FACIAL RECOGNITION USING AIML :

ALGORITHM EXPLANATION:

**1. Data Acquisition (Image Capture)**

* **Step 1: Image Collection**  
  The app starts by capturing images of the person’s face using a camera or receiving an image from a dataset (for training).
* **Step 2: Preprocessing**  
  Before feeding the images into a model, the image might undergo preprocessing steps, such as:
  + **Resizing**: Ensuring all images are the same size.
  + **Normalization**: Adjusting pixel values to a specific range (e.g., [0, 1] or [-1, 1]).
  + **Grayscale Conversion**: Converting to grayscale, if necessary, to reduce complexity.

**2. Face Detection**

* **Step 3: Locating Faces**  
  The first step in facial recognition is to detect where the face(s) is/are located in the image. This is typically done using:
  + **Haar Cascades**: A machine learning-based approach for object detection.
  + **Convolutional Neural Networks (CNNs)**: More robust for detecting faces at different angles and under different lighting conditions.
  + **Pre-trained models** like **MTCNN** or **OpenCV’s Haar Classifier** can also be used to detect facial landmarks (eyes, nose, mouth, etc.).

**3. Face Alignment**

* **Step 4: Aligning Faces**  
  Once a face is detected, it's often aligned to correct any tilt or rotation. This step improves the accuracy of the facial recognition model.  
  Techniques like **Affine Transformation** or using facial landmarks (e.g., aligning eyes and mouth to specific points) are commonly employed for alignment.

**4. Feature Extraction**

* **Step 5: Extracting Facial Features**  
  The aligned face is then passed to a neural network, typically a **Convolutional Neural Network (CNN)**, to extract facial features.
  + The CNN learns to identify key facial features (like the distance between the eyes, nose shape, etc.).
  + **Deep learning architectures** like **FaceNet**, **VGG-Face**, or **ResNet** are commonly used for this task.
  + The output is a **vector of features** (often called an **embedding**) that uniquely represents the face.

**5. Face Recognition (Classification or Matching)**

* **Step 6: Comparing Features**  
  Once the facial features (embeddings) are extracted, they are compared against a database of known faces (also represented as embeddings). The comparison can be done using:
  + **Euclidean Distance**: The distance between two feature vectors in multi-dimensional space.
  + **Cosine Similarity**: Another measure of similarity between the feature vectors.
* **Step 7: Classification/Verification**
  + For **facial verification** (1:1 matching), the app checks if the input face matches a known identity.
  + For **facial identification** (1

matching), the app compares the input face against a database of many identities to find the best match.

**6. Training (Optional)**

If you're building the app from scratch, you may need to train the model on a dataset of labeled faces using a supervised learning approach. Here’s how:

* **Training Data**: You need a large dataset of images with labeled faces (e.g., Labeled Faces in the Wild - LFW dataset).
* **Model Training**: The CNN model is trained to extract the facial features and classify the identity based on the embeddings generated.
* **Loss Function**: Typically, **softmax cross-entropy** or **triplet loss** is used to minimize the distance between embeddings of the same person and maximize it between different people.

**7. Post-processing (Optional)**

* **Step 8: Output Interpretation**  
  Based on the distance or similarity scores, the app makes decisions:
  + **Thresholding**: A threshold is set to decide if two faces match (for verification).
  + **Top-k Matching**: For identification, the app could return the top-k closest matches.

**8. Performance Optimization**

* **Step 9: Speed and Efficiency**  
  To make the app responsive, techniques like:
  + **Model Quantization**: Reducing the size of the model for faster inference.
  + **Edge Computing**: Running the recognition locally on a device (like a smartphone) rather than sending data to the cloud.
  + **Hardware Acceleration**: Using GPUs or TPUs to speed up the process.

**9. Accuracy Improvement**

* **Step 10: Fine-tuning Models**  
  Techniques like **transfer learning** (starting with a pre-trained model like FaceNet and fine-tuning on your own dataset) can improve accuracy.
* **Step 11: Continuous Learning**  
  The app can improve over time by learning from more examples. This can involve retraining the model periodically or using approaches like **online learning** where the model adjusts to new inputs without full retraining.

**Flowchart Summary:**

1. **Capture Image → Preprocess Image → Detect Face → Align Face → Extract Features → Match Features → Output Result**.

This combination of face detection, feature extraction, and face matching forms the core of a facial recognition app using AI and ML techniques.

CODE :

import cv2

import dlib

import numpy as np

from keras.models import load\_model

from scipy.spatial.distance import cosine

# Load pre-trained models

detector = dlib.get\_frontal\_face\_detector()

predictor = dlib.shape\_predictor("shape\_predictor\_68\_face\_landmarks.dat")

facenet\_model = load\_model('facenet\_keras.h5')

# Load pre-saved embeddings of known people (replace with actual embeddings)

known\_face\_embeddings = {

'Person1': np.load('person1\_embedding.npy'),

'Person2': np.load('person2\_embedding.npy')

}

# Function to detect faces

def detect\_faces(image):

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

faces = detector(gray)

return faces

# Function to align faces using facial landmarks

def align\_face(image, face):

shape = predictor(image, face)

points = np.array([[p.x, p.y] for p in shape.parts()])

left\_eye = np.mean(points[36:42], axis=0)

right\_eye = np.mean(points[42:48], axis=0)

dY = right\_eye[1] - left\_eye[1]

dX = right\_eye[0] - left\_eye[0]

angle = np.degrees(np.arctan2(dY, dX))

center = tuple(np.mean([left\_eye, right\_eye], axis=0))

M = cv2.getRotationMatrix2D(center, angle, 1)

aligned\_image = cv2.warpAffine(image, M, (image.shape[1], image.shape[0]))

return aligned\_image

# Function to preprocess face image before feeding into FaceNet model

def preprocess\_image(image):

image = cv2.resize(image, (160, 160))

image = image.astype('float32')

mean, std = image.mean(), image.std()

image = (image - mean) / std

return np.expand\_dims(image, axis=0)

# Function to extract facial features using FaceNet

def extract\_features(image):

preprocessed\_image = preprocess\_image(image)

embedding = facenet\_model.predict(preprocessed\_image)

return embedding

# Function to recognize face by comparing extracted embedding with known embeddings

def recognize\_face(embedding, known\_face\_embeddings, threshold=0.5):

min\_distance = float('inf')

recognized\_person = None

for name, known\_embedding in known\_face\_embeddings.items():

distance = cosine(embedding, known\_embedding)

if distance < min\_distance and distance < threshold:

min\_distance = distance

recognized\_person = name

return recognized\_person if recognized\_person else "Unknown"

# Function to run the complete facial recognition pipeline

def run\_facial\_recognition\_app():

cap = cv2.VideoCapture(0)

while True:

ret, frame = cap.read()

if not ret:

break

faces = detect\_faces(frame)

for face in faces:

x, y, w, h = face.left(), face.top(), face.width(), face.height()

aligned\_face = align\_face(frame, face)

face\_features = extract\_features(aligned\_face)

recognized\_person = recognize\_face(face\_features, known\_face\_embeddings)

cv2.putText(frame, recognized\_person, (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 255, 0), 2)

cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

cv2.imshow("Facial Recognition", frame)

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

# Run the app

run\_facial\_recognition\_app()