Report of Task 5

Introduction

This report will explore the fascinating field of facial recognition technology. Our study will focus on the application of machine learning techniques to identify and recognise human faces from standard face datasets.

We will employ methods to detect faces in the AT&T Database of Faces from https://www.kaggle.com/datasets/kasikrit/att-database-of-faces and utilise Principal Component Analysis (PCA) and Eigen analysis for face recognition. The performance of our approach will be evaluated and presented through the construction of a Receiver Operating Characteristic (ROC) curve. This report aims to provide a comprehensive understanding of the practical application of these techniques in the realm of artificial intelligence and computer vision.

I. Face Detection

Cascade classifiers are a type of **supervised machine learning algorithm** used for binary classification tasks. They are based on the **AdaBoost algorithm** and use a series of weak classifiers in a cascade.

The term "cascade" in this context means that the proposed system consists of several simple classifiers ("weak classifiers") arranged in a cascade, which allows negative input (non-face) to be quickly discarded while spending more computation on promising or positive face-like regions. This significantly reduces the computation time and makes the process more efficient.

Cascade classifiers are most commonly used for **face detection** in images and videos but can be applied to other tasks as well. They are trained with several hundred "positive" sample views of a particular object and arbitrary "negative" images of the same size. After the classifier is trained it can be applied to a region of an image and detect the object in question.

To have good overall performance, each stage of the cascade classifier must validate all faces and can produce many false positives. For example, if stage 1 were to mark as 'does not contain a face' 20% of rectangles containing a face (false negative rate=20%), then the

total performance of the chain cannot be higher than 80% true positive, whatever the next stages are, since 20% of faces have already been rejected. (i.e. if this stage incorrectly classifies 20% of faces as non-faces, these are known as false negatives. This means that 20% of actual faces are being incorrectly rejected at the very first stage. As a result, even if all the following stages were perfect and made no mistakes at all, the best accuracy you could achieve would be an 80% true positive rate. This is because 20% of faces have already been incorrectly rejected at the first stage and hence, are not even considered by the subsequent stages.)

This is why each stage of the cascade classifier needs to have a high detection rate (low false negative rate), as any faces incorrectly rejected at an early stage are lost for the remainder of the detection process.

Haar cascade is a machine learning object detection algorithm proposed by Paul Viola and Michael Jones in their 2001 paper, "Rapid Object Detection using a Boosted Cascade of Simple Features". It's used to detect objects in images or videos.

1. **Haar Features**: Haar features are extracted from rectangular areas in an image. The feature's value is based on the pixel intensities. It's calculated using a sliding window, and the area within the window is partitioned into two or more rectangular areas. The Haar feature is the difference in the sum of pixel intensities between these areas. It is believed that an object's presence will distort the variation of pixel intensity.

2. **Integral Images**: To speed up the feature calculation, an integral image is used. This allows for the sum of all pixels in a given rectangle to be calculated with only four operations. This makes things super-fast.

Integral images allow for the rapid calculation of the sum of pixel values in any rectangular region of the image. This can be done in constant time, regardless of the size of the region.

Given a rectangle with top-left corner at (x1, y1) and bottom-right corner at (x2, y2), the sum of pixel intensities in this rectangle can be computed as follows:

Sum = Integral_Image[x2, y2] - Integral_Image[x1, y2] - Integral_Image[x2, y1] + Integral_Image[x1, y1]

This formula works because each pixel in the integral image represents the cumulative sum of the pixels above and to the left of it in the original image. Therefore, the value at (x2, y2) in the integral image is the sum of all pixel values in the rectangle from (0, 0) to (x2, y2) in the original image. Similarly, the values at (x1, y2), (x2, y1), and (x1, y1) in the integral image represent the sum of all pixel values in the rectangles from (0, 0) to (x1, y2), (0, 0) to (x2, y1), and (0, 0) to (x1, y1) in the original image, respectively.

By subtracting the values at (x1, y2) and (x2, y1) from the value at (x2, y2), we remove the pixel values that are counted twice. However, this also removes the pixel values in the

rectangle from (0, 0) to (x1, y1) in the original image, so we add back the value at (x1, y1) in the integral image.

3. **AdaBoost**: Among all the features calculated, most of them are irrelevant. AdaBoost is used to select the best features and train the classifier. For each feature calculation, we need to find the sum of the pixels under white and black rectangles. To solve this, they introduced the integral image. “white” and “black” rectangles refer to the regions of an image that are being compared.

A Haar feature is essentially a specific pattern of “white” and “black” rectangles. The “white” rectangles represent areas of the image that we expect to be lighter, while the “black” rectangles represent areas that we expect to be darker.

For example, in face detection, one common Haar feature is a set of two adjacent rectangles that lie above the eye region and the bridge of the nose. The rectangle over the eyes is “black” (as the eye region is usually darker) and the rectangle over the bridge of the nose is “white” (as this region is usually lighter).

The value of a Haar feature is calculated as the difference between the sum of the pixel intensities in the “white” rectangles and the sum of the pixel intensities in the “black” rectangles. This is where integral images come in - they allow this sum to be computed very quickly, in constant time

4. **Cascade of Classifiers**: The word “cascade” in the classifier name means that the resultant classifier consists of several simpler classifiers (stages) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed.

The `cv2.CascadeClassifier` is a class in the OpenCV library that is used for object detection. It is based on the Haar cascade object detection method.

In the context of our code, `cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')` is loading a pre-trained Haar cascade classifier for frontal face detection from an XML file.

The `detect_face` function is a method in a class that is designed to detect faces in an image. It uses the Haar cascade classifier, a machine-learning object detection algorithm used to identify objects in an image or video.

1. **Function Definition**: The function is defined with the name `detect_face` and it doesn't take any parameters apart from the `self` keyword, which refers to the instance of the class where this method is defined.

2. **Grayscale to RGB Conversion**: The function begins by converting the current image from grayscale to RGB using the `cv2.cvtColor` function. This is necessary because the Haar cascade classifier requires a grayscale image for face detection, but the rectangles to be drawn on the faces will be in colour (RGB).

3. **Loading the Cascade Classifier**: The function then loads the Haar cascade classifier for frontal face detection using the `cv2.CascadeClassifier` function. The classifier is trained on thousands of positive images (with faces) and negative images (without faces) to recognize faces.

4. **Face Detection**: The function detects faces in the grayscale image using the `detectMultiScale` method of the cascade classifier.

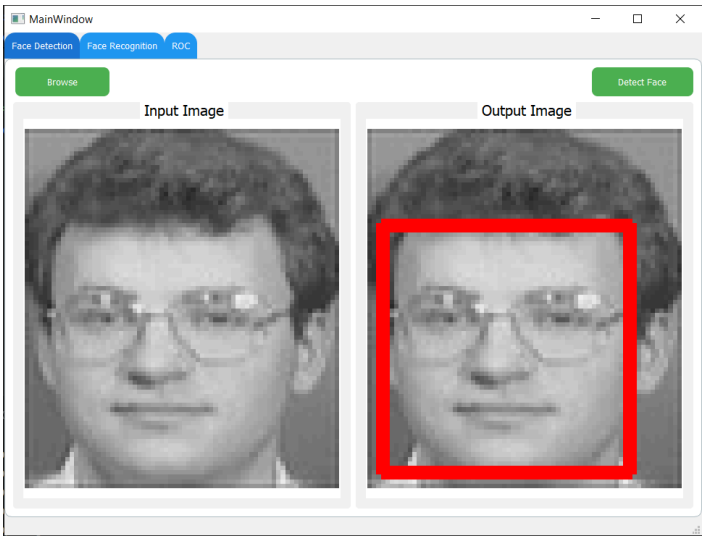
The `detectMultiScale` method is used to detect objects in an image. This method takes in an image and some optional parameters that control the scale factor and minNeighbors. It returns a list of rectangles where it believes it found the object. The parameters `1.1` and `4` are the scale factor and minNeighbors respectively.

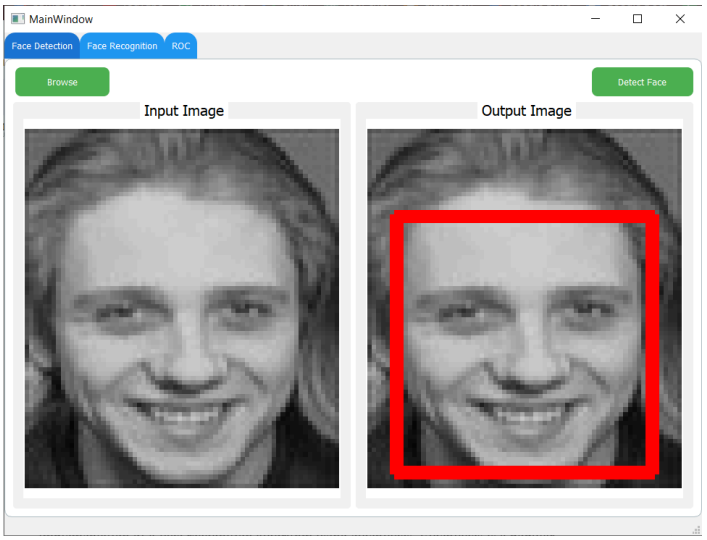
The scale factor parameter specifies how much the image size is reduced at each image scale. This parameter is used to create a scale pyramid, which is a series of scaled images. The scale factor is used to control the size of the sliding window during the scanning process. A smaller scale factor, like 1.05, means that the sliding window is resized in small steps, increasing the chance of a matching size with the model for detection, but also slowing down the detection process. A larger scale factor speeds up the detection process, with the risk of missing some faces.

The minNeighbors parameter specifies how many neighbours each candidate rectangle should have to retain it. This parameter will affect the quality of the detected faces. Higher values result in fewer detections but with higher quality. The idea behind this parameter is that the detector will run on a multiple-scale style and at the same time follow a sliding window strategy. After this step, it will give you multiple responses even for a single face region. So, this parameter helps in removing false positives by ensuring that a candidate face region should have a certain number of neighbours (other candidate regions that are similar to it) to be considered valid face detection.

5. **Drawing Rectangles Around Faces**: For each detected face, the function draws a rectangle around the face using the `cv2.rectangle` function. The parameters `(x, y)` are the coordinates of the top-left corner of the rectangle, `(x+w, y+h)` are the coordinates of the bottom-right corner, `(255, 0, 0)` is the colour of the rectangle (red in this case), and `2` is the thickness of the rectangle lines.

6. **Return Value**: The function does not return any value. Instead, it modifies the current image by drawing rectangles around the detected faces.





II. Face Recognition

Face recognition is a crucial task in computer vision with various applications, including security systems, human-computer interaction, and biometrics. We present an implementation of a face recognition algorithm using eigenfaces. Eigenfaces is a popular technique for face recognition that utilizes principal component analysis (PCA) to reduce the dimensionality of face images and represent them as a linear combination of eigenfaces.

Algorithm Steps

1. Initialization

   FaceRecognition Class Initialization:

   A FaceRecognition class is created to encapsulate methods for face recognition tasks.

   The class constructor initializes attributes such as files_list, dir_to_input, face_matrix, and eigen_faces.

2. Data Preprocessing

   - Retrieving File List:

     Method: get_files_list()

     Retrieves all files in the specified directory and stores them into a list (files_list).

   - Creating Face Matrix:

Method: create_face_matrix()

Reads each image file from the file list, preprocesses them by resizing to a fixed size, and reshaping into vectors.

The resulting vectors are stored in a matrix (face_matrix).

3. Feature Extraction

- Calculating Mean Matrix:

  Method: get_mean_matrix()

  Calculates the mean of the face matrix.

  Subtracts the mean from all samples, resulting in a mean array.

- Computing Covariance Matrix and Eigenfaces:

  Method: get_covariance_matrix()

  Computes the covariance matrix of the mean array.

  Calculates eigenvalues and eigenvectors of the covariance matrix.

  Sorts the eigenvectors in descending order of eigenvalues.

  Normalizes the eigenvectors to obtain eigenfaces.

4. Face Detection

Detecting Faces in Images:

Method: detect_face(img_path)

Reads an image file, and preprocesses it by resizing and subtracting the mean sample.

Projects the preprocessed image onto the eigenfaces space.

Detects faces based on a predefined threshold by comparing the distance between the projected image vector and the original face images.

5. Face Recognition

Recognizing Faces:

Method: face_recognition()

Compares the input image with the dataset of known faces using the Euclidean distance metric.

Identifies the closest match and determines if the face is recognized based on a threshold.

Comparing Distance to Threshold:

If the distance is below the threshold value (threshold), it indicates that the input image closely matches a known face, and the algorithm recognizes the face.

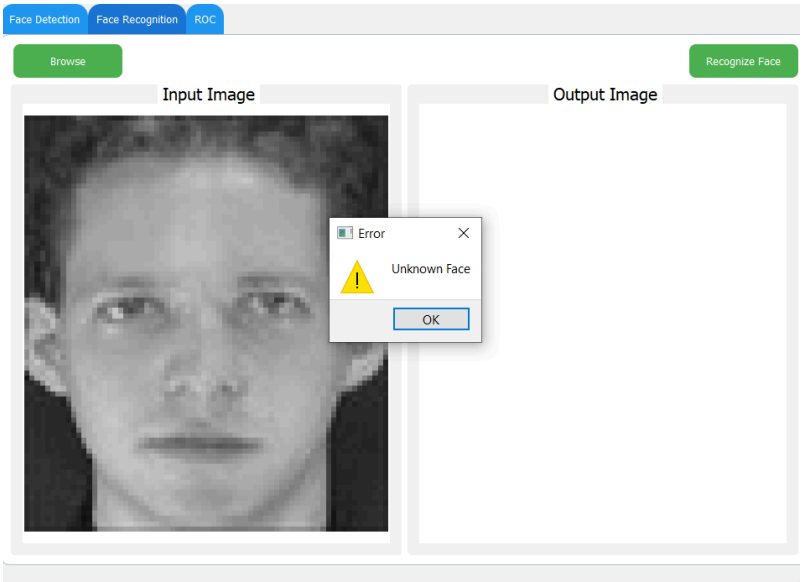
If the distance exceeds the threshold, it suggests that the input image does not closely resemble any known face in the dataset, and the algorithm labels it as an "unknown face".

Face Recognition: Effect of Threshold Value

Threshold Comparison

When applying the face recognition algorithm to the same input image, three different threshold values were used: 2000, 3000 and 10000. The results obtained from these threshold values differed in terms of recognizing the input image.

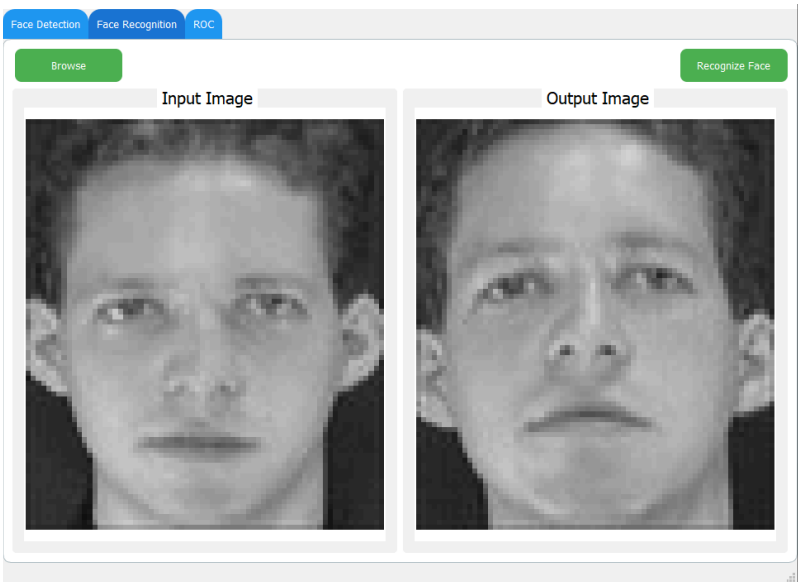
- **Threshold 2000:**



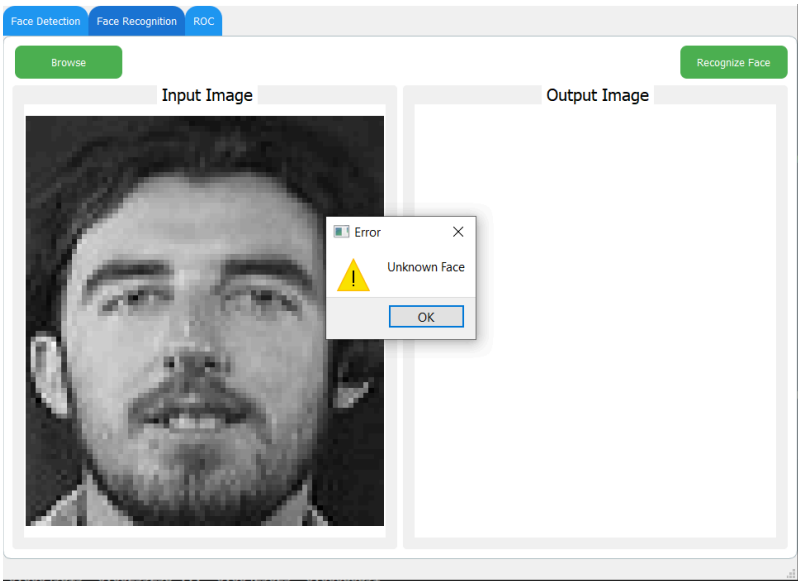
  - **Observation:** Despite using a lower threshold value, the algorithm failed to recognize the input image as a known face.

- **Justification:** A lower threshold value makes the algorithm less likely to recognize known faces. However, it also decreases the risk of false positives.

- **Threshold 3000:**

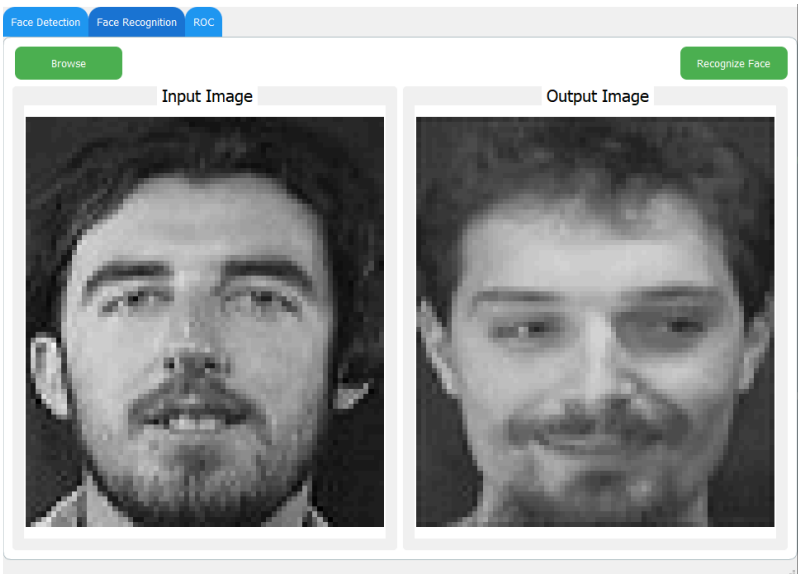


- **Observation:** By increasing the threshold value, the algorithm successfully recognized the input image as a known face. This suggests that a larger threshold is needed in this case to cover the lowest distance
- **Justification:** A higher threshold value increases the likelihood of correctly recognizing unknown faces as known ones. On the other hand, it may increase the risk of false positive recognition



- **Observation:** But in this case, this image does not already exist in the training dataset so, it makes sense to be predicted as a true negative.

- **Threshold 10000:**



- **Observation:** A much higher threshold value leads to a false positive response illustrating the criticality of choosing a suitable threshold value
- **Justification:** A very high threshold value increases the likelihood of incorrectly recognizing unknown faces as known ones.

III. ROC

The Receiver Operating Characteristic (ROC) curve is a graphical representation of a binary classifier's performance across different threshold settings. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold values.TPR, also known as sensitivity, measures the proportion of positive cases correctly identified as positive. FPR, also known as fall-out, measures the proportion of negative cases incorrectly identified as positive.

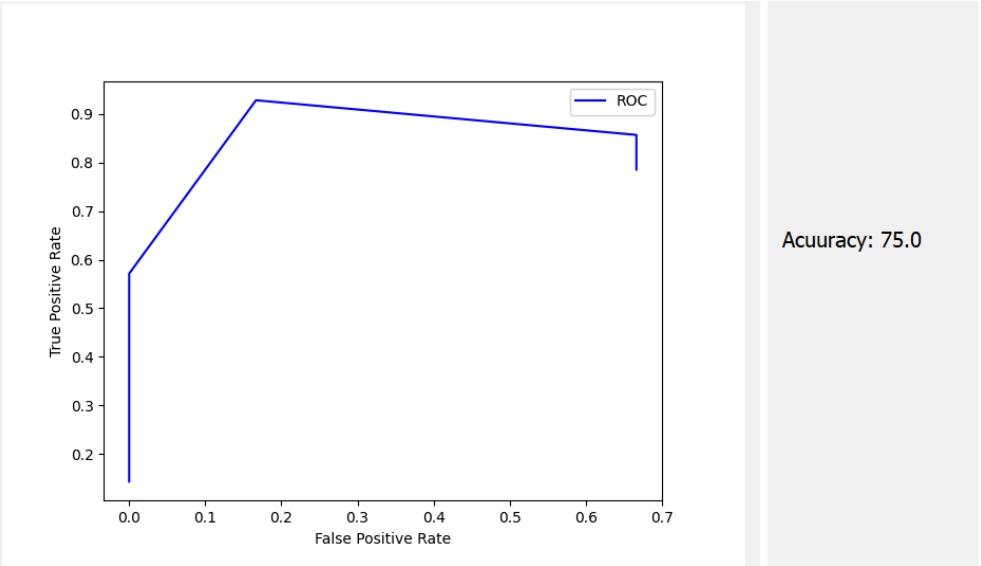
In our case of face recognition:

- True Positive Rate (TPR): The proportion of known faces correctly identified as known.
- False Positive Rate (FPR): The proportion of unknown faces incorrectly classified as known.

The ROC curve is drawn based on a set of predefined threshold points, specifically [0.01, 0.02, 0.05, 0.5, 0.9]. These threshold values represent five operating points at which the algorithm's performance is evaluated.

Effect of Threshold Value of Face Recognition

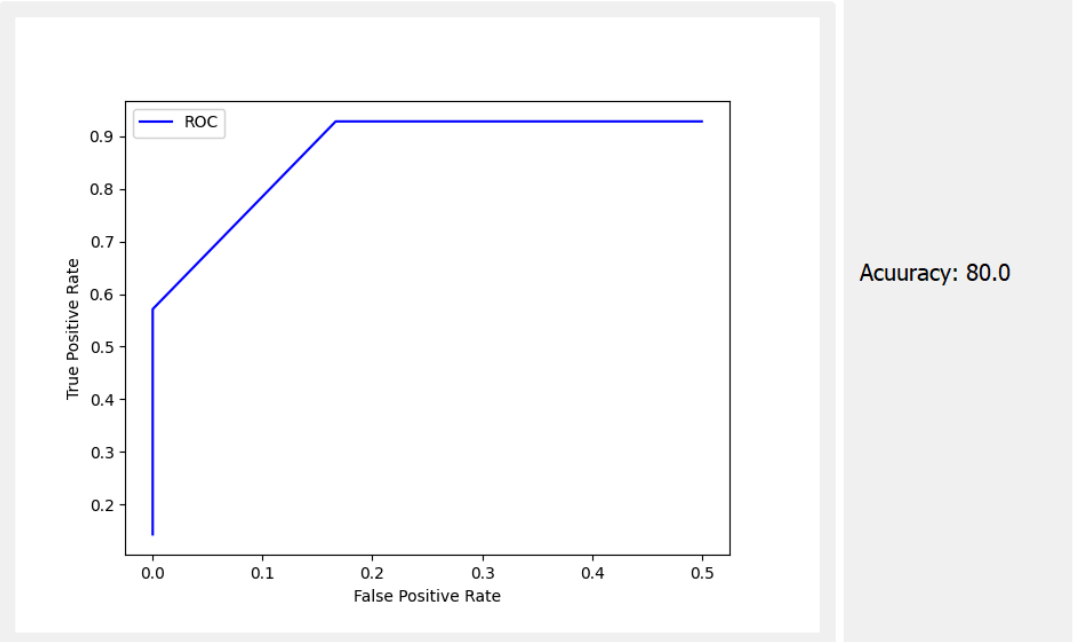
- **Threshold 2000**



  - **Observation:** At threshold 2000, the ROC curve shows a low True Positive Rate (TPR) of approximately 0.6667. This indicates that the proportion of known faces correctly identified as known would be low. Consequently, the False Positive Rate (FPR) is also expected to be low, approximately 0.7857, indicating a minimal number of false positives.

- **Justification:** The lower threshold value of 2000 led to a reduced likelihood of recognizing known faces, resulting in a low TPR of approximately 0.6667. While this decreases the risk of false positives, it also compromises the algorithm's ability to correctly classify known faces. Therefore, both TPR and FPR are expected to be low, indicative of the algorithm's limited discrimination ability at this threshold.

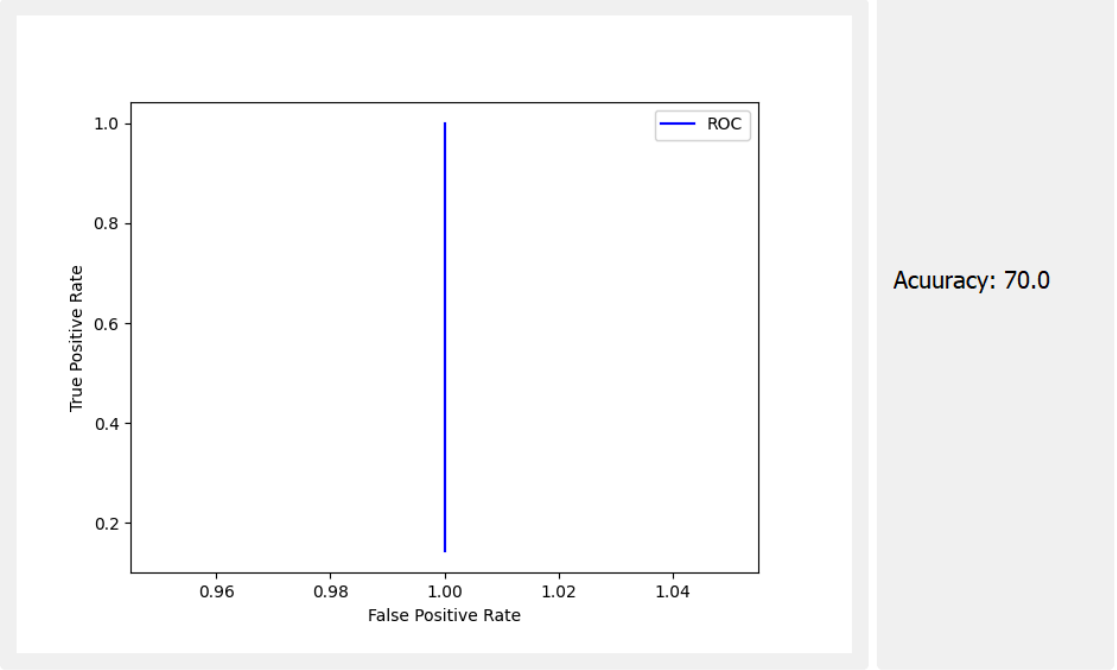
- **Threshold 3000**



- **Observation:** At threshold 3000, the ROC curve shows an improved True Positive Rate (TPR) of approximately 0.929. This indicates that a larger proportion of known faces are correctly identified as known compared to the lower threshold. However, the False Positive Rate (FPR) is relatively high at approximately 0.5, suggesting a significant number of false positives.

- **Justification:** By increasing the threshold value to 3000, the algorithm successfully recognized the input image as a known face, resulting in an improved TPR. This suggests that a higher threshold is needed to cover the lowest distance for accurate recognition. However, the higher threshold also increases the risk of false positive recognition, as indicated by the relatively high FPR. Therefore, while the algorithm performs better in correctly identifying known faces, it also exhibits a higher rate of false positives at this threshold.

- **Threshold 10000**



  - **Observation:** At threshold 10000, the ROC curve shows a perfect True Positive Rate (TPR) of 1.0, indicating that all known faces are correctly identified as known. However, the False Positive Rate (FPR) is also 1.0, suggesting that all unknown faces are incorrectly identified as known.

  - **Justification:** The much higher threshold value of 10000 led to a perfect TPR, as all known faces were correctly recognized. However, setting the threshold this high also resulted in a 1.0 FPR, meaning that all unknown faces were erroneously classified as known. While achieving perfect recognition of known faces, this threshold setting proves impractical due to the extremely high rate of false positive identifications.