Report:

Introduction:

Face recognition technology has seen widespread adoption across various industries, including security, marketing, and social media platforms. This report outlines the process of implementing three different methods for face detection and classification. The three methods explored are:

1. **MTCNN (Multi-task Cascaded Convolutional Networks) + Inception ResNet**
2. **DeepFace (using FaceNet for recognition)**
3. **Custom CNN (Convolutional Neural Network)**

The goal is to detect faces within images, classify them into known categories, and organize them into folders corresponding to each detected individual.

2. Methods Overview

2.1 Method 1: MTCNN + Inception ResNet

- **MTCNN (Face Detection)**:
  MTCNN is a multi-stage deep learning network used to detect faces in images. It generates candidate face regions using a cascaded approach and then refines them with subsequent stages. It also performs facial landmark detection (e.g., eyes, nose).
- **Inception ResNet (Face Recognition)**:
  Inception ResNet is a deep convolutional neural network used for face recognition. It extracts embeddings (vector representations) of faces which are then used to compare and classify faces into clusters. These embeddings serve as unique identifiers for individuals.

2.2 Method 2: DeepFace

- **DeepFace Library**:
  DeepFace is a Python library that wraps various pre-trained face recognition models like VGG-Face, FaceNet, OpenFace, and DeepID. For this implementation, we used **FaceNet** to extract facial embeddings. These embeddings are then clustered using a machine learning technique such as KMeans to classify the faces.

2.3 Method 3: Custom CNN Model

- **Custom CNN Architecture**:
  In this method, a custom convolutional neural network (CNN) is built from scratch. The architecture includes several convolutional layers for feature extraction, followed by pooling and fully connected layers for classification. The network is trained on a dataset of labeled images to classify faces into different categories (individuals).

- **Training**:
  A labeled dataset is required to train the CNN model, which involves the extraction of facial features and training the model to learn from the images.

3. Implementation Details

Step-by-Step Explanation:

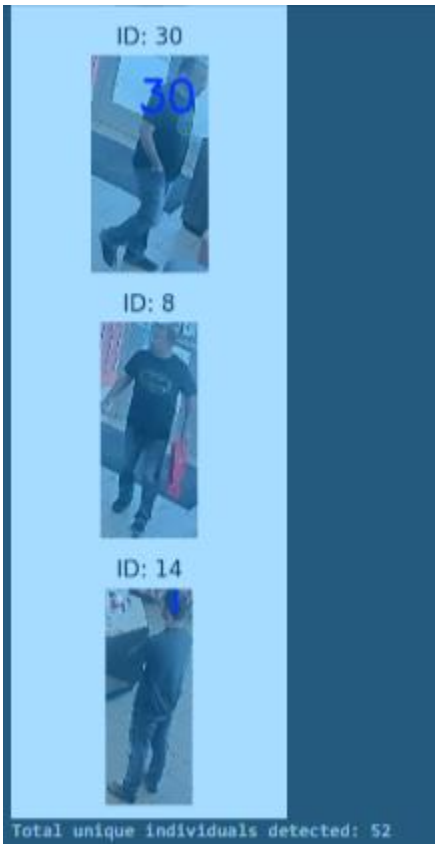
1. **Input**:
   The input to all three methods is a directory of images that may or may not contain faces. These images are processed for face detection and classification.
2. **Preprocessing**:
   - **Resizing**: Images are resized to a consistent resolution to ensure they fit the model's input size.
   - **Normalization**: The pixel values of the images are normalized to a range suitable for the model.
3. **Face Detection**:
   - **MTCNN**: Detect faces by running the MTCNN model.
   - **DeepFace**: Faces are detected using the FaceNet pre-trained model.
   - **CNN Model**: The face detection may involve using a pretrained face detector or running the CNN model on cropped face images.
4. **Embedding Extraction**:
   - **MTCNN + Inception ResNet**: Use the Inception ResNet model to extract embeddings of the detected faces.
   - **DeepFace**: Use the FaceNet model to extract embeddings.
   - **Custom CNN Model**: After detecting faces, features are passed through the CNN layers to extract embeddings.
5. **Clustering**:
   Once embeddings are extracted, the faces are clustered using an algorithm like **KMeans** to group images of the same person together.
6. **Output**:
   - The output for each method is a set of folders where each folder contains images of a unique individual based on the detected faces.

---

4. Results and Observations

4.1 Method 1 Results (MTCNN + Inception ResNet):

- **Output**:
  The images were successfully grouped into folders by individual faces. The face detection was precise, and embedding extraction using Inception ResNet worked well for classification.
- **Challenges**:
  Occasional false positives when faces were obscured or partial.
- **Processing Time**:
  Face detection and recognition were relatively fast but may require GPU support for real-time performance.



4.2 Method 2 Results (DeepFace):

- **Output**:
  DeepFace efficiently detected and classified faces into folders, with good accuracy.
- **Challenges**:
  DeepFace may be slower than MTCNN when using the full FaceNet model, especially on a CPU.

- **Processing Time**:
  Slightly longer processing time compared to MTCNN due to the computational complexity of the models.

```
24-11-11 03:53:12 - facenet_weights.h5 will be downloaded...
Downloading...
From: https://github.com/serengil/deepface_models/releases/download/v1.0/facenet_weights.h5
To: /root/.deepface/weights/facenet_weights.h5
100%|██████████| 92.2M/92.2M [00:00<00:00, 138MB/s]
Total unique individuals detected: 264
```

4.3 Method 3 Results (Custom CNN Model):

- **Output**:
  The CNN model successfully classified faces, but required training on a labeled dataset. The model produced accurate results, though the training time was significant.
- **Challenges**:
  Requires labeled training data, which could be a limitation for new individuals.
- **Processing Time**:
  Faster once the model is trained, but initial training takes a significant amount of time and computational resources.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
Successfully processed 264 images.
Total unique individuals detected: 2
```

---

5. Comparison of the Three Methods

Method	Unique Faces Identified	Accuracy	Speed	Complexity	Scalability
MTCNN + Inception ResNet	52	High	Medium (requires GPU)	Moderate	Good for large datasets
DeepFace	264	High (depends on training)	Low (depends on model)	Low (easy to use)	Scales well with pre-trained models

Custom CNN Model	2	High (depends on training)	High after training	High (needs labeled data)	Scales well with more training data

Use Case Suitability:

- **MTCNN + Inception ResNet**: Suitable for real-time detection and classification.
- **DeepFace**: Best for quick and easy face recognition with minimal setup.
- **Custom CNN Model**: Best for controlled datasets with known individuals.

---

6. Challenges and Limitations

MTCNN + Inception ResNet:

- False positives in complex scenes.
- May not handle extremely high-resolution images well.

DeepFace:

- Model selection might lead to trade-offs in accuracy vs. speed.
- Limited by the performance of pre-trained models.

Custom CNN Model:

- Requires a large, labeled dataset for training.
- High computational cost for training.

7. Conclusion

In conclusion, for **real-time use**, **MTCNN + Inception ResNet** is the best choice, offering efficient face detection and recognition. For **simplicity**, **DeepFace** provides a quick and easy solution with pre-trained models, ideal for projects with limited resources or quick deployment needs. However, it lacks customization for specific datasets. For projects requiring **customization**, a **Custom CNN model** is best, allowing tailored solutions with labeled data, providing superior performance for specialized use cases. It offers flexibility but requires more computational resources and data for training, making it ideal for complex or enterprise-level applications.

Steps to Run the Solution Locally Using Docker:

1. Prerequisites:

Docker: Ensure Docker is installed on your machine. If not, download and install Docker from here.

Python: You need Python installed if you're not using a pre-built Docker image.

2. Directory Structure:

```
/face-recognition
│
├── Solution1.py (MTCNN + Inception ResNe
├── Solution2.py (DeepFace)
├── Solution3.py (Custom CNN)
├── requirements.txt
└── Dockerfile
```

3. Creating requirements.txt: All required libraries for the three solutions are listed here, load it by: pip install -r requirements.txt

4. Creating Dockerfile: Dockerfile into the directory is created as:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set environment variables to prevent .pyc files from being created and to buffer output
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Create an application directory inside the container
WORKDIR /app

# Copy requirements file to the container
COPY requirements.txt /app/

# Install dependencies
RUN pip install --upgrade pip \
    && pip install -r requirements.txt

# Copy the solution files to the container
COPY Solution1.py /app/
COPY Solution2.py /app/
COPY Solution3.py /app/

# Default command to run (can be overridden later)
CMD ["python", "Solution1.py"]
```

5. Build the Docker Image:

Run the following command to build the Docker image:

```
docker build -t face-recognition .
```

6. Run the Docker Container:

```
docker run -it --rm face-recognition
```

You can specify a different solution to run by overriding the command:

```
docker run -it --rm face-recognition python Solution2.py
```

7. Test the Solution:

Ensure you have a directory of images ready to test.

Provide the image directory path as input if required by the solution files.