Face Recognition System

A Comprehensive Manual

Contents

CHAPTER 1

Introduction

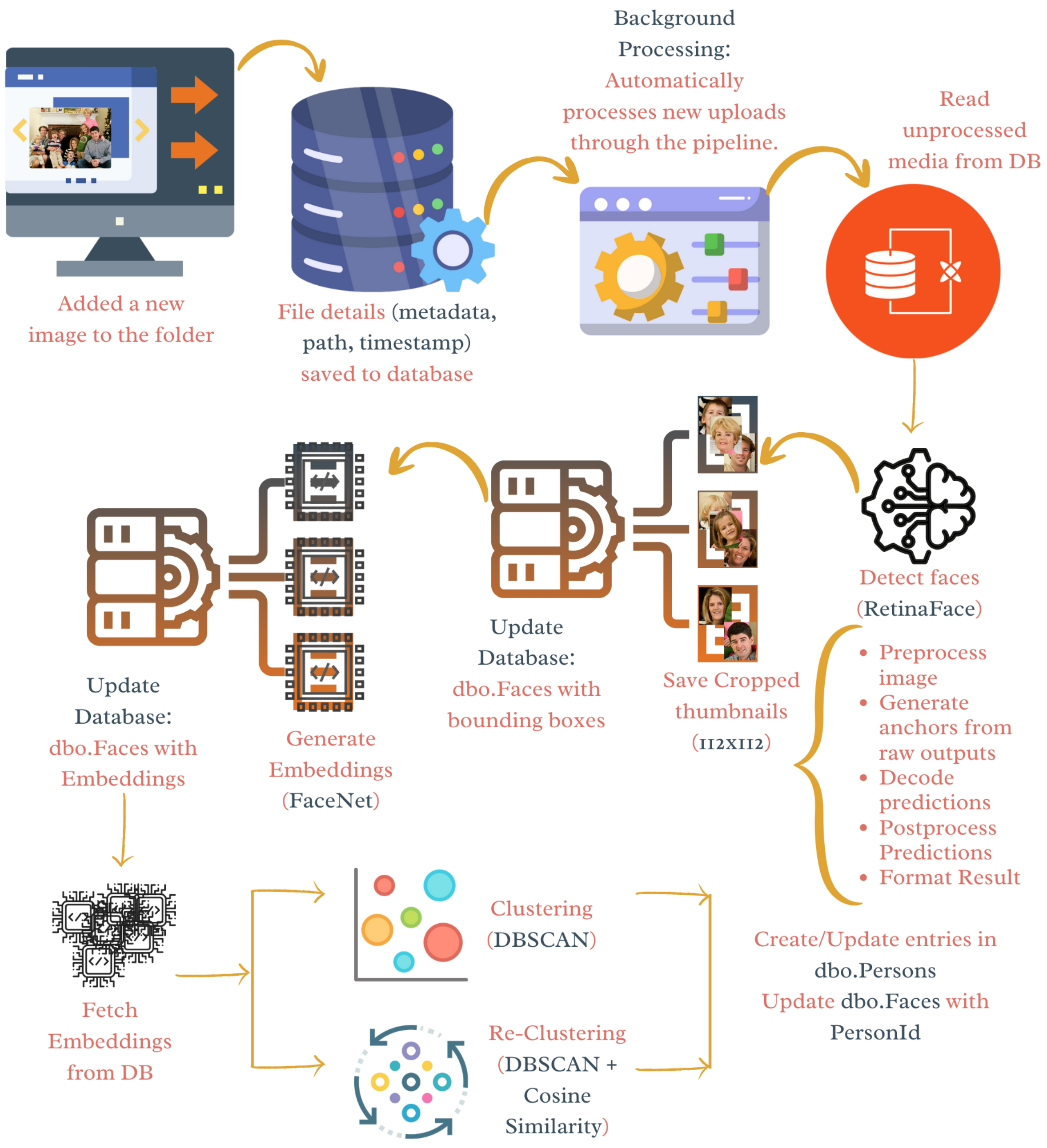
The **Face Recognition System** is a comprehensive solution for detecting and recognizing faces in images and media stored in a database. It integrates multiple stages of the face recognition pipeline, including detection, embedding generation, and clustering, providing both automated and manual processing options. This system is suitable for single-image testing, batch processing, continuous monitoring, and recurring automated pipelines.

1.1 Key Features

- **Face Detection:** Accurately detect faces in images or media items stored in a database.
- **Face Recognition:** Generate embeddings for detected faces and cluster them into distinct individuals.
- **Automated Pipelines:** Run the full detection and recognition process periodically without manual intervention.
- **Reclustering:** Recompute clusters for unassigned faces by comparing them with each other or matching them against previously assigned persons.

The system leverages **FaceNet** for embedding generation and **DBSCAN + UMAP** for clustering embeddings, with **Cosine Similarity** used to assign unlabelled faces to known persons when similarity meets or exceeds 0.8. Detailed logging is provided for both face detection and recognition processes, enabling effective monitoring and debugging.

This manual provides step-by-step instructions for installing the system, configuring the database, running the pipeline in various modes, and understanding the project structure and workflow. It is intended to help users quickly set up and efficiently use the Face Recognition System for their applications.



Added a new image to the folder
File details (metadata, path, timestamp) saved to database
Background Processing: Automatically processes new uploads through the pipeline.
Read unprocessed media from DB
Detect faces (RetinaFace)
• Preprocess image
• Generate anchors from raw outputs
• Decode predictions
• Postprocess Predictions
• Format Result
Save Cropped thumbnails (112x112)
Update Database: dbo.Faces with bounding boxes
Generate Embeddings (FaceNet)
Update Database: dbo.Faces with Embeddings
Fetch Embeddings from DB
Clustering (DBSCAN)
Re-Clustering (DBSCAN + Cosine Similarity)
Create/Update entries in dbo.Persons
Update dbo.Faces with PersonId

Figure 1.1: Face Recognition System Workflow

CHAPTER 2

Installation and setup

2.1 Prerequisites

- Python 3.8.20
- SQL Server 2019+
- SSMS (v21+)
- Conda (recommended)

2.2 Step-by-Step Installation

2.2.1 Clone the Repository

```
git clone https://github.com/DeepFrame/deepframe-backend.git
cd deepframe-backend/services/image_grouping
```

2.2.2 Create and Activate Conda Environment

```
conda create -n face_recognition python=3.8.20
conda activate face_recognition
```

2.2.3 Install Dependencies

```
pip install -r requirements.txt
```

2.3 Environment Configuration

Create a `.env` file with the following content:

```
SQL_CONNECTION_STRING="Driver={SQL Server};
Server=your_server_name;
Database=MetaData;
Encrypt=no;
TrustServerCertificate=no;"
THUMBNAIL_SAVE_PATH="path/to/thumbnails/directory"
```

2.4 Database Configuration

2.4.1 Required Tables

Ensure your database contains these tables with appropriate fields.

MediaFile Table

```sql
CREATE TABLE dbo.MediaFile (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    FilePath NVARCHAR(500) NOT NULL,
    FileName NVARCHAR(255) NOT NULL,
    Extensions NVARCHAR(10) NULL,
    -- other media metadata fields
);
```

MediaItems Table

```sql
CREATE TABLE dbo.MediaItems (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    IsFacesExtracted BIT DEFAULT 0,
    FacesExtractedOn DATETIME,
    -- other media metadata fields
);
```

Faces Table

```sql
CREATE TABLE dbo.Faces (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    MediaFileId INT,
    PersonId INT NULL,
    BoundingBox NVARCHAR(255) NULL,
    Embedding VARBINARY(MAX) NULL,
    -- other face metadata
);
```

Persons Table

```sql
CREATE TABLE dbo.Persons (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    Name VARCHAR(255) NULL,
    -- other person metadata
);
```

CHAPTER 3

Face Detection Pipeline

3.1 Objective

- Read unprocessed media entries from SQL Server
- Detect faces using RetinaFace
- Save cropped face thumbnails
- Update database with results

3.2 Working of RetinaFace Model

The **RetinaFace** model is used for face detection, providing bounding boxes and facial landmarks. Its workflow can be summarized as follows:

Model Loading

- The image is loaded from disk.
- The model is instantiated using a **Singleton pattern**:
  - If no model weights exist in memory, a new one is built.
  - If already loaded globally, the existing model is reused.
  - The model supports flexible image dimensions and batch sizes.

Constants and Parameters

- **Feature strides:** $[32, 16, 8]$ determine downsampling levels of the original image.
  - Stride 32 $\rightarrow$ large faces, less detail.
  - Stride 8 $\rightarrow$ small faces, high detail.
- Anchor templates per stride.
- Non-Max Suppression (NMS) threshold = 0.4.
- Score decay = 0.4 to reduce the influence of coarse stride predictions.

Preprocessing

- Resize image while preserving aspect ratio (shorter $\leq$ 1024 px, longer $\leq$ 1980 px).
- Convert image to `float32`.
- Convert from OpenCV BGR to RGB.
- Normalize pixel values.
- Create 4D tensors of shape: `[Batch, Height, Width, Channels]`.
- Output preprocessing returns `[tensor, image_input, scale_factor]`:
  - Preprocessed tensor.
  - Scaled input image.
  - Scale factor (original $\div$ resized).

Inference

- The model processes the tensor and outputs raw feature maps:
  - Classification scores.
  - Bounding box regression deltas.
  - Landmark deltas.
- Tensors are converted to NumPy arrays for further processing.

Post-Processing

- Generate anchors for each stride level.
- Reshape and align scores/deltas with anchors.
- Apply regression to refine anchor boxes into predicted bounding boxes.
- Clip boxes to stay within image boundaries.
- Apply score decay to small-scale anchors to avoid overconfidence.
- Filter out low-confidence detections.
- Rescale predicted boxes back to the original image coordinates.
- Extract and decode landmark positions (eyes, nose, mouth corners).
- Merge proposals from all FPN strides $(32, 16, 8)$.
- Sort scores in descending order, align boxes and landmarks accordingly.
- Apply Non-Maximum Suppression (NMS) to retain final predictions.
- Format output as:
  - Confidence score.
  - Bounding box coordinates.
  - Facial landmarks (5-point).

3.3 Face Cropping

Inputs

- Original image (NumPy array from OpenCV).
- Final detected bounding boxes after NMS.
- Extracted facial landmarks for each face.

Cropping Process

- Crop face regions with margin from the original image in square shape.
- Resize each crop to a fixed thumbnail size (e.g., 112×112).
- If crop is smaller than expected, skip saving.

Saving Cropped Faces

- Cropped faces are saved using OpenCV `cv2.imwrite()`.
- Filenames include the original image name, face index, and file extension.

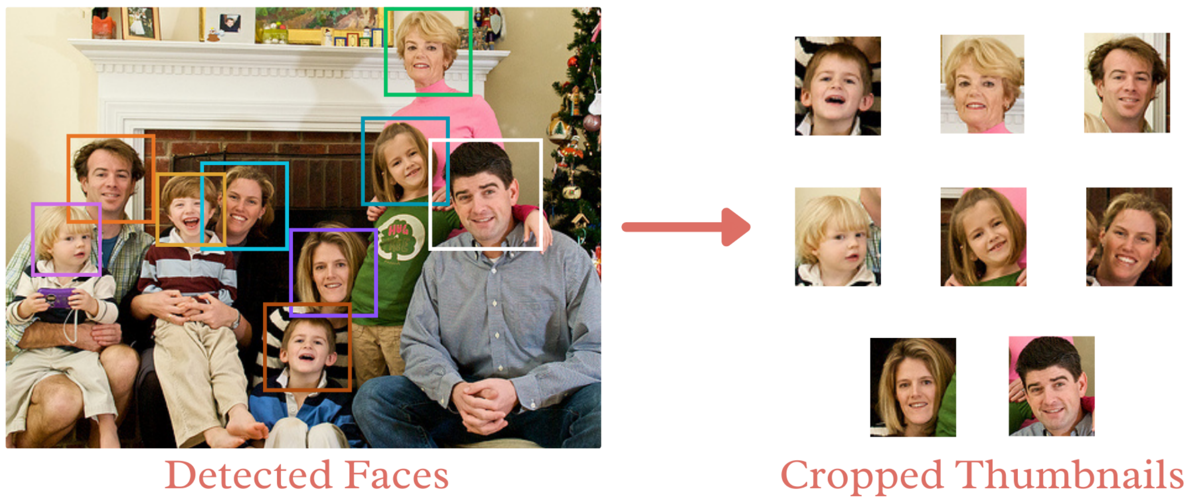


Figure 3.1: Face Detection and Cropping Workflow

CHAPTER 4

Face Clustering Pipeline

4.1 Fetch Faces Without Embeddings

- Retrieve face records from the database where `Embedding IS NULL`.
- Each record contains:
    - FaceId
    - Image path (`FilePath`)
    - Bounding box (`BoundingBox`)

4.2 Generate Face Embeddings

For each face:

- Load image using OpenCV.
- Crop face using bounding box coordinates.
- Preprocess face:
    - Convert BGR $\rightarrow$ RGB.
    - Resize to 160$\times$160 pixels (FaceNet input size).
- Extract embedding using FaceNet:
    - FaceNet: deep CNN trained with triplet loss.
    - Minimizes distance between embeddings of the same person.
    - Maximizes distance between embeddings of different people.
    - Output: 512-dimensional embedding vector unique to that face.
- Update database with generated embedding (float32 stored as bytes).

4.3 Retrieve Unassigned Faces

- Faces with embeddings but no `PersonId`.

Recluster Mode

- Fetch all faces with embeddings (labelled & unlabelled).
- Ensure embeddings exist (generate if missing).
- Fetch labelled faces $\rightarrow$ embeddings + `PersonId`.
- Compare unlabelled embeddings with labelled ones using cosine similarity.
- If similarity ≥ 0.8 (threshold) $\rightarrow$ assign unlabelled face to that person.
- Remaining unassigned faces $\rightarrow$ clustered using DBSCAN (cosine distance).

Default Mode

- Fetch only unlabelled faces with embeddings.
- Apply UMAP for dimensionality reduction (better cluster separation).
- Cluster embeddings with DBSCAN (cosine distance).
- Assign cluster results to new persons.

4.4 Parse Stored Embeddings

- Database embeddings may be stored as:
  - Binary bytes (`np.frombuffer()`)
  - JSON string (`json.loads()` → NumPy array)
- Converted back to NumPy arrays for clustering/similarity checks.

4.5 Assign Clusters to Faces

- Direct Assignment (Existing Person)
  - If a face matches an already known person, update `Faces.PersonId`.
- Clustering Mode
  - If a new cluster is detected:
    - Create new entry in `Persons` table.
    - Link all faces in that cluster to the new `PersonId`.

4.6 Update Database

- Update `Faces.PersonId` with assignments.
- Insert new `Persons` records when needed.
- Maintain audit fields: `CreatedAt`, `ModifiedAt`.

CHAPTER 5

Execution of the System

The Face Recognition System can be executed in multiple modes through the main command-line interface (CLI) script `main.py`. Execution is divided into two major stages: **Face Detection** and **Face Recognition**.

5.1 Face Detection Execution

Execution Modes

- **Single Image Test** Runs the detector on one image file.

  ```
  python main.py --test path/to/image.jpg
  ```

- **Database Batch Processing** Reads all unprocessed media items from the database and updates results.

  ```
  python main.py --db
  ```

- **Continuous Monitoring** Watches the database and processes new media items periodically.

  ```
  python main.py --watch
  ```

Detection Flow

1. Connect to database and fetch unprocessed media.
2. Load each media item, preprocess, and run the RetinaFace model.
3. Extract bounding boxes and facial landmarks.
4. Crop and resize faces (112×112 thumbnails).
5. Save thumbnails to the `Thumbnails/` directory.
6. Update database:
   - Mark `IsFacesExtracted = TRUE`.
   - Save timestamp and bounding box metadata.
7. Log results to `logs/face_detection.log`.

5.2 Face Recognition Execution

Execution Modes

- **Recognition (Clustering)** Generate embeddings for unlabelled faces and cluster them.

  ```
      python main.py --recognize
  ```

- **Reclustering** Performs detection first, then matches unlabelled faces against labelled ones using cosine similarity, and finally builds new clusters for the remaining unlabelled faces with DBSCAN.

  ```
      python main.py --recluster
  ```

- **Full Pipeline (Detection + Recognition)** Runs the entire pipeline once.

  ```
      python main.py --all
  ```

- **Automated Pipeline** Runs detection and recognition every 3 minutes.

  ```
      python main.py --automate
  ```

Recognition Flow

1. Fetch detected faces with missing embeddings from the database.
2. Crop faces and generate embeddings using **FaceNet**.
3. Store embeddings in the database.
4. Cluster embeddings using **UMAP + DBSCAN**.
5. Assign clusters as `PersonId` for identified groups.
6. Incrementally match new embeddings with existing clusters using cosine similarity.
7. Store logs in `logs/embeddings_clustering.log`.

5.3 Outputs

- **Filesystem:** Cropped thumbnails saved in the `Thumbnails/` directory.
- **Database:** Updated `MediaItems`, `Faces`, and `Persons` tables containing:
  - Media item status (`IsFacesExtracted`).
  - Extraction and recognition timestamps.
  - Cropped face metadata (bounding box, landmarks, embedding vectors).
  - Cluster assignments (`PersonId`).

- **Logs:**
  - `logs/face_detection.log` – detection events, errors, database updates.
  - `logs/embeddings_clustering.log` – embeddings, clustering results, and errors.