Name: Nguyen Minh Trang

ID: 411021365

# Image Processing Final Project

# Code Explanation

### Import libraries and get the current working directory for later use

```
import os
import cv2
import numpy as np
import csv

HOME = os.getcwd()
```

Essential libraries, including OS, OpenCV (‘cv2’), NumPy, and CSV are imported for directory management, image processing, mathematical operations, and file handling.

The global variable ‘HOME’ stores the working directory’s path, used later for file management.

### Euclidean Distance Function

```
def euclidean_distance (vector1, vector2):
  sum = 0.0
  for i in range(len(vector1)):
    sum += (vector1[i] - vector2[i]) ** 2
  return np.sqrt(sum)
```

This function calculates the Euclidean distance between two vectors, this function will later be used for comparing facial feature vectors to measure the similarity between two faces.

The code for calculation is written based on this formula:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}.$$

### Instance class

```
# Define class 'instance' to represent one face in the database
class instance:
  # Each 'instance' object is initialize with two attributes: name and landmark
  # Name is the name of the person that the instance belongs to
  # Landmark is the landmark data corresponding to that instance in the dataset
  def __init__(self, name, landmark):
```

```
    self.name = name
    self.landmark = landmark

  # Function to create a feature vector from facial landmarks
  # A number of distances between some pairs of landmark points are calculated,
normalized by the width of the left eye
  # All of the calculated distances along with the flattened landmark points are
combined into one feature vector to represent the facial structure
  def get_feature_vector(self):
    landmark = self.landmark

    # Reference distance: width of the left eye (landmark 36 to 39)
    eye_width = euclidean_distance(landmark[36], landmark[39])

    # Mouth area feature extraction
    mouth_width = euclidean_distance(landmark[48], landmark[54]) / eye_width
    mouth_height = euclidean_distance(landmark[51], landmark[57]) / eye_width
    upper_lip_height = euclidean_distance(landmark[51], landmark[62]) / eye_width
    lower_lip_height = euclidean_distance(landmark[57], landmark[66]) / eye_width
    upper_lip_reflective_distances = [euclidean_distance(landmark[i],
landmark[108 - i]) / eye_width for i in range(49, 54)]
    lower_lip_reflective_distances = [euclidean_distance(landmark[i],
landmark[128 - i]) / eye_width for i in range(61, 64)]


    # Jawline area feature extraction
    jawline_width = euclidean_distance(landmark[0], landmark[16]) / eye_width
    jaw_to_mouth = euclidean_distance(landmark[8], landmark[57]) / eye_width
    jaw_to_nose = euclidean_distance(landmark[8], landmark[30]) / eye_width
    jaw_to_lowest_jaw = [euclidean_distance(landmark[i], landmark[8]) / eye_width
for i in range(1, 8)]

    # Combine all the features into one feature vector
    feature_vector = np.array([
        mouth_width,
        upper_lip_height, lower_lip_height, jawline_width,
        jaw_to_mouth, jaw_to_nose
    ] + jaw_to_lowest_jaw + [
        mouth_height
    ] + upper_lip_reflective_distances + lower_lip_reflective_distances)

    feature_vector *= 100

    # Include the facial landmarks in the feature vector
```

```
    # The jawline landmark points are not included because the location of the
jawline keypoints is the most sensitive to the variation of face position
    for i in range(17, len(landmark)):
      feature_vector = np.concatenate([feature_vector, landmark[i]])
    return feature_vector

  # Function to compare the face of this instance with that of another instance
  # Calculate Euclidean distance between the two feature vectors
  # The distance is returned as a measure of similarity
  def compare_face(self, B):
    vectorA = self.get_feature_vector()
    vectorB = B.get_feature_vector()
    return euclidean_distance(vectorA, vectorB)

  # Function to check if two instances represent the same person
  def is_the_same_person(self, B):
    return self.name == B.name
```

This class is used to represent an individual face in both the database and the test set.

1. The **function '__init__'** is used to initialize an 'instance' object with two attributes: 'name' and 'landmark', with 'name' being the name of the person that this instance belongs to, and 'landmark' being the feature landmarks of the face image corresponding to this instance.
2. The **function 'get_feature_vector'** is used to extract a feature vector from the facial landmarks of this instance, a numerical representation of the image itself. A number of distances between some pairs of the feature landmarks are calculated and normalized by the width of the left eye, which are then combined together along with the flattened feature landmarks to form a feature vector. The landmark points that represent the jawline is not included directly into the feature vector because it seems to be the most sensitive to the change in face position and the person's pose in the image (the same person can have two different looking jawlines if they change the angle and position of the face in the photo). Instead, the jawline landmark points are utilized to calculate the distances of landmarks within the face, then normalized by the left eye's width, offering a more stable feature representation.
3. The **function 'compare_face'** is used to compare the face of this instance with that of another instance (B). The feature vector of each instance is calculated, then they are compared using the Euclidean distance method to measure the similarity of the two faces (the smaller the distance is, the more the faces look alike).
4. The **function 'is_the_same_person'** is used to verify if this instance and instance B both belong to the same person by simply comparing the names of the instances.

**Database class**

```
# Define the 'database' class to handle a collection of instances
class database:

  # Each 'database' object is initialized with two attributes: people and
threshold
  # People is the list of instances in the database
  # Threshold is the acceptable distance that a face instance can have with at
least one instance in the database in order for it to be authrorized by the
system
  def __init__(self, threshold, people):
    self.people = people
    self.threshold = threshold

  # Function to add a new person to the database
  def add_person(self, A):
    self.people.append(A)

  # Function to match a face (B) with the faces in the database
  # The function returns 3 values: 'is_in_DB', 'is_auth' and 'authorize_person'
  # 'is_in_DB' is a boolean value indicating whether the face actually belongs to
a person that appears in the database
  # 'is_auth' is a boolean value indicating whether the system recognizes the
face
  # 'authorize_person' is the name of the person that the system matches with the
given face
  def face_matching(self, B):
    people = self.people
    is_in_DB = False
    is_auth = False
    min_dist = self.threshold
    authorize_person = ''
    for person in people:
      if person.is_the_same_person(B):
        is_in_DB = True
      dist = person.compare_face(B)
      if dist <= min_dist:
        is_auth = True
        min_dist = dist
        authorize_person = person.name

    return is_in_DB, is_auth, authorize_person
```

This class is designed to manage a collection of face instances and handle the authentication process.

1. The **function '__init__'** defines how a 'database' object is created. When a 'database' object is created, it is initialized with a 'threshold' and a list of 'people'. The 'threshold' determines the sensitivity of the face matching process, and 'people' is a list of 'instance' objects that represent the faces stored in the database.
2. The **function 'add_person'** allows new 'instance' objects to be added to the database. It's crucial for expanding the database with new faces, making the system adaptable and scalable.
3. The **function 'face_matching'** compares a given face (represented by an 'instance' object B) against all faces in the database. The comparison relies on the feature vectors of each face and uses the Euclidean distance to measure similarity. If the distance between B and any face in the database is less than or equal to the 'threshold', the face is considered authorized. If B is authorized, the face in the database that has the closest distance to B will be a match with B.

**Landmark extraction and face alignment**

1. Landmark extraction:

```
# Function to extract the facial landmarks from a given CSV file
# The data from CSV is read and converted into a usable format
# The landmark data is returned as a numpy array
def get_landmark (landmark_path):
  landmark = []
  with open(landmark_path, 'r') as file:
    reader = csv.reader(file)
    next(reader, None)
    for row in reader:
      x=int(float(row[1]))
      y=int(float(row[2]))
      landmark.append((x, y))
  return np.array(landmark)
```

   This function reads facial landmark data from a CSV file and converts it into a format usable by the system. It opens the specified landmark file, reads each line using 'csv.reader', and extracts the x and y coordinates of each landmark point. These coordinates are appended to a list, which is then converted to a NumPy array.
2. Face alignment

```
# Function to align a face in an image based on the eye area landmarks
def align (image, landmark):

  # Set the desired dimensions and left_eye position for the aligned face
  # After alignment, the left eye center is expected to be at the pixel
(desired_height*0.35, desired_width*0.35)
```

```
  desired_width = image.shape[0]
  desired_height = image.shape[1]
  desired_left_eye_position = (0.35, 0.35)

  # Extract the left eye and right eye landmark points from the landmark set
  left_eye_points = landmark[36:42]
  right_eye_points = landmark[42:48]

  # Calculate the center point of the left and right eyes
  left_eye_center = (np.mean(left_eye_points[:, 0], axis=0).astype(int),
np.mean(left_eye_points[:, 1], axis=0).astype(int))
  right_eye_center = (np.mean(right_eye_points[:, 0], axis=0).astype(int),
np.mean(right_eye_points[:, 1], axis=0).astype(int))

  # Compute the angle between 2 lines:
  # The first line is the line that passes through both eye centers
  # The second is the horizontal line
  dY = right_eye_center[1] - left_eye_center[1]
  dX = right_eye_center[0] - left_eye_center[0]
  angle = np.degrees(np.arctan2(dY, dX))

  # Calculate the desired X position for the right eye center
  desired_right_eye_X = 1.0 - desired_left_eye_position[0]

  # Calculate the scale factor based on the current and desired distances
  current_dist = np.sqrt((dX**2) + (dY**2))
  desired_dist = (desired_right_eye_X - desired_left_eye_position[0]) *
desired_width
  scale = desired_dist / current_dist

  # Calculate the center point between the eyes
  eye_center = ((left_eye_center[0] + right_eye_center[0])//2,
(left_eye_center[1] + right_eye_center[1])//2)
  eye_center = (int(eye_center[0]), int(eye_center[1]))

  # Get the rotation matrix for aligning the image
  # The image is supposed to be rotated around the eye_center, with the
calculated angle and scale factors
  M = cv2.getRotationMatrix2D(eye_center, angle, scale)

  # Update the translation component of the matrix to make sure that the
center of the eye is in a fixed position in every aligned image
  tX = desired_width * 0.5
  tY = desired_height * desired_left_eye_position[1]
  M[0, 2] += (tX - eye_center[0])
```

```
  M[1, 2] += (tY - eye_center[1])

  # Apply the affine transformation to the image to align it
  output = cv2.warpAffine(image, M, (desired_width, desired_height),
flags=cv2.INTER_CUBIC)

  # Adjust the landmark according to the transformation
  aligned_landmark = []
  i=0
  for point in landmark:
    i+=1
    point_mul = [point[0], point[1], 1]
    rotated_point = np.matmul(M, point_mul)
    aligned_landmark.append([i, int(rotated_point[0]),
int(rotated_point[1])])

  # Return the aligned image and the adjusted landmarks
  return output, aligned_landmark
```

This function is responsible for adjusting each face in the dataset (including both the database and the test set) so that they have a standard orientation and size. This is done based on the eye landmarks. The function calculates the position of the eyes in the image and rotates, scales, and translates the image so that the eyes are aligned horizontally and positioned consistently in every processed image. By aligning the images, it is ensured that all faces are presented to the system in a uniform manner, which is important for accurate feature extraction and comparison.

**Image and landmark processing in database directories**

```
# Process images and landmark from the given database directory
# Read, aligned and preprocess images from the database
# Save the aligned version into the databased
data_path=os.path.join(HOME, 'IP_Database')

# Loop through both 'Face_DB' and 'Test_DB' folders in the database
for folder in ('Face_DB', 'Test_DB'):
  folder_path = os.path.join(data_path, folder)
  image_folder_path = os.path.join(folder_path, 'Images')

  # Create directories for storing aligned images and features if they haven't
already existed
  aligned_image_folder = os.path.join(folder_path, 'Aligned_Images')
  aligned_feature_folder = os.path.join(folder_path, 'Aligned_Features')
  os.makedirs(aligned_image_folder, exist_ok=True)
```

```
  os.makedirs(aligned_feature_folder, exist_ok=True)

  # Path to the folder containing landmark data
  landmark_folder_path = os.path.join(folder_path, 'Landmark_data')

  # Iterate over each image in the Images folder
  for image_file in os.listdir(image_folder_path):
    image_path = os.path.join(image_folder_path, image_file)
    image = cv2.imread(image_path)

    # Replace the image file extension with .csv to find the corresponding
landmark file
    landmark_file = image_file.replace('.jpg', '.csv')
    landmark_path = os.path.join(landmark_folder_path, landmark_file)

    # Extract facial landmarks from the landmark file
    features = get_landmark(landmark_path)

    # Align the face in the image based on the landmarks
    faceAligned, featureAligned = align(image, features)

    # Save the aligned image to the Aligned_Images folder
    cv2.imwrite(os.path.join(aligned_image_folder, image_file), faceAligned)

    # Prepare to write the aligned landmark data to a CSV file
    aligned_landmark_file = os.path.join(aligned_feature_folder, landmark_file)
    sym_landmark = []
    with open(aligned_landmark_file, 'w', newline='') as csvfile:
      writer = csv.writer(csvfile)
      writer.writerow(['Landmark index', 'x', 'y'])
      tmp=0

      # Initialize a list to store the data to be written
      data_to_write = []

      # Iterate over each landmark in the aligned feature set
      # Create a symmetrical landmark point list to stabilize the feature
landmarks across different poses
      for i in range(len(featureAligned)):
        if i<17:
          tmp = [400 - featureAligned[16-i][1]-1, featureAligned[16-i][2]]
        elif i<27:
          tmp = [400 - featureAligned[43-i][1]-1, featureAligned[43-i][2]]
        elif i<31:
          tmp = [400 - featureAligned[i][1]-1, featureAligned[i][2]]
```

```
        elif i<36:
          tmp = [400 - featureAligned[66-i][1]-1, featureAligned[66-i][2]]
        elif i<48:
          if (i in range (36, 40)) or (i in range (42, 46)):
            tmp = [400 - featureAligned[81-i][1]-1, featureAligned[81-i][2]]
          else:
            tmp = [400 - featureAligned[87-i][1]-1, featureAligned[87-i][2]]
        else:
          if i==51 or i==57 or i==62 or i==66:
            tmp = [400 - featureAligned[i][1]-1, featureAligned[i][2]]
          elif (i in range(48, 51)) or (i in range(52, 55)):
            tmp = [400 - featureAligned[102-i][1]-1, featureAligned[102-i][2]]
          elif (i in range(55, 57)) or (i in range(58, 60)):
            tmp = [400 - featureAligned[114-i][1]-1, featureAligned[114-i][2]]
          elif (i in range(60, 62)) or (i in range(63, 65)):
            tmp = [400 - featureAligned[124-i][1]-1, featureAligned[124-i][2]]
          else:
            tmp = [400 - featureAligned[132-i][1]-1, featureAligned[132-i][2]]

        # Calculate the symmetrical feature point
        sym_feature = [(tmp[0]+featureAligned[i][1])//2,
(tmp[1]+featureAligned[i][2])//2]

        # Append the symmetrical feature point to the data to be written
        data_to_write.append([i+1, sym_feature[0], sym_feature[1]])

      # Write all the transformed landmark data to the CSV file
      writer.writerows(data_to_write)
```

1. Directory path setup:
   The code starts by establishing the necessary paths for the facial image and landmark data. It creates paths for two main folders ('Face_DB' and 'Test_DB'). 'Face_DB' is designated for the facial images and landmarks used to train and populate the system's database, while 'Test_DB' is reserved for evaluating the system's performance.
2. Preparing storage directories:
   For each of these folders, the code sets up specific paths for storing various types of data:
   - 'image_folder_path' for the original facial images.
   - 'aligned_image_folder' for the images after they have been processed and aligned.
   - 'aligned_feature_folder' for storing feature data extracted from the aligned images.
   - 'landmark_folder_path' for the landmark data files.

   The 'os.makedirs' function is employed to create the directories for aligned images and features if they have not already existed. This step ensures that all necessary directories are

available for storing processed data, maintaining an organization structure for easy access and systematic processing.

3. Processing images and landmarks:
   For every image file in the 'Images' folder of both the 'Face_DB' and 'Test_DB' directories, the code performs the following operations:
   - Reads the image file.
   - Retrieves the corresponding facial landmark data.
   - Aligns the face in the image using these landmarks. This alignment step is critical. as it standardizes the orientation and scale of the faces across all images, making subsequent feature extraction and face comparison more consistent and reliable.
   - Saves the aligned images in the 'Aligned_Images' folder. This creates a set of standardized images that will later be used for feature extraction and face recognition tasks.

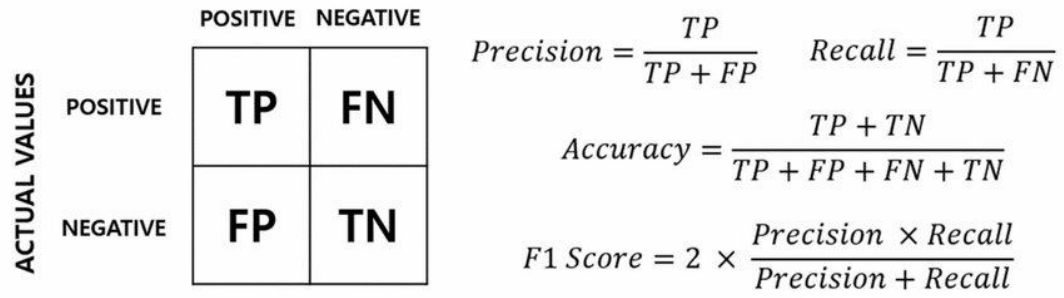
**Model initialization and performance evaluation**

```
# initialize a list of 'instance' objects, which will be fed into the database
people = []
DB_folder = os.path.join(data_path, 'Face_DB', 'Aligned_Features')
for landmark_file in os.listdir(DB_folder):
  landmark_path = os.path.join(DB_folder, landmark_file)
  index = landmark_file.find('_')
  name = landmark_file[:index]
  landmark = get_landmark(landmark_path)
  person = instance(name, landmark)
  people.append(person)

# Initialize a list of models with different thresholds
models = []
for i in range (50, 121):
  models.append(database(i, people))

test_folder = os.path.join(data_path, 'Test_DB', 'Aligned_Features')

# Train models with different thresholds and evaluate their performance
# Test each model, calculate the performance metrics, and save the performance to
a CSV file
thresholds = []
accuracies = []
precisions = []
recalls = []
f1_scores = []
precision_vs_recall = []
```

```
for model in models:
  authorize = 0
  correct_authorize = 0
  incorrect_authorize = 0
  unauthorize = 0
  correct_unauthorize = 0
  incorrect_unauthorize = 0

  for landmark_file in os.listdir(test_folder):
    landmark_path = os.path.join(test_folder, landmark_file)
    index = landmark_file.find('_')
    name = landmark_file[:index]
    landmark = get_landmark(landmark_path)
    person = instance(name, landmark)
    is_in_DB, is_auth, authorize_person = model.face_matching(person)
    # print(f"{is_in_DB} {is_auth} {authorize_person}")
    if is_auth == True:
      authorize += 1
      if authorize_person == name:
        correct_authorize += 1
      else:
        incorrect_authorize += 1
    else:
      unauthorize += 1
      if is_in_DB == False:
        correct_unauthorize += 1
      else:
        incorrect_unauthorize += 1

  accuracy = float(correct_authorize +
correct_unauthorize)/float(authorize+unauthorize)
  precision =
float(correct_authorize)/float(correct_authorize+incorrect_authorize)
  recall =
float(correct_authorize)/float(correct_authorize+incorrect_unauthorize)
  f1_score = float(correct_authorize)/float(correct_authorize+
0.5*float(incorrect_authorize + incorrect_unauthorize))
  pvr = precision/recall
  accuracy = round(accuracy, 4)
  precision = round(precision, 4)
  recall = round(recall, 4)
  f1_score = round(f1_score, 4)
  pvr = round(pvr, 4)
  thresholds.append(model.threshold)
```

```
  accuracies.append(accuracy)
  precisions.append(precision)
  recalls.append(recall)
  f1_scores.append(f1_score)
  precision_vs_recall.append(pvr)

csv_path = os.path.join(HOME, 'performance.csv')
with open(csv_path, 'w', newline = '') as file:
  writer = csv.writer(file)
  writer.writerow(["threshold", "accuracy", "precision", "recall", "f1_score",
"precision vs recall"])
  for i in range(len(thresholds)):
    writer.writerow([thresholds[i], accuracies[i], precisions[i], recalls[i],
f1_scores[i], precision_vs_recall[i]])
```

1. Creating multiple models with different thresholds
   The loop 'for i in range(50, 121)' iterates through a series of numerical values, each representing a potential threshold for face authentication. For each threshold value in the range from 50 to 120 inclusive, a new 'database' object is created and appended to the 'models' list. Each 'database' instance is initialized with the current threshold 'i' and the list of known face authentication systems, each operating with a different level of strictness or leniency in authenticating faces.
2. Purpose of varied thresholds:
   Different thresholds change how the system decides if two faces are similar enough to be considered the same person. A lower threshold makes system stricter (potentially rejecting more authentic faces), whereas a higher threshold makes it more lenient (potentially accepting more imposter faces). By creating models with a range of thresholds, I was able to explore and identify which threshold level offers the best balance between accurately authenticating genuine faces and rejecting imposters.
3. Performance evaluation of models:
   After setting up the models, each one is put through a series of tests to evaluate its performance. These tests typically involve trying to authenticate faces from the 'Test_DB' and measuring how well the model performs. Key performance metrics such as accuracy, precision, recall, and F1-score are calculated for each model. These metrics give a comprehensive understanding of each model's effectiveness.




In the figure above, TP (True Positive) is the number of correct authorizations, FP (False Positive) is the number of incorrect authorizations, TN (True Negative) is the number of correct unauthorizations, and FN (False Negative) is the number of incorrect unauthorizations.

4. The performance metrics for each model are then compiled and stored in a CSV file ('performance.csv'), crucial for analyzing and comparing the performance of the system across different threshold settings, allowing me to select the most effective threshold setting.

**Final evaluation and result recording**

```
# Find the best threshold based on calculate f1_scores
max_f1_score = 0
best_threshold = 0
for i in range(len(precision_vs_recall)):
  if (f1_scores[i] > max_f1_score):
    max_f1_score = f1_scores[i]
    best_threshold = thresholds[i]

# Evaluate the model with the best threshold and record the results into a CSV
file
best_model = database(best_threshold, people)
image_names = []
is_in_DBs = []
is_auths = []
authorize_persons = []
results = []

for landmark_file in os.listdir(test_folder):
  landmark_path = os.path.join(test_folder, landmark_file)
  index = landmark_file.find('_')
  name = landmark_file[:index]
  landmark = get_landmark(landmark_path)
  person = instance(name, landmark)
  is_in_DB, is_auth, authorize_person = best_model.face_matching(person)
  image_name = landmark_file.replace('.csv', '')
  result = ''
```

```
  if is_auth == True:
    authorize += 1
    if authorize_person == name:
      correct_authorize += 1
      result = 'correct authorization'
    else:
      incorrect_authorize += 1
      result = 'incorrect authorization'
  else:
    unauthorize += 1
    if is_in_DB == False:
      correct_unauthorize += 1
      result = 'correct unauthorization'
    else:
      incorrect_unauthorize += 1
      result = 'incorrect unauthorization'
  image_names.append(image_name)
  is_in_DBs.append(is_in_DB)
  is_auths.append(is_auth)
  authorize_persons.append(authorize_person)
  results.append(result)

csv_path = os.path.join(HOME, 'best_threshold_result.csv')
with open(csv_path, 'w', newline = '') as file:
  writer = csv.writer(file)
  writer.writerow(["image_name", "is authentic", "is authorized", "matched
person", "verdict"])
  for i in range(len(image_names)):
    writer.writerow([image_names[i], is_in_DBs[i], is_auths[i],
authorize_persons[i], results[i]])
```

1. Choosing the optimal model:
   The ‘best_threshold’ is identified from earlier tests as the value that yielded the most reliable balance between recognizing authorized individuals and rejecting unauthorized ones (maximum F1-score). The ‘best_model’ is then created with this ‘best_threshold’ and the list of known faces ‘people’ obtained above.
2. Preparing for result compilation:
   Several lists (‘image_names’, ‘is_in_DBs’, ‘is_auths’, ‘authorize_persons’, and ‘results’) are initialized. These will be used to store detailed results for each image processed during the final testing phase.
   - ‘image_names’ will hold the names of the test images.
   - ‘is_in_DBs’ is a Boolean list indicating whether each tested face exists in the database.

- ‘is_auths’ is another boolean list showing whether each face was authenticated by the system.
- ‘authorize_persons’ stores th names of the persons as recognized by the system for each face (if it is authorized)
- ‘results’ will hold the final outcome of each test, categorizing it as correct or incorrect authentication.

3. Conducting the final test:
   The best model is now used to authenticate faces from the ‘Test_DB’. For each face in the test database, the system attempts to identify if it matches any face in the ‘Face_DB’. The decision is based on whether the facial features fall within the acceptable range defined by ‘best_threshold’.
4. Documenting the results
   As the system processes each image, the outcomes are systematically recorded in the initialized lists. This included whether each test face is correctly identified, misidentified or not recognized, along with the system’s decision on whether it believes the face is in the database. All the results are saved in a CSV file (‘best_threshold_result.csv’).

**REFERENCE**

Face alignment tutorial: https://pyimagesearch.com/2017/05/22/face-alignment-with-opencv-and-python/