Facet

A Specification for Organizing Local Photos by Detected Faces

(Includes Full Source Code Appendices)

Ashutosh Gupta

Version 1.0

Contents

1 Introduction

1.1 Motivation

Digital photo collections often grow uncontrollably, making it difficult to find pictures of specific individuals. While cloud services offer face recognition, privacy concerns arise from uploading personal photos to remote servers. There is a need for a tool that can organize photos based on faces locally, keeping user data private.

1.2 Project Goal

The **Facet** project aims to develop a command-line tool that automatically scans a user-specified directory of photos, detects faces within those photos, groups photos containing the same individuals using facial recognition and clustering, and organizes copies of these photos into separate folders, all running entirely on the user's local machine.

1.3 Scope

This blueprint outlines the design for Facet Version 1.0, focusing on:

- Scanning a local folder (and subfolders) for common image file types.
- Detecting faces in images using established computer vision libraries.
- Generating unique numerical embeddings (face fingerprints) for each detected face.
- Clustering similar face embeddings together using unsupervised machine learning (DBSCAN) to identify unique individuals without prior knowledge.
- Creating output folders (e.g., `Person_1`, `Person_2`) representing each identified cluster (person).
- Copying the original photo files containing faces belonging to a cluster into the respective person's output folder.
- Providing command-line arguments for specifying input and output directories and tuning clustering parameters.

Features outside the initial scope include: a graphical user interface (GUI), real-time processing, identifying known people based on reference images, merging clusters, tagging photo metadata instead of copying files, and advanced error handling for corrupted images.

2 System Architecture

Facet operates as a command-line script that processes images sequentially through a pipeline involving file system scanning, face detection, face embedding generation, clustering, and file organization.

Workflow Description:

1. **Initialization:** The script (`sort_faces.py`) parses command-line arguments for input and output directories and optional clustering parameters.
2. **File Discovery:** The script scans the specified input directory recursively to find all supported image files.
3. **Face Processing Loop:** For each image found:
    - The image is loaded into memory.

Architecture Diagram Placeholder

(A diagram showing: User Input (CLI Arguments: Input Path, Output Path) → Python Script (`sort_faces.py`) → Scan Files (`os.walk`) → For Each Image: Load Image (`face_recognition/Pillow`) → Detect Faces (`face_recognition`) → Generate Embeddings (`face_recognition`) → Collect All Embeddings → Cluster Embeddings (`sklearn.DBSCAN`) → Map Images to Cluster IDs → Create Output Folders (`os.makedirs`) → Copy Image Files (`shutil.copy2`) → Output Folders (`Person_1`, `Person_2`...).)

Figure 1: High-Level System Architecture Diagram

- The `face_recognition` library detects the locations (bounding boxes) of all faces within the image.
- For each detected face, a unique 128-dimension numerical embedding (vector) is generated.
- Each embedding is stored along with the path to the original image file.

4. **Clustering:** After processing all images, all collected face embeddings are passed to the DBSCAN clustering algorithm from `scikit-learn`. DBSCAN groups embeddings that are close to each other in the 128-dimensional space, effectively grouping faces of the same person. Each face embedding is assigned a cluster label (integer). Noise points (faces that don't fit well into any cluster) are assigned label -1.
5. **Image Grouping:** The script creates a mapping between cluster labels and the set of original image paths associated with faces belonging to that cluster.
6. **File Organization:**
   - The specified output directory is created if it doesn't exist.
   - For each cluster label (excluding noise label -1):
     - A new subdirectory is created within the output directory (e.g., `Person_1`, `Person_2`, ...).
     - The script iterates through the unique image paths associated with the current cluster.
     - A copy of each unique image file is placed into the corresponding person's subdirectory using `shutil.copy2` (preserving metadata).
7. **Completion:** The script finishes after processing all clusters and copying files, logging summary information.

3 Core Components

3.1 Main Script (`sort_faces.py`)

- **Technology:** Python 3

- **Role:** Orchestrates the entire workflow. Handles argument parsing, file system scanning (`os`), image loading, calls face detection/encoding functions, initiates clustering, maps results, and performs file copying (`shutil`). Includes logging for progress feedback.

3.2 Face Detection & Recognition Library

- **Technology:** `face_recognition` (Python library, based on `dlib`)
- **Role:** Provides high-level functions to:
  - Load image files.
  - Detect the locations of faces within images (using HOG or CNN models).
  - Generate 128-dimension face embeddings (face fingerprints) from detected faces.

3.3 Clustering Library

- **Technology:** `scikit-learn` (Python library)
- **Role:** Provides the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm used to group similar face embeddings together without needing to know the number of people beforehand.

3.4 Supporting Libraries

- **Technology:** `numpy`, `Pillow`, `opencv-python`
- **Role:**
  - `numpy`: Used for efficient numerical operations, particularly handling the face embedding arrays required by `scikit-learn`.
  - `Pillow`: An imaging library, often used as a backend or dependency by `face_recognition` for image loading and manipulation.
  - `opencv-python`: While `face_recognition` can use its own methods, OpenCV is often installed as it provides underlying image processing capabilities or alternative face detectors if needed. The script might implicitly benefit from its presence.

4 Technology Stack

- **Programming Language:** Python 3
- **Face Recognition:** `face_recognition` library (using `dlib` backend)
- **Clustering:** `scikit-learn` library (DBSCAN algorithm)
- **Numerical Computation:** `numpy`
- **Image Handling:** `Pillow`, `opencv-python` (often dependencies)
- **Command-Line Interface:** `argparse` (Python standard library)
- **File System Operations:** `os`, `shutil` (Python standard library)
- **Version Control:** Git

5 Setup and Installation

This section provides the detailed steps required to set up the Facet project environment locally.

5.1 Prerequisites

- **Python:** Version 3.x. Verify with `python -version` or `python3 -version`. (Download)
- **pip:** Python package installer (usually included with Python).
- **Build Tools (Potentially Required for `dlib`):** The `face_recognition` library depends on `dlib`. Installing `dlib` often requires:
  - **CMake:** (Download)
  - **C++ Compiler:**
    - **Linux (Debian/Ubuntu):** `sudo apt update && sudo apt install build-essential cmake`
    - **macOS:** Install Xcode Command Line Tools: `xcode-select -install` (includes Clang compiler and CMake might need separate install via Homebrew: `brew install cmake`).
    - **Windows:** Install C++ build tools for Visual Studio (select "Desktop development with C++" workload during VS Installer). Make sure CMake is installed and added to PATH.

  Refer to the official `face_recognition` installation guide for OS-specific details if you encounter issues.

5.2 Detailed Setup Steps

1. **Obtain Project Files:**
   - **Manual:** Download or create the project files (`sort_faces.py`, `requirements.txt`, etc.) in a directory named `Facet`. Navigate into this directory using your terminal.
2. **Create Python Virtual Environment (Recommended):** Isolates project dependencies. Execute within the `Facet` directory:

   ```
   python -m venv venv
   # Activate (Linux/macOS): source venv/bin/activate
   # Activate (Windows CMD): venv\Scripts\activate.bat
   # Activate (Windows PS): .\venv\Scripts\Activate.ps1
   ```

   Your terminal prompt should indicate the active environment (e.g., `(venv)`).
3. **Install Python Dependencies:** Install the required packages using pip. **Ensure prerequisites (CMake, C++ compiler) are installed before this step.**

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

   (See Appendix C for contents. This might take some time, especially compiling dlib.)

Setup is complete. Proceed to Section 6 for running the tool.

6 Usage

1. Activate the Python virtual environment (if not already active):
   `source venv/bin/activate` (or equivalent for your OS).
2. Navigate to the project's root directory (`Facet`).
3. Run the main script from the command line, providing the required input and output paths:

   ```
   python sort_faces.py --input_folder "/path/to/your/photos" --output_folder "/path/to/
       sorted_output"
   ```

   - Replace `"/path/to/your/photos"` with the full path to the directory containing the images you want to scan.
   - Replace `"/path/to/sorted_output"` with the full path where the organized folders (`Person_1`, `Person_2`, etc.) should be created. This directory will be created if it doesn't exist.
4. **(Optional)** Tune clustering parameters:

   ```
   python sort_faces.py -i "/photos" -o "/sorted" --eps 0.5 --min_samples 3
   ```

   - Use `-eps` (default: 0.55) to control similarity tolerance (lower is stricter).
   - Use `-min_samples` (default: 2) to set the minimum number of times a face must appear to form a distinct cluster.
5. The script will log its progress to the console. Processing can take a significant amount of time depending on the number and size of photos.
6. Once finished, check the specified output folder for subdirectories named `Person_X` containing copies of the photos organized by detected faces.

7 Project Structure

The directory structure is as follows (items marked (*) are typically managed locally and ignored by Git):

```
Facet/
         .gitignore                # Specifies files/folders for Git to ignore
         LICENSE                   # Project license file
         README.md                 # Project documentation file
         sort_faces.py             # Main Python script for face sorting
         requirements.txt          # Python package dependencies

         venv/ (*)                 # Python virtual environment (local)

# Potential additional files/folders created locally (*)
# __pycache__/ (*)            # Python bytecode cache (local)
# Output folder specified by user (e.g., sorted_output/) (*)
#            Person_1/ (*)
#            Person_2/ (*)
#                ...
```

Listing 1: Project Directory Layout

Note: Files/folders marked with (*) are generally created locally during setup or runtime and are typically excluded from the Git repository via the `.gitignore` file (Appendix D).

8 Future Work

Potential enhancements for Facet:

- **GUI Interface:** Develop a graphical user interface (e.g., using Tkinter, PyQt, Kivy) for easier folder selection, progress visualization, and viewing results.
- **Naming/Merging Clusters:** Allow users to review the generated `Person_X` folders and assign actual names or merge folders representing the same person clustered separately.
- **Known Face Recognition:** Implement functionality to recognize pre-defined individuals based on reference photos.
- **Metadata Tagging:** Add an option to write face information (identified person name/ID) to image metadata (e.g., EXIF/XMP keywords) instead of, or in addition to, copying files.
- **Performance Optimization:** Explore batch processing of images, GPU acceleration (if `dlib` is compiled with CUDA support), or alternative, faster face detection models (e.g., from OpenCV Zoo).
- **Incremental Updates:** Add capability to scan a folder again and only process newly added photos, integrating them into existing clusters or forming new ones.
- **Improved Clustering:** Experiment with other clustering algorithms or post-processing steps to refine cluster quality.
- **Configuration File:** Use a configuration file (e.g., YAML, JSON) for settings like `eps`, `min_samples`, image extensions, etc., instead of only command-line arguments.

9 License (Summary)

This project is assumed to be distributed under the MIT License, based on common practice. The full license text placeholder is provided in Appendix A.

A License (MIT)

The following is a placeholder for the MIT License. Replace with the actual license if available, or use this standard text.

```
MIT License

Copyright (c) 2025 Ashutosh Gupta

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
```

```
SOFTWARE.
```

Listing 2: MIT License Text Placeholder (‘LICENSE‘)

B Main Script (sort_faces.py)

```python
import face_recognition
import os
import shutil
import argparse
import numpy as np
from sklearn.cluster import DBSCAN
from collections import defaultdict
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# --- Configuration ---
# Supported image file extensions
IMAGE_EXTENSIONS = ('.jpg', '.jpeg', '.png', '.gif', '.bmp', '.tiff')

# DBSCAN parameters (tune these based on your results)
# eps: Max distance between samples for one to be considered as in the neighborhood of the other.
#      Lower values mean faces need to be more similar to be grouped. Start around 0.5-0.6.
DBSCAN_EPS = 0.55
# min_samples: Number of samples (faces) in a neighborhood for a point to be considered as a core point.
#              Essentially, how many times does a face need to appear to be considered a distinct person?
DBSCAN_MIN_SAMPLES = 2 # Increase if you only want to group people appearing more often

# --- Helper Functions ---

def find_image_files(folder_path):
    """Recursively finds all image files in the given folder."""
    image_files = []
    logging.info(f"Scanning for images in: {folder_path}")
    for root, _, files in os.walk(folder_path):
        for filename in files:
            if filename.lower().endswith(IMAGE_EXTENSIONS):
                image_files.append(os.path.join(root, filename))
    logging.info(f"Found {len(image_files)} image files.")
    return image_files

def extract_face_data(image_paths):
    """Extracts face encodings and their corresponding paths from images."""
    face_data = [] # List of dictionaries: {'path': path, 'encoding': encoding_vector}
    total_images = len(image_paths)
    logging.info(f"Starting face detection and encoding for {total_images} images...")

    for i, image_path in enumerate(image_paths):
        logging.info(f"Processing image {i + 1}/{total_images}: {os.path.basename(image_path)}")
        try:
            # Load image
            image = face_recognition.load_image_file(image_path)
            # Find face locations (using CNN model is more accurate but slower, default is HOG)
            # face_locations = face_recognition.face_locations(image, model="cnn")
            face_locations = face_recognition.face_locations(image)

            if not face_locations:
                logging.debug(f"No faces found in {os.path.basename(image_path)}")
                continue

            # Get face encodings
```

```
            # Providing known locations speeds up encoding
            face_encodings = face_recognition.face_encodings(image, known_face_locations=face_locations)

            logging.debug(f"Found {len(face_encodings)} face(s) in {os.path.basename(image_path)}")

            # Store each encoding with its path
            for encoding in face_encodings:
                face_data.append({'path': image_path, 'encoding': encoding})

        except Exception as e:
            logging.warning(f"Could not process image {image_path}: {e}")

    logging.info(f"Finished encoding. Found {len(face_data)} total face instances.")
    return face_data

def cluster_faces(face_data):
    """Clusters face encodings using DBSCAN."""
    if not face_data:
        logging.warning("No face data to cluster.")
        return None, {} # Return None for labels, empty dict for data

    logging.info("Starting face clustering...")
    encodings = np.array([data['encoding'] for data in face_data])

    # Create and fit the DBSCAN model
    # n_jobs=-1 uses all available CPU cores
    clusterer = DBSCAN(eps=DBSCAN_EPS, min_samples=DBSCAN_MIN_SAMPLES, metric='euclidean', n_jobs=-1)
    clusterer.fit(encodings)

    labels = clusterer.labels_ # Get cluster labels for each face encoding
    num_clusters = len(set(labels)) - (1 if -1 in labels else 0) # -1 label is for noise/outliers
    logging.info(f"Clustering complete. Found {num_clusters} distinct clusters (people) excluding noise.")

    return labels, face_data

def organize_photos_by_cluster(labels, face_data, output_dir):
    """Copies photos into folders based on cluster labels."""
    if labels is None:
        logging.error("Cannot organize photos, clustering failed or produced no results.")
        return

    logging.info(f"Organizing photos into: {output_dir}")
    os.makedirs(output_dir, exist_ok=True)

    # Group image paths by cluster label
    images_by_cluster = defaultdict(set) # Use set to store unique image paths per cluster
    for label, data in zip(labels, face_data):
        if label != -1: # Ignore noise points
            images_by_cluster[label].add(data['path'])

    if not images_by_cluster:
        logging.warning("No valid clusters found to organize photos.")
        return

    # Copy files for each cluster
    cluster_count = 0
    for label, image_paths in images_by_cluster.items():
        cluster_count += 1
```

```
        person_folder_name = f"Person_{cluster_count}" # Assign sequential names
        person_output_path = os.path.join(output_dir, person_folder_name)
        os.makedirs(person_output_path, exist_ok=True)
        logging.info(f"Copying {len(image_paths)} unique images for {person_folder_name} (Cluster Label {label})...")

        for image_path in image_paths:
            try:
                dest_path = os.path.join(person_output_path, os.path.basename(image_path))
                # Avoid copying if somehow the exact same file is listed twice for the cluster
                if not os.path.exists(dest_path):
                     # copy2 preserves metadata like creation/modification time
                    shutil.copy2(image_path, dest_path)
                else:
                    logging.debug(f"Skipping already copied file: {dest_path}")
            except Exception as e:
                logging.error(f"Failed to copy {os.path.basename(image_path)} to {person_folder_name}: {e}")

    logging.info("Finished organizing photos.")


# --- Main Execution ---
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Sort photos locally by detected faces using clustering.")
    parser.add_argument("-i", "--input_folder", required=True, help="Path to the folder containing photos to scan.")
    parser.add_argument("-o", "--output_folder", required=True, help="Path to the folder where sorted photos (Person_1, Person_2, ...) will be copied.")
    # Optional arguments for tuning
    parser.add_argument("--eps", type=float, default=DBSCAN_EPS, help=f"DBSCAN epsilon (max distance). Default: {DBSCAN_EPS}")
    parser.add_argument("--min_samples", type=int, default=DBSCAN_MIN_SAMPLES, help=f"DBSCAN min samples per cluster. Default: {DBSCAN_MIN_SAMPLES}")


    args = parser.parse_args()

    # Validate input path
    if not os.path.isdir(args.input_folder):
        logging.error(f"Input folder not found or is not a directory: {args.input_folder}")
        exit(1)

    # Use provided tuning parameters if given
    DBSCAN_EPS = args.eps
    DBSCAN_MIN_SAMPLES = args.min_samples
    logging.info(f"Using DBSCAN settings: eps={DBSCAN_EPS}, min_samples={DBSCAN_MIN_SAMPLES}")


    # 1. Find images
    image_paths = find_image_files(args.input_folder)
    if not image_paths:
        logging.info("No image files found in the specified folder.")
        exit(0)

    # 2. Extract face data (encodings)
    face_data = extract_face_data(image_paths)
    if not face_data:
        logging.info("No faces detected in any of the images.")
        exit(0)
```

```
170
171     # 3. Cluster faces
172     cluster_labels, clustered_face_data = cluster_faces(face_data)
173
174     # 4. Organize photos based on clusters
175     organize_photos_by_cluster(cluster_labels, clustered_face_data, args.output_folder)
176
177     logging.info("Face sorting process finished.")
```

Listing 3: Face Sorting Script (‘sort$_f aces.py$‘)

C Python Dependencies (`requirements.txt`)

```
face_recognition>=1.3.0
scikit-learn>=1.0.0
numpy>=1.19.0
opencv-python>=4.5.0
Pillow>=8.0.0
```

Listing 4: Python Dependencies (‘requirements.txt‘)

Note: Specific versions might vary. Use ‘pip freeze > requirements.txt‘ after installation for exact versions.

D Git Ignore Rules (`.gitignore`)

A standard Python ‘.gitignore‘ is recommended.

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# Distribution / packaging / Build artifacts
build/
dist/
*.egg-info/
*.egg

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# IDE / Editor specific
.idea/
.vscode/
*.sublime-*

# OS specific
.DS_Store
Thumbs.db
ehthumbs.db

# Test / Coverage reports
htmlcov/
.coverage
.pytest_cache/
nosetests.xml
coverage.xml

# Logs and databases
*.log
*.sqlite
*.sqlite3

# User-generated output (if not intended to be committed)
# sorted_output/ # Example output directory name
```

```
# LaTeX temporary files (if compiling blueprint in project dir)
*.aux
*.log
*.out
*.toc
*.synctex.gz
```

Listing 5: Example Git Ignore Rules (‘.gitignore‘)