     count += 1
336 # Save the result
337 cv2.imwrite(output_path, combined_image)
338 results.append(result)
339
340 # Save all results to a file
341
342 # Calculate and print the recognition accuracy
343 accuracy = correct_predictions / len(test_paths)
344 print(f"Recognition accuracy: {accuracy * 100:.2f}%")
345 print(f"Unknown recognition: {unknown} out of {len(test_paths)}")
346 results.append(f"Recognition accuracy: {accuracy * 100:.2f}%")
347 results.append(f"Unknown recognition: {unknown} out of {len(
348     test_paths)}")
349 results_file = os.path.join(results_subdirectory, "results.txt")
350 save_results(results, results_file)

```