Facial Recognition

Architecture Prototype

August 11, 2024

CS 661 / CRN 40600 - Python Programming Summer 2024 - Final Paper

Instructor: Professor Brian Harley

Group Members: Joshua Gottlieb (jg05394n@pace.edu), Cristian Bolanos (cb67828p@pace.edu), Raashil Aadhyanth (ra36066n@pace.edu)

Table of Contents

1. Introduction

Facial recognition is a popular and important product of the computer vision field and has a broad variety of potential use-cases across industries, including cybersecurity, fraud detection, and healthcare ("What is facial recognition", n.d.). As such, the development of an easy-to-use facial recognition architecture has many business applications.

The software takes a person's picture and compares it against a database of saved images. If it finds a match, some basic information about the found person is displayed, along with confidence metrics for the match. If the person is not in the database, the software reports that the person is not found.

The software provides a user interface, allowing the user to click to take a picture and initiate the facial recognition process, attempting to match the captured image against a collection of images stored in the database.

In this project, a facial recognition architecture prototype was developed as a proof-of-concept for deployment in real-world scenarios. The primary use case for the system is for determining if two photos taken under similar controlled conditions match. It is not intended to support real-time video facial recognition, or facial recognition of photos taken under unconstrained conditions. Additionally, in most usage scenarios, the facial recognition system is meant to supplement human screening rather than replace it.

The goal of the project is to demonstrate the ability to provide a user interface, a facial recognition engine, and a database access layer working in tandem. This is significant in that it demonstrates the viability of this architecture for future use in developing a more industrial-strength version of the software.

This paper will discuss the functional and non-functional requirements as well as the scope of the project. Then, a high-level design overview will be presented, covering each of the three components of the facial recognition architecture - the database access layer, the user interface layer, and the facial recognition engine.

After describing the implementation specifics of the project, an analysis of the testing performed to evaluate the effectiveness of the facial recognition engine, accompanied by the theoretical background necessary to understand the complex challenges of facial recognition, will be presented.

The paper concludes with a user manual and the lessons learned by the group during this project. An appendix details potential future enhancements to transform this facial recognition architecture from a proof-of-concept prototype to an industrial-strength software solution.

2. Requirements

2.1. Functional Requirements

2.1.1. Use Case

2.1.1.1. Take a Picture and Search

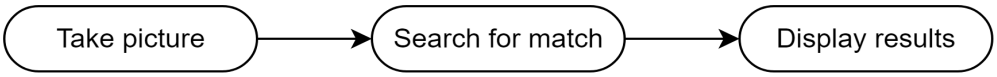

Figure 1. Top-Level Functionality for Project

- Prompt the user to start the process or to exit.
- If they choose to start, take a picture.
- Pass the picture to facial recognition.
- Receive found/not found results.
- If found, display database information for the found person, the picture that was taken, the picture on record in the database, and metrics regarding the accuracy of the search/confidence in the match.
- After viewing the information, the user can click OK, which returns to the prompt screen.

2.2. Features

- User interface
- Facial recognition search
- Display of search results, captured image, database matched image, and search metrics

2.3. Image Processing and Matching (Flow Diagram)

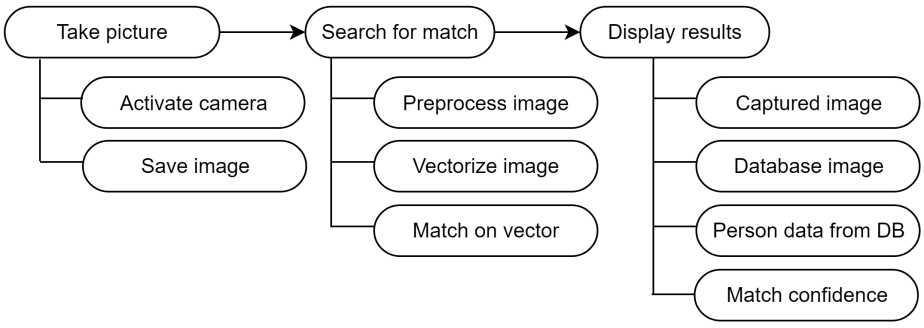

Figure 2. Expanded Functionality Flow Diagram

- The image is pre-processed to:

    - Find the face in the image.
    - Rotate the image to align the face in an upright orientation.
    - Crop away excess environmental image data so the image to search for will only contain important facial data.
- Generate vector of identifying characteristics
    - The input image is fed into the neural net to generate a vector that can be used for matching against the vectors for the images already stored in the database.
- Match against the known images
    - The vector generated in the previous step is compared to the known vectors stored in the database to find a match, if one exists.

2.4. Non-Functional Requirements

2.4.1. Speed

The user interface should be responsive and the search for a matching face should not take too long. Speed is not a primary consideration for this application, as it is not intended to be used for real-time facial recognition. See Section 8.4 for further discussion of speed enhancements.

2.4.2. Effectiveness

See Section 5. Testing for a detailed investigation into evaluating the project's effectiveness.

2.4.3. Scalability

Scalability is not required for the prototype produced by this project. However, by using a robust, scalable cloud-based data store for the database and local packages for the user interface and facial recognition engine, it should be possible to scale the system, if this ever becomes a requirement. See Section 8.2 for further discussion of scalability.

2.5. Project Scope

The project provides the following components and utilities:

- User interface to capture photo and drive the facial recognition engine
- Facial recognition engine
- Database storing person data, photos for each person, and vectorized data relevant to image matching
- User interface for displaying the results of the search
- Utility to initially load the database and generate vectors

See Section 8.1 for further discussion regarding additional functionality beyond the scope of this project.

3. Design

3.1. Design Overview

3.1.1. Components

- User interface layer
- Facial recognition engine
- Database access layer

3.1.2. Component Interfaces Diagram

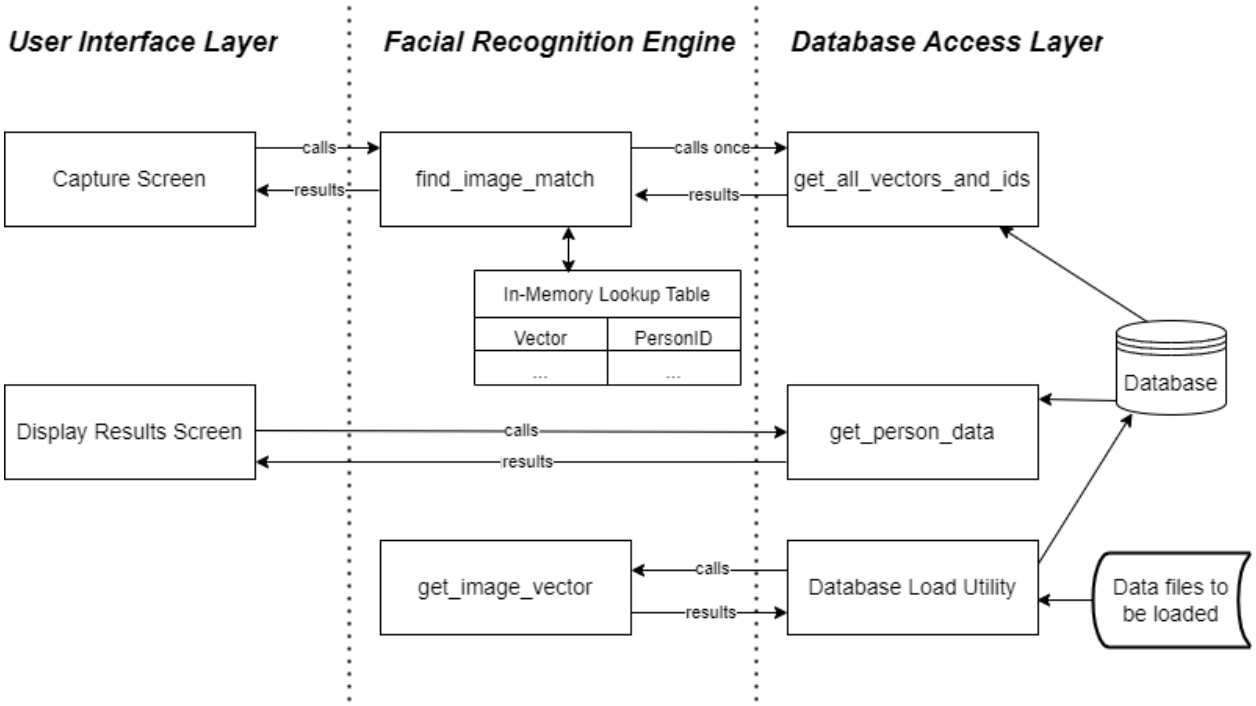

Figure 3. Component Interfaces Diagram

3.1.3. Communication Between Components

The components communicate via function calls from one component into another. These are the external interfaces exposed by each of the components as service APIs for use by the other components.

3.2. Database Access Layer

3.2.1. Database Fields

The following data needs to be stored for each person record in the database:

- PersonID - unique
- Reference photo
- A vector generated by the facial recognition engine, which will be used later for matching
- Additional data about the person, also retrieved from the database, such as Name, Address, Phone number, and Email address

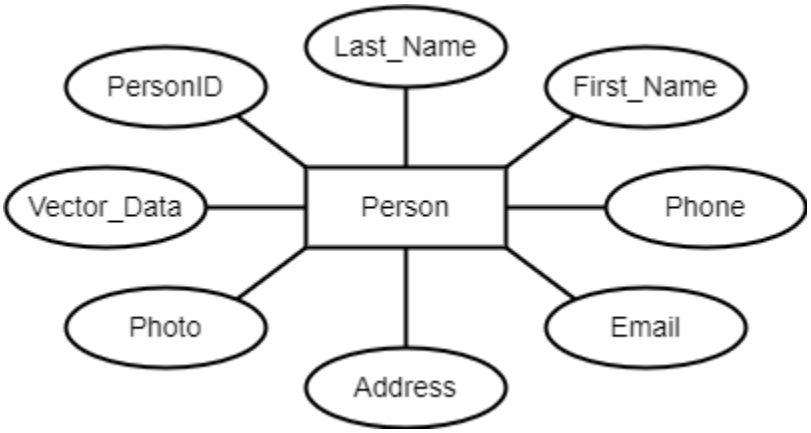

Figure 4. Entity Diagram representation of Person entity for database storage

3.2.2. Service Functions and Utilities

The database access layer provides a number of functions as service APIs for use by both the user interface and the facial recognition engine:

3.2.2.1. Functions Providing Services to the User Interface Layer

3.2.2.1.1. get_person_data

- Input:
  - PersonID
- Output: a data structure containing:
  - The text fields for the person
  - The reference photo of the person

3.2.2.2. Functions Providing Services to the Facial Recognition Engine

3.2.2.2.1. get_all_vectors_and_ids

- Input:
  - none
- Output:
  - Returns a data structure containing all the vectors and PersonIDs from the database, with one entry per person in the database. For each person, the following data is provided:
    - PersonID

        - Vector of identifying attributes

3.2.2.3. Program for Loading the Database

The database is loaded from files.

- The following files are needed:
    - A CSV file containing
        - PersonID
        - The other text fields that will be stored for each person
    - A set of image files, one for each row in the CSV file, where the file name is the PersonID.jpg (or .img, .png, using the appropriate extension).
- Processing:
    - For each row in the CSV file:
        - Parse the text fields
        - Generate the image file name as PersonID.jpg (or .img, .png, etc.)
        - Call the get_image_vector function in the facial recognition engine, passing the image file name.
            - Receive back the vector of identifying attributes for the image
        - Add the person record to the database, including the following data:
            - The text fields from the CSV
            - The image from the image file
            - The generated vector of identifying attributes

3.2.3. Database Layer Design

3.2.3.1. NoSql Mockup

The database can be implemented using a NoSQL database, as shown below:

Person (Object)	
PK	**PersonID**
	Last_Name
	First_Name
	Address
	Phone
	Email
	Photo
	Vector_Data

Figure 5. Mockup of Person Object stored in a NoSQL Database

3.3. User Interface layer

This has two screens, described below.

3.3.1. Capture Screen Mockup

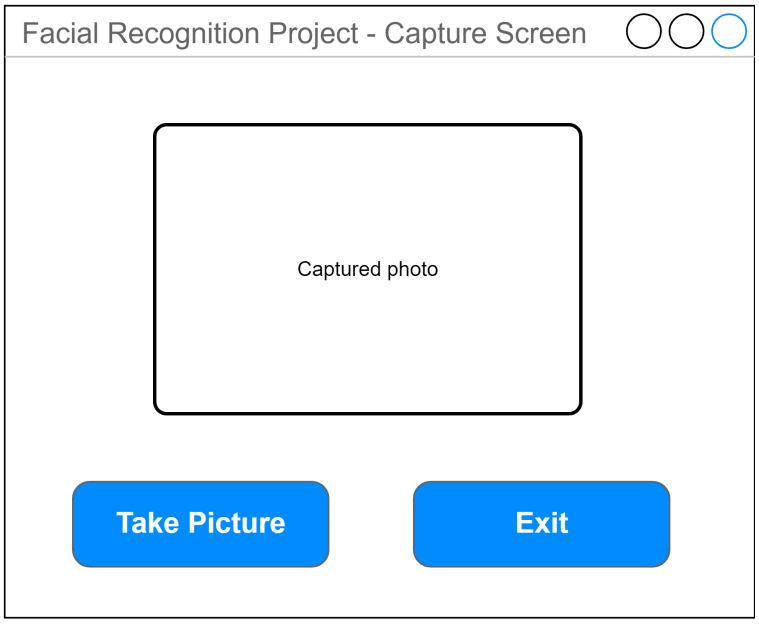

Figure 6. Capture Screen Mockup

- This allows the user to click to take their picture or to exit.
- When the user clicks to take their picture, activate the webcam to take their picture.
- Display the image on the screen and write the image to a file.
- Call a function in the facial recognition engine (find_image_match) to find a match for the image. Pass the image file name and dictionary of vectors as parameters.
- When find_image_match returns, display the Results Display Screen (see below).

3.3.2. Results Display Screen Mockup

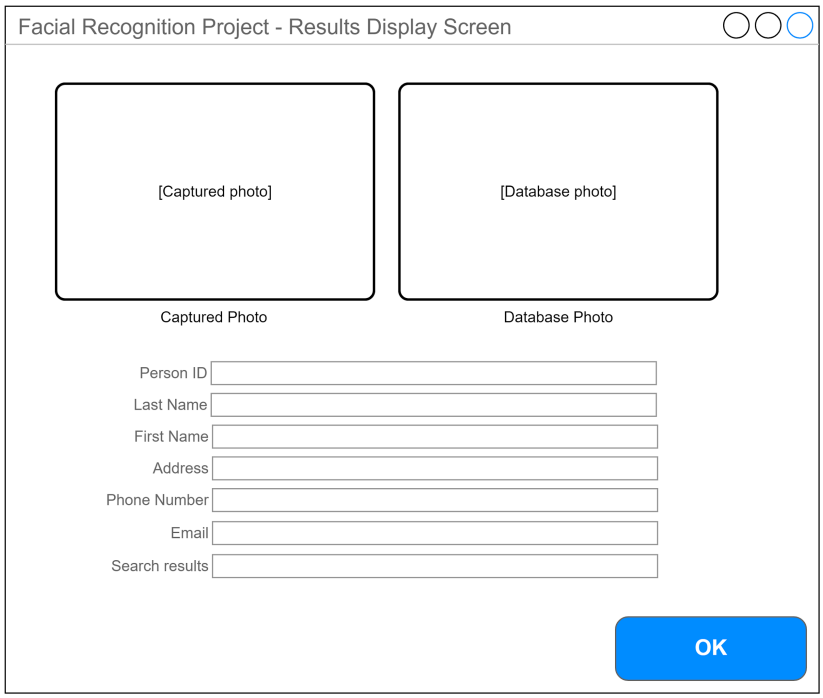

Figure 7. Results Display Screen Mockup

- If a match is not found, i.e., find_image_match returns an empty PersonID, the Results Display Screen will display an appropriate “not found” message in a dialog box. When the user clicks OK in the dialog box, the application will return to the initial Capture Screen.
- If a match is found, the Results Display Screen should call a function in the database access layer (get_person_data) to get the person data from the database and display the following:
    - The image that was captured by the webcam
    - The reference image retrieved from the database
    - Additional text fields data about the person, also retrieved from the database. See the “Database Requirements” section above or the screen mockup for what those fields will be.
    - The match details returned by find_image_match, shown in the mockup above as “Search results.”
- When the user clicks OK, the user interface will return to the initial capture screen, so that the user can once again take a new picture and repeat the process.

3.3.3. Screen Display Flowchart

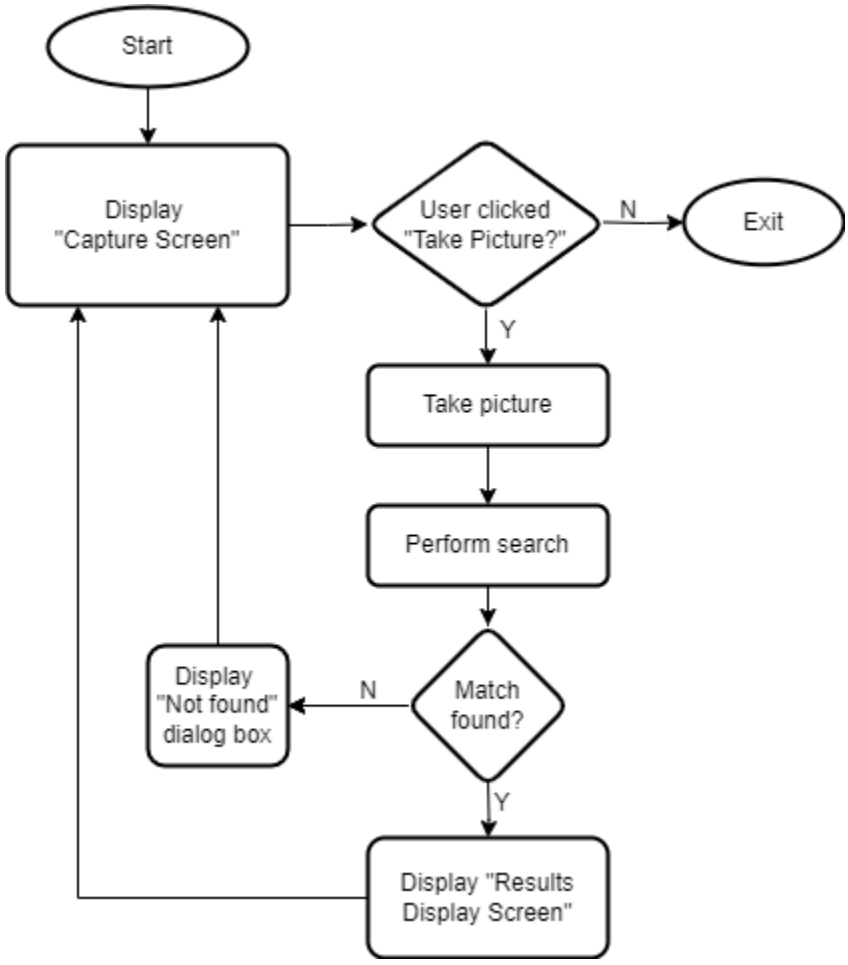

Figure 8. Screen Display Flowchart

3.4. Facial Recognition Engine

3.4.1. Service Functions

The facial recognition engine provides a number of functions as service APIs for use by the other components.

3.4.1.1. Functions Providing Services to the User Interface Layer

3.4.1.1.1. find_image_match

- Input:
  - Image file name, dictionary of database vectors
- Output:
  - If a match is found, returns:
    - matches: list of match tuples of form ('match_id', 'distance')

        - threshold: float representing highest threshold allowed for matching distances
        - metric: string representing metric used for distances
    - If a match is not found:
        - return a no_matches_found string containing details about the matching criteria

3.4.1.2. Functions Providing Services to the Database Access Layer

3.4.1.2.1. get_image_vector

- Input:
    - Image file name
- Output:
    - The vector of identifying attributes for the image
- Processing:
    - The facial recognition engine function will analyze the image and return a data structure containing the vector of identifying attributes for that image
- This function will be used by the database access layer database loading utility program.

4. Implementation

4.1. Development Process

4.1.1. Personnel Allocation

The team consists of three developers. Each one was responsible for one of the main components, which are:

- User interface (Cristian Bolanos)
- Image processing and facial recognition engine (Joshua Gottlieb)
- Database access layer (Raashil Aadhyanth)

Each developer was responsible for their own unit testing and documentation.

4.2. Image Preprocessing Steps

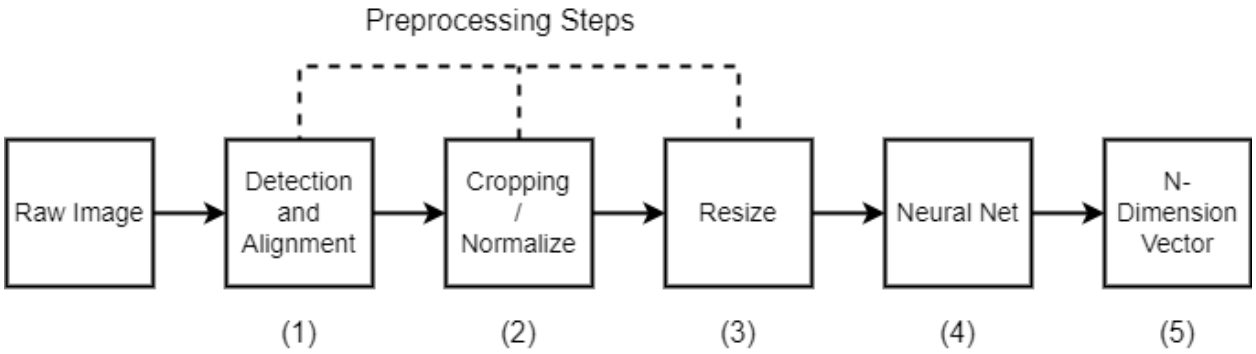

Figure 9. Preprocessing pipeline visualization

The image processing pipeline is outlined above. First, the raw image is fed into the preprocessing pipeline, which consists of (1) detecting and aligning the primary face in the image, (2) normalizing the image by cropping around the detected face boundaries, and then (3) resizing to the appropriate shape for input into the neural net. Then, the image is entered into the trained neural net (4), which produces an N-Dimension vector as its output (5).

Image preprocessing for the project requires 4 basic steps: facial detection, facial alignment, normalization, and resizing. Alignment and normalization are important aspects for improving the performance of neural nets, especially for the DeepFace neural net used by this project, as the DeepFace architecture presupposes alignment and similarity of local areas (Taigman et al., 2014).

For a more detailed discussion of facial preprocessing pipelines, see Section 8.6 in Appendix A.

4.2.1. Facial Preprocessing Backend

This project uses the pre-trained weights for DeepFace developed by Swarup Ghosh using the VGGFace2 dataset (Ghosh, 2019, Cao et al., n.d.). Ghosh used the dlib package for detection, alignment, and normalization. Other preprocessing libraries were tested, but best results occur by following Ghosh's preprocessing steps exactly, as demonstrated in Section 5.1.1 and 5.1.3.

Dlib ("dlib Homepage", n.d.) is a C++ library with a Python API. The primary pretrained recognizer that the facial recognition engine uses is the 5-point face landmark shape predictor (davisking, 2024). This predictor is extremely quick to execute and is excellent at aligning faces, although it has an extremely generous bounding box for facial detection, making it less ideal for normalization.

4.2.2. Full Image Preprocessing Pipeline Used by Ghosh

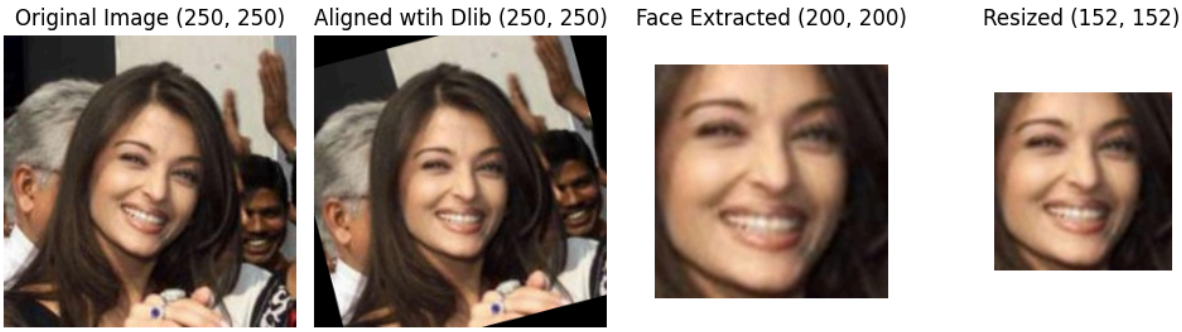

Figure 10. Full image preprocessing pipeline used by Ghosh

The general preprocessing steps employed by Ghosh are simple. First, dlib is used to detect the face in the image, and then the image is extracted using dlib's extract_image_chip function, producing an aligned and cropped image that is automatically resized to size (200, 200) using bilinear interpolation. Finally, OpenCV is used to reduce the size of the image using the INTER_AREA flag to (152, 152), as required by the DeepFace neural network input architecture.

4.3. Vectorization and Facial Recognition Implementation

4.3.1. Neural Net Architecture

The neural net from Facebook's DeepFace paper was utilized in this project. Previously, it was proposed that the neural net would be trained from the ground up but due to time constraints, Swarup Ghosh's pre-trained weights were utilized instead.

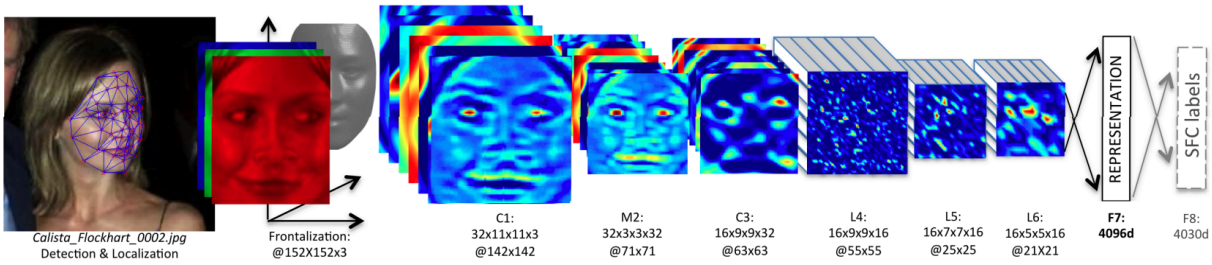

Figure 11. DeepFace neural net architecture (Taigman et al., 2014)

The above visualization is from the original DeepFace paper. The way the neural net functions is that it attempts to classify images as belonging to a specific label, in this case, a person's name. It accomplishes this by first passing images through a two dimensional convolutional layer (C1), followed by a two dimensional max-pooling layer (M2), and then a second two dimensional convolutional layer (C3). The purpose of these layers is to capture general low-level information, such as edge detections and textures. These then feed into a series of three locally-connected layers (L4, L5, and L6) which are designed to learn features about sub-regions of the input image, such as information about eyes or the mouth. The final two layers are fully-connected (F7 and F8) and are designed to capture information about positional correlations between features, such as position and shape of eyes. All of the layers except for F8 have rectified linear unit (ReLU) activations. F8 has a K-way softmax activation, where K is the number of classes of image.

4.3.2. Vectorization and Image Comparison

The final fully-connected layer from the DeepFace model is dropped when using the neural net for facial recognition purposes, as the facial recognition engine goal is not to classify a face as belonging to a particular label, but rather to create a way to compare two images for similarities. By using the output of F7, a 4,096 dimensional vector representation of the images is obtained. This vector representation is used for matching an input image against a reference set.

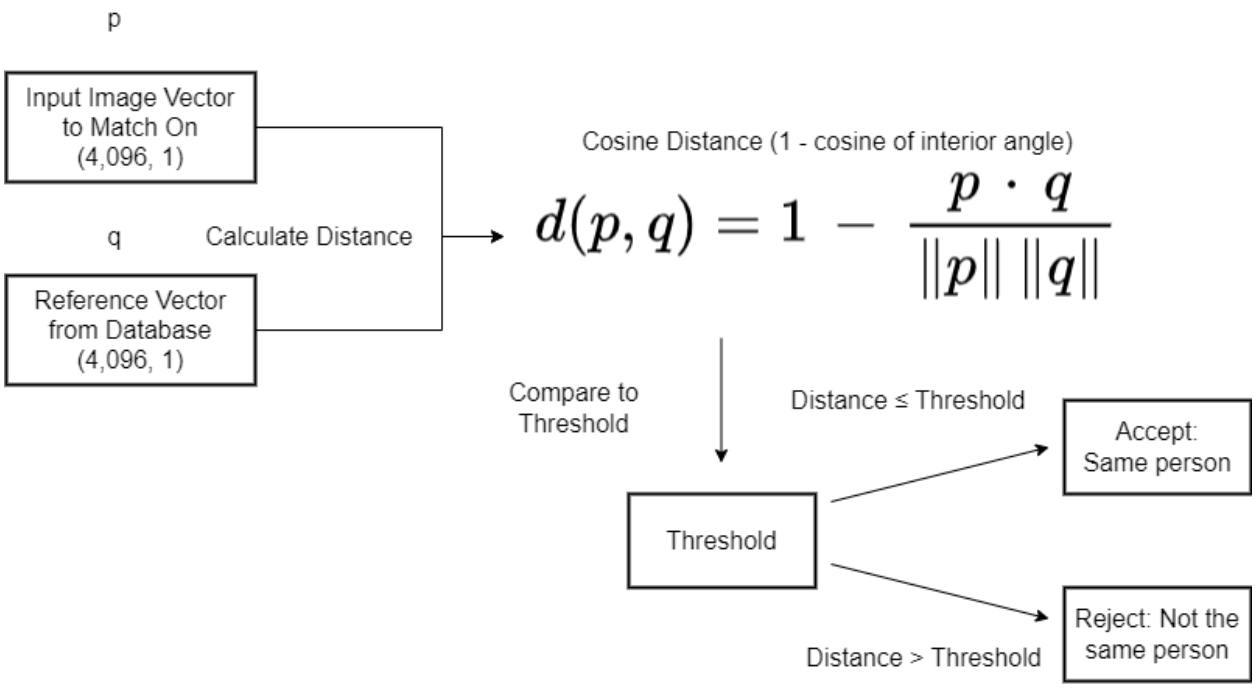

Figure 12. Vector comparison process, distance formula subimage from Eq. 1.2

Comparison of two images is made by comparing the distance between the two images. This distance provides a measurement of "closeness" for the two images, and a threshold is chosen to make the decision to accept or reject the images as representing the same person. Any chosen threshold has pros and cons, as it is nearly impossible for this thresholding process to guarantee perfect accuracy in matching and rejecting images. Analysis of distance metrics is discussed in Section 5.1.2, and the process for determining the appropriate threshold for classifying two vectors is covered under Section 5.1.4.

For code snippets of the facial recognition engine, see Section 4.6.3.

For code snippets of a Jupyter Notebooks implementation of the Facial Recognition Engine, see Section 4.6.6 and for a Usage Guide for the Jupyter Notebooks implementation, see Section 6.

4.4. UI Implementation

Kivy (Kivy Documentation, n.d.) is a free and open source Python App development framework created for the purpose of building cross-platform GUI apps.

The program uses Kivy's FloatLayout to structure the UI elements on the screen. The goal was to keep the UI as simple as possible to provide a user-friendly experience and focus on functionality, as a highly customized user-interface is unnecessary for a proof-of-concept prototype. A FloatLayout allows for precise control and placement of elements and was chosen over other layout types, such as BoxLayout, for its flexibility. The FloatLayout was used to control the placement of the matched image, which appears in the top-right corner of the interface.

The camera feed occupies most of the screen at the center of the application. The camera functionality uses OpenCV's cv2.VideoCapture(0) method to continuously provide real-time video frames to the screen. This ensures a smooth and responsive user experience, with the camera feed being updated at 30 frames per second through the method Clock.schedule_interval(), which comes from Kivy.

At the top of the camera feed, the program provides real-time feedback to the user. A status label displays messages such as "No image loaded" or "No match found" or displays the image file name of the matched user when a face recognition match occurs. If a match is identified, the program displays the user's name and also the matched image at the top-right corner of the screen. This is achieved through the display_matched_image() method, which is explicitly designed to handle the visual confirmation of the match. The method works by extracting the image path using the extract_image_filename() method, which isolates the image file name from the match result string. This file name is then used to locate the reference image in the dataset stored locally on disk, ensuring that the correct image is displayed.

The user interface also includes a "Capture" button at the bottom center of the screen. The button is strategically positioned at the bottom of the screen to ensure easy access while maintaining a clean and uncluttered interface. The "Capture" button lets users capture an image from the live camera feed, which is then processed to generate a face vector. To find a match, this vector is compared against stored vectors retrieved from a Firebase Firestore database, as outlined in Section 4.5. If a match is found, the program updates the labels and displays the corresponding image, providing immediate feedback to the user.

For code snippets of the User Interface Layer, see Section 4.6.5.

4.5. Database Implementation

The database layer is implemented with a two-part design. Image vectors are stored on Firebase Cloud Firestore, while the reference images for those vectors are stored locally on disk. With access to a paid account of Firebase, storage of the reference images could be moved entirely to the cloud.

Firebase Cloud Firestore (Firebase, n.d.) is a flexible, scalable NoSQL cloud database that is built on the Google Cloud infrastructure. Firebase offers a number of benefits for this project's proof-of-concept database access layer, including the ability to set security rules and the ability to easily upload and access data using Python through the firebase_admin Python library.

The Firebase Firestore is set up using a service account key for secure read/write access, enabling the application to store and retrieve face vectors. Each vector is linked to a unique identifier using UUID v4 (32 digits), allowing efficient retrieval and comparison during the facial recognition process. The seed_vectors.py script shown in Section 4.6.4.1 initializes the database with vectors from a directory of images.

4.5.1. Data Model

Cloud Firestore uses a document-collection model, which is well-suited for the facial recognition application.

4.5.1.1. Physical Model/Data Structure

```
firestore_root/
|
|-- vectors/
|       |-- Brian_Harley941bad87-86e4-4a41-b62e-aa319419728d/
|       |       |-- face_vector: [0.123, 0.456, ..., 0.789]
|       |
|       |-- Jane_Smith965411a9-6695-4038-95c6-777f45d233fd/
|       |       |-- face_vector: [0.987, 0.654, ..., 0.321]
```

4.5.2. User Collection

Each document in the users collection represents a single user. The document ID is a unique identifier for each user. Each user has a single field under the current design, representing the image vector of that user.

4.5.3. Reference Images

Due to time and monetary constraints, reference images are stored locally on disk. When the facial recognition layer matches on vectors retrieved from the database, the corresponding local image path is extracted from the Firebase ID and loaded into the user interface for display.

4.6. Code Snippets

For full code, see the referenced Github under src/ and src/modules (Gottlieb et al., 2024).

4.6.1. Image Preprocessing Code Snippets

4.6.1.1. ImagePreprocessor Class

```python
class ImagePreprocessor:
    def __init__(self, image_path, shape_predictor_path = './pretrained_models/shape_predictor_5_face_landmarks.dat'):
        self.image_path = image_path
        self.image = None
        self.aligned_image = None
        self.cropped_image = None
        self.resized_image = None
        self.a_backend = None
        self.c_backend = None
        self.dlib_details = None
        self.rf_details = None
        self.mp_details = None
        self.resize_dim = None
        self.pad = None
        self.pad_color = None
        self.error_code = None
        self.error_string = None
        self.facial_region = None
        self.shape_predictor_path = shape_predictor_path

        return
```

Code Snippet 1. ImagePreprocessor Class Constructor

```python
def load_image(self):
    self.image = cv2.imread(self.image_path)
    self.aligned_image = self.image.copy()
    self.cropped_image = self.image.copy()
    self.resized_image = self.image.copy()
    return

def get_image(self):
    self.load_image()
    return self.image

def get_aligned_image(self):
    return self.aligned_image

def get_cropped_image(self):
    return self.cropped_image

def get_resized_image(self):
    return self.resized_image
```

Code Snippet 2. ImagePreprocessor Utility Functions

```python
def _align_image(self, backend = ['dlib', 'retinaface', 'mediapipe']):
    """
    Uses designated backends to align an image along the centermost face.

    args:
        image: numpy array representing the input image
        backend: str or list of designated backends to use for aligning images in order of preference.
                 defaults to ['dlib', 'retinaface', 'mediapipe']
    """
    self._clear_backend_details()
    self._clear_error_code()
    # Format and confirm backend parameter
    backend = utils.confirm_valid_list_input(backend, valid_options = ['dlib', 'retinaface', 'mediapipe'])
    # Handling error code from backend confirmation, exiting due to invalid backend input
    if type(backend) == int:
        self.error_code = -1
        return
    # Loop through backends, when one succeeds, return the rotated image and the backend used.
    for b in backend:
        if b == 'dlib':
            self._get_dlib_faces(self.image)
            if self.dlib_details is None:
                continue
            self._dlib_align()
            self.a_backend = b
            return
```

```python
        if b == 'retinaface':
            self._get_retinaface_faces(self.image)
            if self.rf_details is None:
                continue
            self._retinaface_align()
            self.a_backend = b
            return

        if b == 'mediapipe':
            self._get_mediapipe_faces(self.image)
            if self.mp_details is None:
                continue
            self._mediapipe_align()
            self.a_backend = b
            return

    # If no backends found faces, set error_code to -1.
    self.error_code = -1
    return
```

Code Snippets 3 and 4. align_image()

```python
def _crop_image(self, backend = ['mediapipe', 'retinaface', 'dlib']):
    """
    Uses designated backends to crop an image around the centermost face.

    args:
        image: numpy array representing the input image
        backend: str or list of designated backends to use for cropping images in order of preference.
                 defaults to ['mediapipe', 'retinaface', 'dlib']
    """
    self._clear_backend_details()
    self._clear_error_code()
    # Format and confirm backend parameter
    backend = utils.confirm_valid_list_input(backend, valid_options = ['mediapipe', 'retinaface', 'dlib'])
    # Handling error code from backend confirmation, exiting due to invalid backend input
    if type(backend) == int:
        self.error_code = -1
        return
    # Loop through backends, when one succeeds, return the cropped image and the backend used.
    for b in backend:
        if b == 'mediapipe':
            self._get_mediapipe_faces(self.aligned_image)
            if self.mp_details is None:
                continue
            self._get_mediapipe_region(self.aligned_image)
            self.cropped_image = self.aligned_image[self.facial_region[2]:self.facial_region[3],
                                                    self.facial_region[0]:self.facial_region[1], :]
            self.c_backend = b
            return
```

```python
        if b == 'retinaface':
            self._get_retinaface_faces(self.aligned_image)
            if self.rf_details is None:
                continue
            self._get_retinaface_region()
            self.cropped_image = self.aligned_image[self.facial_region[2]:self.facial_region[3],
                                                    self.facial_region[0]:self.facial_region[1], :]
            self.c_backend = b
            return
        if b == 'dlib':
            self._get_dlib_faces(self.aligned_image)
            if self.dlib_details is None:
                continue
            self._get_dlib_region()
            self.cropped_image = self.aligned_image[self.facial_region[2]:self.facial_region[3],
                                                    self.facial_region[0]:self.facial_region[1], :]
            self.c_backend = b
            return

    # If no backends found faces, set error_code to -1.
    self.error_code = -1
    return
```

Code Snippets 5 and 6. crop_image()

```python
def _resize_image(self):
    """
    Resizes an image to desired end dimensions.
    Compresses on axes where image shape is larger than end dimension.
    Pads on axes images shape is smaller than end dimension, using specified padding color.

    args:
        image: numpy array representing the input image
        end_dim: (width, height) array-like representing numpy slice-form end dimensions
        pad: Whether to pad the image. If True, applies padding using color, otherwise uses interpolation for upscaling
        color: scalar or array-like to pass to np.full for padding image, only used if pad is True;
               default (0, 0, 0), tuples are in BGR format from OpenCV.

    return: numpy array representing resized image
    """
    self.resized_image = self.cropped_image.copy()

    if self.pad:
        # First, compress image along axes bigger than end dimension using cv2.resize with area interpolation.
        if self.resized_image.shape[0] > self.resize_dim[0] or self.resized_image.shape[1] > self.resize_dim[1]:
            self.resized_image = cv2.resize(self.resized_image, np.minimum(self.resized_image.shape[0:-1],
                                                                           self.resize_dim)[::-1],
                                            interpolation = cv2.INTER_AREA)
```

```python
    # Second, pad image using specified color.
    # We want to pad second so that the interpolation does not pick up the padded data.
    if self.resized_image.shape[0] < self.resize_dim[0] or self.resized_image.shape[1] < self.resize_dim[0]:
        old_height, old_width, channels = self.resized_image.shape
        new_height, new_width = np.maximum(self.resized_image.shape[0:-1], self.resize_dim)
        x_center = (new_width - old_width) // 2
        y_center = (new_height - old_height) // 2

        y_top = y_center + old_height
        x_right = x_center + old_width

        padded_image = np.full((new_height, new_width, channels), self.pad_color, dtype = np.uint8)

        padded_image[y_center:y_top, x_center:x_right] = self.resized_image

        self.resized_image = padded_image.copy()
else:
    if self.resized_image.shape[0] != self.resize_dim[0] or self.resized_image.shape[1] != self.resize_dim[1]:
        # If purely shrinking, use cv2.INTER_AREA, else, use cv2.INTER_CUBIC
        if self.resized_image.shape[0] > self.resize_dim[0] and self.resized_image.shape[1] > self.resize_dim[1]:
            interpolation = cv2.INTER_AREA
        else:
            interpolation = cv2.INTER_CUBIC
        self.resized_image = cv2.resize(self.resized_image, self.resize_dim[::-1], interpolation = interpolation)

return
```

Code Snippets 7 and 8. resize_image()

```python
def preprocess_image(self, alignment_backend = ['dlib', 'retinaface', 'mediapipe'],
                     crop_backend = ['mediapipe', 'retinaface', 'dlib'],
                     end_dim = (152, 152), color = (0, 0, 0), pad = True,
                     steps = ['align', 'crop', 'resize']):

    if type(steps) == str:
        if steps == 'full':
            steps = ['align', 'crop', 'resize']
        else:
            steps = [steps]

    steps = utils.confirm_valid_list_input(steps, valid_options = ['align', 'crop', 'resize'])

    if type(steps) == int:
        self.error_code = -1
        return

    self._clear_attributes()
    self.load_image()
    self.resize_dim = end_dim
    self.pad = pad
    self.pad_color = color
```

```python
for step in steps:
    self._clear_error_code()
    if step == 'align':
        self._align_image(backend = alignment_backend)
        if self.a_backend is None:
            if 'crop' not in steps:
                self.error_code = -1
                return
            else:
                # Checks if alignment backend and crop backend are the same
                # If all backends fail for alignment, they will fail for backend, so exit
                if sorted(alignment_backend) == sorted(crop_backend):
                    self.error_code = -1
                    return
        continue
    if step == 'crop':
        self._crop_image(backend = crop_backend)
        if self.c_backend is None:
            self.error_code = -1
            return
        continue
    if step == 'resize':
        self._resize_image()
        continue
return
```

Code Snippets 9 and 10. preprocess_image() (Used for Testing and Future Enhancements)

```python
def preprocess_ghosh(self):
    self._clear_attributes()
    self.load_image()
    self._get_dlib_faces(self.image)
    if self.dlib_details is None:
        self.error_code = -1
        return

    self.cropped_image = dlib.extract_image_chip(self.image, self.dlib_details)
    self.resize_dim = (152, 152)
    self.pad = True
    self.pad_color = (0, 0, 0)
    self._resize_image()
    return
```

Code Snippet 11. preprocess_ghosh() (Used in Facial Recognition Layer)

4.6.1.2. Preprocessing Utility Functions

```python
def get_angle_from_eyes(left_eye, right_eye, degrees = True):
    """
    Obtains the angle and direction of the line running through the (x, y) coordinates of two eyes.

    args:
        left_eye: (x, y) array-like containing pixel coordinates for the left eye
        right_eye: (x, y) array-like containing pixel coordinates for the right eye
        degree: If False, returns angle in radians. If True, returns angle in degrees.

    returns: angle between eyes and a direction (+1 or -1) for rotation instructions in the form (angle, direction)
    """

    # If the left and right eye have the same vertical y-values, the angle is zero.
    if right_eye[1] - left_eye[1] == 0:
        return 0, 1

    # Compute the cosine between the two coordinates using the dot product and norm definition of cosine.
    # Clips at plus or minus one for input into arccos function.
    cos_angle = np.clip(np.dot(left_eye,right_eye) / (np.linalg.norm(left_eye, 2) * np.linalg.norm(right_eye, 2)),
                        -1.0, 1.0)
    direction = math.copysign(1, (left_eye[1] - right_eye[1]))
    angle = np.arccos(cos_angle)
    if degrees:
        angle = np.rad2deg(angle)
    return angle, direction
```

Code Snippet 12. get_angle_from_eyes() (For use in calculating rotation in image alignment)

```python
def rotate_from_angle(image, center, angle, direction, degrees = True):
    """
    Rotates an image around specified center given an angle and direction.

    args:
        image: numpy array of image to be rotated
        center: (x, y) array-like containing pixel coordinates for the center around which to rotate
        angle: float or int representing the angle magnitude used to rotate
        direction: int representing the direction used to rotate;
                   values with magnitude != 1 will dilate or contract the resulting image
        degrees: If True, angle represents degrees. If False, angle represents radians.

    returns: numpy array representing the rotated image
    """
    if not degrees:
        angle = np.rad2deg(angle)

    rotation_matrix = cv2.getRotationMatrix2D(center, angle * direction, 1)
    rotated_image = cv2.warpAffine(image, rotation_matrix, image.shape[1::-1], flags = cv2.INTER_LINEAR)

    return rotated_image
```

Code Snippet 13. rotate_from_angle() (Rotates an image around a center of rotation)

```python
def get_distance_center(image_center, face_centers):
    """
    Calculates the distance to the image center from given face centers.

    args:
        image_center: (x, y) array-like containing pixel coordinates of the center of the image
        face_centers: 2-D array-like containing (x, y) pixel coordinates of the centers of faces

    return: 2-D array-like containing absolute distances in (x, y) form
    """
    return np.sum(np.abs(np.array(image_center) - np.array(face_centers)), axis = 1)

def get_index_center_face(image_center, face_centers):
    """
    Calculates the index of the centermost face in an image from face_centers array.

    args:
        image_center: (x, y) array-like containing pixel coordinates of the center of the image
        face_centers: 2-D array-like containing (x, y) pixel coordinates of the centers of faces

    return: int, index of the face_centers array corresponding to the most central face in the image
    """
    return np.argmin(get_distance_center(image_center, face_centers))
```

Code Snippet 14. get_distance_center() and get_index_center_face()
(For selecting centermost face from an image)

4.6.2. Vectorization Code Snippets

4.6.2.1. ImageVectorizer Class

```
class ImageVectorizer:
    def __init__(self, weights_path = './pretrained_models/VGGFace2_DeepFace_weights_val-0.9034.h5'):
        self.weights_path = weights_path
        self.model = None

    def _get_deepface_graph(self, input_shape = (152, 152, 3), num_classes = 8631):
        deepface = Sequential(name = 'DeepFace')
        deepface.add(Convolution2D(32, (11, 11), activation = relu, name = 'C1', input_shape = input_shape))
        deepface.add(MaxPooling2D(pool_size = 3, strides = 2, padding = 'same', name = 'M2'))
        deepface.add(Convolution2D(16, (9, 9), activation = relu, name = 'C3'))
        deepface.add(LocallyConnected2D(16, (9, 9), activation = relu, name = 'L4'))
        deepface.add(LocallyConnected2D(16, (7, 7), strides = 2, activation = relu, name = 'L5'))
        deepface.add(LocallyConnected2D(16, (5, 5), activation = relu, name = 'L6'))
        deepface.add(Flatten(name = 'F0'))
        deepface.add(Dense(4096, activation = relu, name = 'F7'))
        deepface.add(Dropout(rate = 0.5, name = 'D0'))
        deepface.add(Dense(num_classes, activation = softmax, name = 'F8'))

        self.model = deepface
        return
```

Code Snippet 15. ImageVectorizer Constructor and Tensorflow Model Graph Creation

```
def _load_model(self):
    self.model.load_weights(self.weights_path)
    return

# Expects a preprocessed image. Performs Tensorflow conversions needed to send the image through the model.
def _prep_for_deepface(self, image):
    return tf.expand_dims(tf.cast(image, tf.float32) / 255.0, axis = 0)

def initialize(self, input_shape = (152, 152, 3), num_classes = 8631):
    self._get_deepface_graph(input_shape = input_shape, num_classes = num_classes)
    self._load_model()

    self.model = Model(inputs = self.model.layers[0].input, outputs = self.model.layers[-3].output)

    dummy_arr = np.zeros((152, 152, 3))

    self.model.predict(self._prep_for_deepface(dummy_arr), verbose = 0)
    return
```

Code Snippet 16. Utility functions to load model weights and initialize the tensorflow graph by passing a blank image through it (greatly speeds up future vectorizations)

```
def vectorize_image(self, image_path, preprocess = True, preprocess_type = 'normal',
                    input_shape = (152, 152, 3), num_classes = 8631,
                    shape_predictor_path = './pretrained_models/shape_predictor_5_face_landmarks.dat', **kwargs):
    if self.model == None:
        self.initialize(input_shape = input_shape, num_classes = num_classes)

    image_wrapper = ImagePreprocessor(image_path, shape_predictor_path = shape_predictor_path)

    if preprocess:
        if preprocess_type == 'normal':
            image_wrapper.preprocess_image(**kwargs)
            input_image = self._prep_for_deepface(image_wrapper.get_resized_image())
        elif preprocess_type == 'ghosh':
            image_wrapper.preprocess_ghosh()
            input_image = self._prep_for_deepface(image_wrapper.get_resized_image())
        else:
            print(f"Unable to determine preprocessing type.\n"
                  + "Valid preprocessing types are 'normal' and 'ghosh'. Using unprocessed image.")
            input_image = self._prep_for_deepface(image_wrapper.get_image())
    else:
        input_image = self._prep_for_deepface(image_wrapper.get_image())

    return self.model.predict(input_image)[0]
```

Code Snippet 17. vectorize_image(), Vectorizes an Image by instantiating an ImagePreprocessor, processing the image, and then predicting using the DeepFace model.

4.6.3. Facial Recognition Layer Code Snippets

These are the interface functions to be called by the other components (UI, database load program).

```
def init_model(weights_path = './pretrained_models/VGGFace2_DeepFace_weights_val-0.9034.h5'):
    """
    Initializes a vectorizer for processing images.

    args:
        weights_path: str representing path to pre-trained model weights (.h5 file)

    returns: An initialized tensorflow.keras.Model object
    """
    model = ImageVectorizer(weights_path)
    model.initialize()
    return model
```

Code Snippet 18. init_model(), returns an initialized model for faster operation in other functions

```
def get_image_vector(image_path, model = None, preprocess_type = 'ghosh',
                     shape_predictor_path = './pretrained_models/shape_predictor_5_face_landmarks.dat', **kwargs):
    """
    Generates a vector from an image loaded from an image path.

    args:
        image_path: str representing path to image to be processed
        model: tensorflow.keras.Model object to use to generate predictions.
               If no model given, one will be initialized; however, work is much faster using a pre-initialized model.
        preprocess_type: str representing the type of preprocessing to perform. Valid parameters are 'normal' and 'ghosh'
        shape_predictor_path: str representing path to dlib pre-trained shape predictor.
        **kwargs: key-word arguments to be passed to the image preprocessor if preprocess_type is set to 'normal'

    returns: np.array representing the image vector
    """
    if model is None:
        model = init_model()

    if preprocess_type != 'normal' and preprocess_type != 'ghosh':
        print(f'Invalid preprocess type specified, defaulting to Ghosh preprocessing.')
        preprocess_type = 'ghosh'

    vector = model.vectorize_image(image_path, preprocess_type = preprocess_type,
                                   shape_predictor_path = shape_predictor_path, **kwargs)

    return vector
```

Code Snippet 19. get_image_vector(), generates a vector from an input image path

By default, initializes a model, but using a pre-instantiated model is significantly faster. Default behavior is to use Ghosh preprocessing, although generalized preprocessing as outlined in Section 8.6.1 can be implemented using preprocess_type = 'normal'.

```
def find_image_match(image_path, db_vectors, model = None, metric = 'cos', preprocess_type = 'ghosh',
                     threshold_strategy = 'matthews_cc', threshold_strictness = 2, match_num = 1,
                     shape_predictor_path = './pretrained_models/shape_predictor_5_face_landmarks.dat'):
    """
    Finds the best matching image to an image loaded from an image path in a database by converting the image to a vector.
    Compares the vector to database vectors and returns the database id of the closest vector and matching information.

    args:
        image_path: str representing the image to be processed
        db_vectors: dict of {id:vector} key-value pairs to compare against
        model: tensorflow.keras.Model object to use to generate predictions.
               If no model given, one will be initialized; however, work is much faster using a pre-initialized model.
        metric: str representing the distance metric to use. Valid parameters are 'cos' and 'l2' (not implemented).
        threshold_strategy: str representing the criterion to maximize when selecting the threshold. Criterion listed below.
            'matthews_cc': Matthews Correlation Coefficient. Attempts to balance predictive recall on both classes with
                           predictive precision on both classes. Default criterion. Threshold = 0.269697
            'f1_score': F1-Measure. Attempts to balance precision and recall on the positive class (images match).
                        Less precise than matthew_cc, with slightly better recall. Threshold = 0.293939
            'balanced_acc': Balanced Accuracy. Maximizes accuracy across class imbalances. Prediction favors matching the
                            underlying distribution of the data, which tends to favor the negative class (images do not
                            match) and has low precision, which may result in more false positives. Threshold = 0.447475
            'precision': Precision; positive predictive value. Emphasizes that predictions on the positive class (images
                         match) should come from the positive class, reducing the number of false positives. High
                         precision naturally reduces the ability to identify any positive predictions, which may lead to
                         low identification rate for matching images; i.e., low recall. Threshold = 0.253535
```

```
        threshold_strictness: int representing the number of successively less strict thresholds to test.
                              Valid options range from 1 to 5. Used to allow looser matches along similar threshold criterion.
        match_num: int representing the top N matches to return. Default 1.
        shape_predictor_path: str representing path to dlib pre-trained shape predictor.

    returns:
        matches: list of match tuples of form ('match_id', 'distance')
        threshold: float representing highest threshold allowed for matching distances
        metric: string representing metric used for distances
    """

    # Set thresholds based on selected strategy
    if threshold_strategy == 'matthews_cc':
        thresholds = [0.269697, 0.277778, 0.285859, 0.293939, 0.302020]
    elif threshold_strategy == 'f1_score':
        thresholds = [0.293939, 0.310101, 0.318182, 0.326263, 0.334343]
    elif threshold_strategy == 'balanced_acc':
        thresholds = [0.447475, 0.455556, 0.463636, 0.471717, 0.479798]
    elif threshold_strategy == 'precision':
        thresholds = [0.253535, 0.261616, 0.269697, 0.277778, 0.285859]
    else:
        print("threshold_strategy should have a value among 'matthews_cc', 'f1_score', 'balanced_acc', or 'precision'\n"
              + "Defaulting to 'matthews_cc' threshold_strategy.")
        thresholds = [0.269697, 0.277778, 0.285859, 0.293939, 0.302020]
```

```
# Check threshold strictness to determine max number of thresholds to test against
if not 1 <= threshold_strictness <= 5:
    print("threshold_strictness should have an integer value between [1, 5]. Defaulting to 2.")
    threshold_strictness = 2

# Check number of matches to return
if match_num <= 0:
    match_num = 1

vector = get_image_vector(image_path, model = model,
                          preprocess_type = preprocess_type, shape_predictor_path = shape_predictor_path)

# Initialized list to capture ids and distances
ids = []
distances = []

# Using dictionary of reference vectors from database API function
for key, value in db_vectors.items():
    if metric == 'cos':
        distance = utils.cosine_distance(vector, value)
    elif metric == 'l2':
        distance = utils.l2_distance(vector, value)
    else:
        print("Invalid metric. Defaulting to cosine distances.")
        metric = 'cos'
        distance = utils.cosine_distance(vector, value)
```

```
    ids.append(key)
    distances.append(distance)

ids = np.array(ids)
distances = np.array(distances)
# Select match_num indices from the match_num smallest distances
possible_indices = np.argpartition(distances, match_num)[:match_num]

match_indices = np.nonzero(distances[possible_indices] < thresholds[threshold_strictness - 1])

# If no matches found below threshold, exit with comparison evaluations
if len(match_indices[0]) == 0:
    no_match_string = f"No matching images found.\nDistance threshold: {thresholds[threshold_strictness - 1]}"\
                      + f"\nLowest Distance: {min(distances[possible_indices])}"
    return no_match_string

# Otherwise, return all matches (up to match_num) found, sorted by lowest distance
match_ids = ids[possible_indices[match_indices[0]]]
match_distances = distances[possible_indices[match_indices[0]]]
return sorted(tuple(zip(match_ids, match_distances)), key = lambda x: x[1]), thresholds[threshold_strictness - 1], metric
```

Code Snippets 20, 21, 22, and 23. find_image_match()

find_image_match() is the backbone of the Facial Recognition Layer. Default threshold_strategy is to use MCC, as outlined in Section 5.1.4.6, with a threshold_strictness of 2 to allow for more matches. By default, match_num is 1 to match only on the closest image. It works by vectorizing the image from the given image path, then compares the vector generated from the input image against all vectors in the passed-in vector dictionary. Selects up to the closest N matches as specified by match_num if distances are below the required threshold and returns matching information. If no matches are found, the function returns a string with details of the search.

4.6.4. Database Access Layer Code Snippets

The following code snippet demonstrates how vectors are seeded into Firebase Firestore.

4.6.4.1. seed_vectors.py

```python
import firebase_admin
from firebase_admin import credentials, firestore
import uuid
from modules.face_api import init_model, get_image_vector
import os
import cv2
import tensorflow as tf
import numpy as np
import shutil

# Initialize Firestore DB
cred = credentials.Certificate("serviceAccountKey.json")
firebase_admin.initialize_app(cred)
db = firestore.client()
# Path to the folder containing images
image_folder = '../data/dummy_dataset/images_to_vectorize'
destination_folder = '../data/dummy_dataset/input_images'
# Reference to your Firestore collection
collection_ref = db.collection('vectors')
# Model for vectorization
model = init_model('../pretrained_models/VGGFace2_DeepFace_weights_val-0.9034.h5')
# Get image_names
image_names = os.listdir(image_folder)
```

```python
# Loop through each image in the folder
for i, image_name in enumerate(image_names):
    image_path = os.path.join(image_folder, image_name)

    # Get the image vector
    image_vector = get_image_vector(image_path, model = model,
                                    shape_predictor_path = '../pretrained_models/shape_predictor_5_face_landmarks.dat')

    # Convert numpy array to list
    image_vector_list = image_vector.tolist()

    # Generate a unique ID for the image
    unique_id = str(uuid.uuid4())

    # Create a dictionary to store in Firestore
    data = {
        image_name + unique_id: image_vector_list,
    }

    # Add the data to Firestore
    collection_ref.document(unique_id).set(data)

    # Move the processed image to the destination folder
    shutil.move(image_path, os.path.join(destination_folder, image_name))

print("All images have been processed and stored in Firestore.")
```

Code Snippets 24 and 25. seed_vectors.py

4.6.5. UI Code Snippets

```python
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.image import Image
from kivy.uix.label import Label
from kivy.uix.textinput import TextInput
from kivy.uix.popup import Popup
from kivy.clock import Clock
from kivy.graphics.texture import Texture
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.behaviors import ButtonBehavior
from kivy.graphics import Color, RoundedRectangle
import cv2
import firebase_admin
from firebase_admin import credentials, firestore
import numpy as np
from face_api import get_image_vector, init_model
import os
import re


class RoundedButton(ButtonBehavior, Label):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (0, 0, 0, 0)
        self.background_normal = ''
        self.bind(size=self.update_canvas, pos=self.update_canvas)

    def update_canvas(self, *args):
        self.canvas.before.clear()
        with self.canvas.before:
            Color(0.2, 0.6, 1, 1)  # Set button color (light blue)
            RoundedRectangle(pos=self.pos, size=self.size, radius=[20, ])

    def on_press(self):
        self.canvas.before.clear()
        with self.canvas.before:
            Color(0.1, 0.5, 0.9, 1)  # Slightly darker when pressed
            RoundedRectangle(pos=self.pos, size=self.size, radius=[20, ])

    def on_release(self):
        self.update_canvas()
```

Code Snippet 26. Library imports and RoundedButton Class

```python
class DeepFaceApp(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Initialize Firebase
        cred = credentials.Certificate("serviceAccountKey.json")
        firebase_admin.initialize_app(cred)
        self.db = firestore.client()
        self.model = init_model()
        self.match_result = None

    def build(self):
        self.layout = FloatLayout()

        # Camera view
        self.image = Image(size_hint=(1, 1), pos_hint={'center_x': 0.5, 'center_y': 0.5})
        self.layout.add_widget(self.image)

        # Label for status
        self.label = Label(text='No image loaded', size_hint=(0.8, 0.1), pos_hint={'center_x': 0.5, 'top': 1})
        self.layout.add_widget(self.label)

        # Match result label
        self.match_label = Label(text='', size_hint=(0.8, 0.1), pos_hint={'center_x': 0.5, 'y': 0.05})
        # self.layout.add_widget(self.match_label)

        # Matched image view
        self.matched_image = Image(size_hint=(0.3, 0.3), pos_hint={'center_x': 0.5, 'y': 0.35})

        # Capture button
        capture_button = RoundedButton(text='Capture Image', size_hint=(0.2, 0.1), pos_hint={'center_x': 0.3, 'y': 0.02})
        capture_button.bind(on_press=self.capture_image)

        # Sign up button
        signup_button = RoundedButton(text='Sign Up', size_hint=(0.2, 0.1), pos_hint={'center_x': 0.7, 'y': 0.02})
        signup_button.bind(on_press=self.show_signup_popup)

        self.layout.add_widget(capture_button)
        self.layout.add_widget(signup_button)
```

Code Snippet 27. DeepFaceApp Class Constructor and most of build()

```
        # Start the camera automatically
        self.start_camera()
        return self.layout

    def start_camera(self):
        self.capture = cv2.VideoCapture(0)
        Clock.schedule_interval(self.update, 1.0 / 30.0)

    def update(self, dt):
        ret, frame = self.capture.read()
        if ret:
            # Convert it to texture
            buf = cv2.flip(frame, 0).tobytes()
            image_texture = Texture.create(size=(frame.shape[1], frame.shape[0]), colorfmt='bgr')
            image_texture.blit_buffer(buf, colorfmt='bgr', bufferfmt='ubyte')
            # Display image from the texture
            self.image.texture = image_texture

    def capture_image(self, instance):
        if hasattr(self, 'capture'):
            ret, frame = self.capture.read()
            if ret:
                # Save the captured image
                cv2.imwrite("captured_image.jpg", frame)
                self.process_captured_image()
            else:
                self.label.text = 'Failed to capture image'
        else:
            self.start_camera()
```

Code Snippet 28. Rest of build() and all of start_camera(), update(), and capture_image()

```
def process_captured_image(self):
    # Get vector for captured image
    captured_vector = get_image_vector("captured_image.jpg", self.model)
    # Match with vectors in Firestore
    self.match_result = self.match_vector(captured_vector)
    if self.match_result:
        self.label.text = f"Match found: {self.match_result}"
        self.match_label.text = f"Match found: {self.match_result}"
        self.display_matched_image(self.match_result)
    else:
        self.label.text = "No match found"
        self.match_label.text = "No match found"
        self.clear_matched_image()

def clear_matched_image(self):
    # Clear the texture of the matched image
    self.matched_image.texture = None
```

Code Snippet 29. Functions to match images and write to screen

```
def match_vector(self, captured_vector):
    threshold = 0.6  # Adjust this threshold as needed
    # Get all documents from the 'vectors' collection
    docs = self.db.collection('vectors').get()
    for doc in docs:
        data = doc.to_dict()
        for key, stored_vector in data.items():
            similarity = self.cosine_similarity(captured_vector, np.array(stored_vector))
            if similarity > threshold:
                return key.split('+')[0]  # Return the image name
    return None

def cosine_similarity(self, a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

Code Snippet 30. Functions to match images

```python
def extract_image_filename(text):
    # Split the text into parts
    parts = text.split()
    # Iterate over each part to find the one ending with an image file extension
    for part in parts:
        if part.endswith('.jpg') or part.endswith('.png') or part.endswith('.jpeg'):
            return part
    # Return None if no image file name is found
    return None


def extract_image_filename(self, text):
    # Regular expression to match the image file name ending with .jpg
    match = re.search(r'([\w\-]+\.jpg)', text)
    if match:
        return match.group(1)
    return None

def display_matched_image(self, image_name):
    print('image_name', image_name)
    image_file_name = self.extract_image_filename(image_name)
    image_path = os.path.join('../data/dummy_dataset/input_images', image_file_name)
    if os.path.exists(image_path):
        matched_frame = cv2.imread(image_path)
        buf = cv2.flip(matched_frame, 0).tobytes()
        image_texture = Texture.create(size=(matched_frame.shape[1], matched_frame.shape[0]), colorfmt='bgr')
        image_texture.blit_buffer(buf, colorfmt='bgr', bufferfmt='ubyte')
        self.matched_image.texture = image_texture
        self.matched_image.pos_hint = {'right': 1, 'top': 1}

        # Check if the widget already has a parent
        if self.matched_image.parent:
            self.matched_image.parent.remove_widget(self.matched_image)

        self.layout.add_widget(self.matched_image)
```

Code Snippet 31. Functions to find and display matched reference image

```
def show_signup_popup(self, instance):
    popup_layout = BoxLayout(orientation='vertical', spacing=10, padding=10)
    self.name_input = TextInput(hint_text='Enter name')
    self.email_input = TextInput(hint_text='Enter email')
    popup_layout.add_widget(self.name_input)
    popup_layout.add_widget(self.email_input)
    submit_button = RoundedButton(text='Submit', size_hint=(1, 0.3))
    submit_button.bind(on_press=self.submit_signup)
    popup_layout.add_widget(submit_button)
    self.popup = Popup(title='Sign Up', content=popup_layout, size_hint=(0.8, 0.4))
    self.popup.open()

def submit_signup(self, instance):
    name = self.name_input.text
    email = self.email_input.text
    if name and email:
        self.label.text = f'Signed up: {name}, {email}'
        self.popup.dismiss()
    else:
        self.label.text = 'Please enter both name and email'
```

Code Snippet 32. Functions related to sign-up button

4.6.6. Jupyter Notebooks Implementation Code Snippets

Code snippets of the Jupyter Notebook used for demonstration and testing of the facial recognition engine.

```
# Library and Personal Module Imports
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import src.modules.face_api as face_api
```

```
# Initialize a model to pass into future functions for fast vectorization.
model = face_api.init_model()
```

Code Snippet 33. Library Imports and Model Initialization

```
# Specify threshold strategy: default matthews_cc, least strict is balanced_acc, see face_api.py for more details
# Specify threshold strictness: 1 to 5 from most to least strict
# Specify maximum number of matches to return if distances are less than threshold, positive integer
threshold_strategy = 'balanced_acc'
threshold_strictness = 5
match_num = 3

# threshold_strategy = 'matthews_cc'
# threshold_strictness = 2
# match_num = 1
```

Code Snippet 34. Facial Recognition Parameter Settings

```
# Specify reference directories for images and vectors to compare against.
# Specify number of digits as suffix for images
# reference_dir = './data/dummy_dataset/reference_images'
# vector_dir = './data/dummy_dataset/reference_vectors'
# num_digits = 4

reference_dir = './data/personal_images_batch2/reference_images'
vector_dir = './data/personal_images_batch2/reference_vectors'
num_digits = 2

vector_dict = face_api.load_vector_dict(vector_dir, num_digits = num_digits)
```

```
# Specify input image directory and name to check against.
# image_dir = './data/dummy_dataset/input_images'
# image_name = 'Wayne_Gretzky_0002.jpg'
# image_name = 'Sarah_Michelle_Gellar_0002.jpg'

image_dir = './data/personal_images_batch2/input_images'

# image_name = 'Jack-04.jpg'
# image_name = 'Adam-02.jpg'
# image_name = 'Ed-02.jpg'
image_name = 'Dan1-02.jpg'

image_path = os.path.join(image_dir, image_name)
```

Code Snippet 35. Reference and Vector Directories, Numbering Schema, and Input Image Path

```
# Load in base image for inspection
base_image = cv2.imread(image_path)[:, :, ::-1]
```

```
# Show base image
fig, ax = plt.subplots(figsize = (4, 3))

ax.imshow(base_image);

ax.set_title(f'{image_name.split(".")[0]}')
ax.set_xlabel('Input Image');
ax.set_xticks([]);
ax.set_yticks([]);
ax.grid(False);

plt.tight_layout();
```

Code Snippet 36. Load in base image and display for verification.

```
# Perform Matching
match_details = face_api.find_image_match(image_path, vector_dict, model = model,
                                          threshold_strategy = threshold_strategy,
                                          threshold_strictness = threshold_strictness,
                                          match_num = match_num)
```

```
# If no matches are found, print match details, else show matches found
if type(match_details) == str:
    print(match_details)
else:
    # Display results
    # Split results into fields
    matches, threshold, metric = match_details

    # Number of subplots
    n = len(matches) + 1

    # Load matched images
    matched_images = [cv2.imread(os.path.join(reference_dir, x[0] + '.jpg'))[:, :, ::-1] for x in matches]

    # Create subplots for base image and all matched images
    fig, ax = plt.subplots(1, n, figsize = (round(n * 2.5), 4))

    # Display base image
    ax[0].imshow(base_image);
    ax[0].set_title(f'{image_name.split(".")[0]}');
    ax[0].set_xlabel('Input Image');

    # Turn off axis numbers
    for a in ax:
        a.set_xticks([]);
        a.set_yticks([]);
        a.grid(False);

    # Display matched images
    for i in range(len(matched_images)):
        ax[i + 1].imshow(matched_images[i]);
        ax[i + 1].set_title(f'{matches[i][0]}');
        ax[i + 1].set_xlabel(f'Distance: {matches[i][1]:.03f}');

    plt.tight_layout();
```

Code Snippet 37. Run find_image_match() and Display Matching Results

4.7. Libraries and Tools Used

For more details on library requirements, see Section 6.

- Python 3.9
- User Interface
    - Kivy 2.3.0
- Database access packages
    - Firebase Admin Python SDK
        - firebase_admin 6.5.0
- Image processing libraries and tools
    - Numpy and OpenCV for image manipulation
        - numpy 1.17.4
        - opencv_python 4.8.0.76
    - dlib for face detection, alignment, and normalization of facial recognition
        - dlib 19.24.4
    - RetinaFace and MediaPipe, unused in main application but used for image preprocessing testing and in investigating future enhancements
        - mediapipe 0.10.11
        - retina_face 0.0.17
- Neural network libraries for pre-trained model
    - Tensorflow
        - tensorflow 2.13.1
- Visualization
    - Matplotlib, Seaborn, Scikit-learn
- Other packages
    - pickle, os, math, json, re, pandas
        - pandas 1.5.2

5. Testing

5.1. Facial Recognizer Evaluation and Threshold Fine-Tuning

5.1.1. Pipelines Evaluated

Using the DeepFace neural network with pre-trained weights by Swarup Ghosh, the Labeled Faces in the Wild (LFW) dataset (Huang et al., 2007) was processed using three different pipelines in order to test the effectiveness of the facial recognizer.

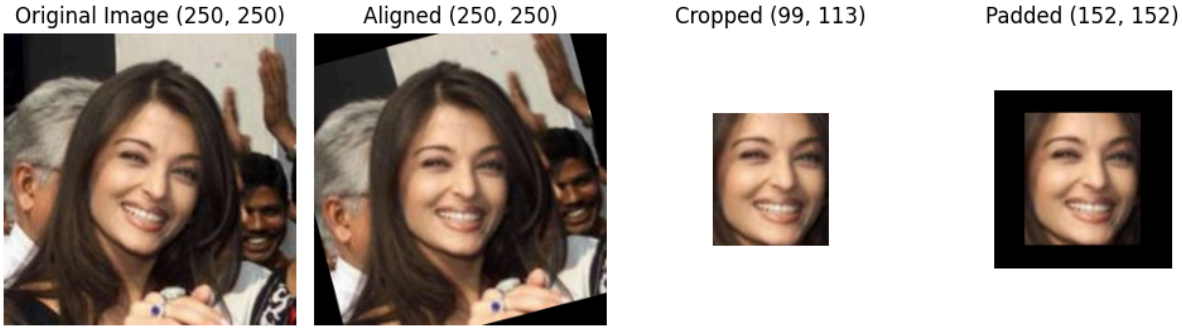

Figure 13. General Image Preprocessing Pipeline Steps: 1) Detect and Align, 2) Crop, 3) Resize

As a reminder, the general steps required for image preprocessing for input into a neural network are to detect the face in the image, align the image, extract the facial region by cropping, and then to finally resize the image to the desired end dimensions. There are many different ways to approach the tasks of facial alignment, extraction, and resizing.

Figure 14. Three-tested pipelines: 1) MediaPipe/pad, 2) MediaPipe/stretch, 3) dlib/shrink

The first and second pipelines involved using the mixed backend process of using dlib to align the face and using MediaPipe to extract the facial regions, as outlined in Section 8.6.1. The only differences between these two pipelines is that the first pipeline would pad the final images with zeroes to reach the desired end dimensions of (152, 152, 3), while the second pipeline would use OpenCV's bicubic interpolation to stretch the image to the final shape. Both processes

would use OpenCV's INTER_AREA interpolation in the event that downscaling was required. The third and final pipeline was the reimplementation of the original Ghosh preprocessing pipeline, which simply used dlib to align and extract the facial regions, as described in Section 4.2. dlib's default behavior is to resize the final image to (200, 200, 3) using bilinear interpolation, so the final image was shrunk using OpenCV's INTER_AREA interpolation to obtain the final preprocessed image. Going forward, these pipelines will be referenced as Padded, Stretched, and Ghosh.

The entire LFW dataset was processed using each of these pipelines, and then the distances between each image were calculated. These distances were then grouped into intra-label or inter-label groups, representing the distances between images representing the same or different canonical individual, respectively.

5.1.2. Distance Metrics Tested

This process was performed twice; using the Euclidean (L_2) distance and using cosine distance. Euclidean distance (Eq. 1.1) is defined as the square root of the sum of square distances between vector components. Euclidean distance is always positive, and ranges from 0 to positive infinity, with distances closer to 0 indicating more similarity between vectors. Cosine distance (Eq 1.2) is defined as 1 - cosine similarity (Eq 1.3), which is itself defined as the cosine of the angle between the two vectors. Cosine similarity ranges from -1 to 1, and thus so does cosine distance; however, cosine distance is only negative when vector components are able to take on negative values, which never occurs during the neural net vectorization process. This means that cosine distance is limited to the range between 0 and 1, inclusive. For those who remember trigonometry, the cosine of the angle is maximal (1) when the angle between vectors is 0, meaning that the lower the cosine distance, the more similar the vectors are to each other. Consider the following image:

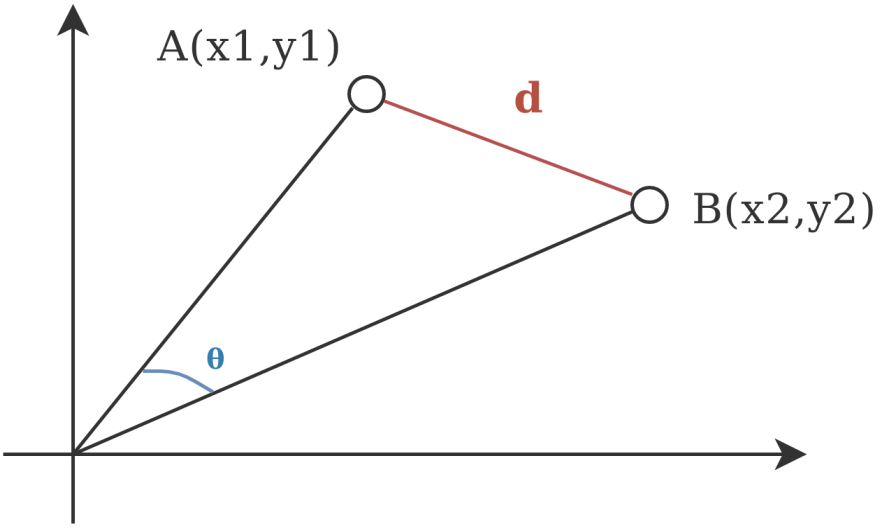

Figure 15. Cosine similarity vs. Euclidean distance (Emmery, 2017)

The distance *d* represents the Euclidean distance, while the cosine distance is represented by 1 - cos(θ). Why does choice of distance matter, and how should one choose? From a theoretical standpoint, Euclidean distances illustrate whether two vectors exist near each other in coordinate-space, while cosine distances illustrate whether two vectors lie in the same direction, as measured from the origin, in coordinate-space. Euclidean distances tend to be more useful when magnitude of the vector matters, while cosine distances tend to be more useful when magnitude of the vector does not matter. Without a priori knowledge, it is best to investigate both distance metrics.

5.1.3. Distributions Under Different Preprocessing Pipelines and Distance Metrics

A variety of different preprocessing pipelines were tested. For the sake of readability, all graphs of non-Ghosh pipelines can be found in Appendix C, rather than inline.

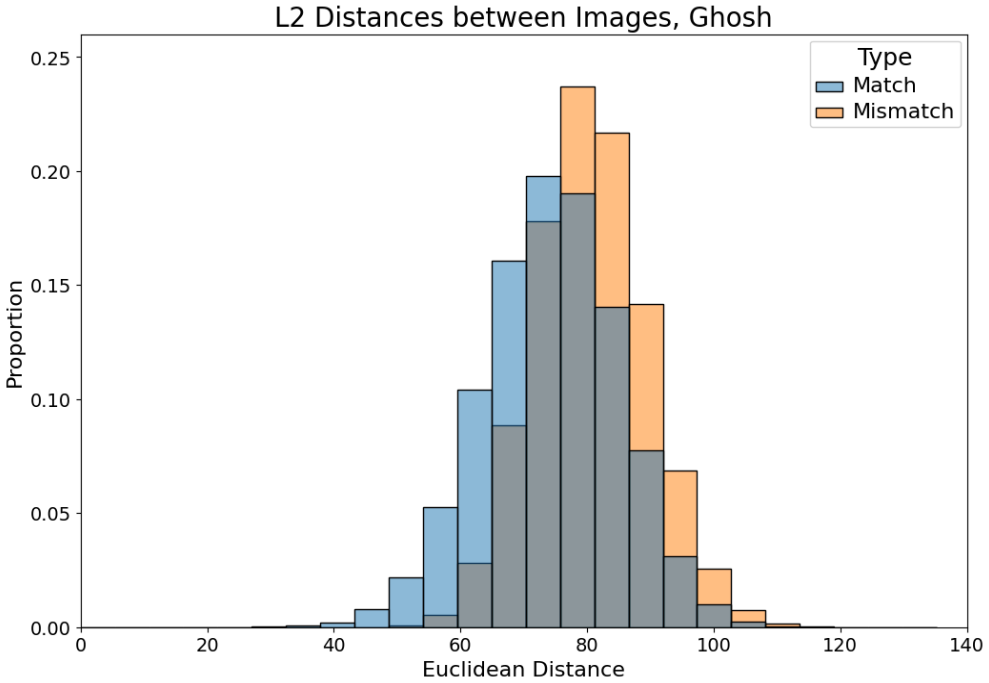

Figure 16a. Euclidean Distances between Images using Ghosh Preprocessing

Euclidean distances show poor performance on the processed images in the LFW dataset. Notice that for each of the distance histogram figures, each distribution is calculated by scaling based on the size of the population. This is because the population sizes between intra-label distances and inter-label distances have an extreme innate imbalance. In the LFW dataset, an individual with two images has one intra-label distance, but has a combined 26,462 inter-label distances, 13,231 per image. As such, scaling is required to make any sense of the overlapping histogram graphs.

The intra-label and inter-label distance distributions nearly overlap for all preprocessing pipelines, meaning that there is little ability to distinguish between images that should be classified as matching versus images that should be classified as mismatching. Although it is difficult to discern, Padded and Stretched images (Figures 16b and 16c in Appendix C) consistently perform worse than Ghosh. This performance deficit is not due to a shakier theoretical foundation, but due to the fact that the pre-trained DeepFace model used the Ghosh pipeline, and thus the weights are calibrated to work best on Ghosh pre-processed images.

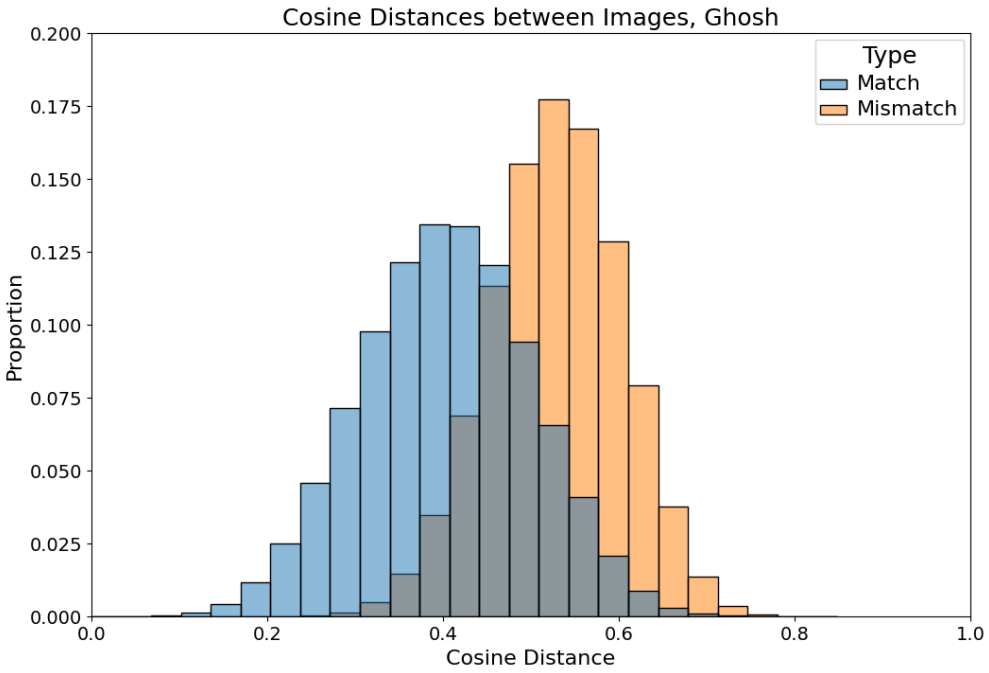

Figure 17a. Cosine Distances between Images using Ghosh Preprocessing

Instead, when investigating the cosine distances between images, a much more desirable pattern emerges. Not only do the histograms show less overlap, but the matched image distances appear to exhibit a positively skewed distribution compared to the normal or slightly negatively skewed distribution of the mismatched image distances, although this separation is much less pronounced on Padded and Stretched images (Figures 17b and 17c in Appendix C). This indicates a much stronger ability to discern between positive and negative matches; however, it still leaves much to be desired.

5.1.3.1. Using a Manually-Inspected Sample Dataset for Diagnosis

This low performance is the result of the fact that the LFW dataset is inherently messy; it contains low-quality images with wildly different environmental and positional setups. Many images do not have the individual facing directly into the camera, occlude portions of the face, and lack standardized exposure levels and white balances.

This is atypical for the intended use-case of this project. For example, when a facial recognizer is used as part of the security measures of a building, the expected reference identification images will all have controlled lighting and posture, and the security system can require the individual to stand in a certain position under certain lighting while facing the camera for the purposes of facial recognition. As such, a compromise needed to be made to attempt to simulate a more clean dataset. 846 images were extracted from the LFW dataset, representing 250 randomly chosen individuals with 3-4 images in LFW. These images were then manually inspected. Images with low quality, extreme non-frontal face positions (looking sideways), or occluded faces (such as sports equipment, hands, or sunglasses blocking the face) were removed from this sample set. In addition, if the images for a particular individual exhibited extreme variance in brightness or white balance, those images were removed. If an individual had only one image left after this process, that image was also removed. In the end, 504 images remained representing 156 different individuals (Gottlieb et al., 2024).

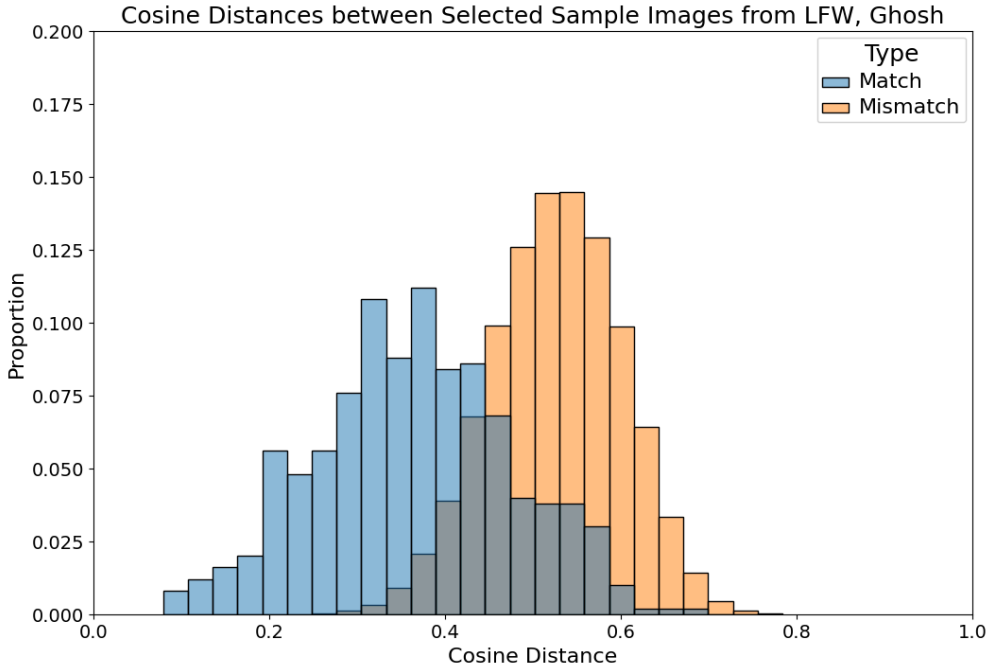

Figure 18. Cosine Distances on Sampled LFW using Ghosh Preprocessing

After sampling, the cosine distances were calculated, resulting in the above histograms. Although there is more overlap than ideal, greater separation was achieved, using this smaller subsample of the LFW dataset. This process introduces intentional bias into the dataset as it is not truly random; however, it more closely simulates a true controlled population which would be expected for the use-case of this project. However, even this sampling is not perfect. Consider the following distance calculations between images.

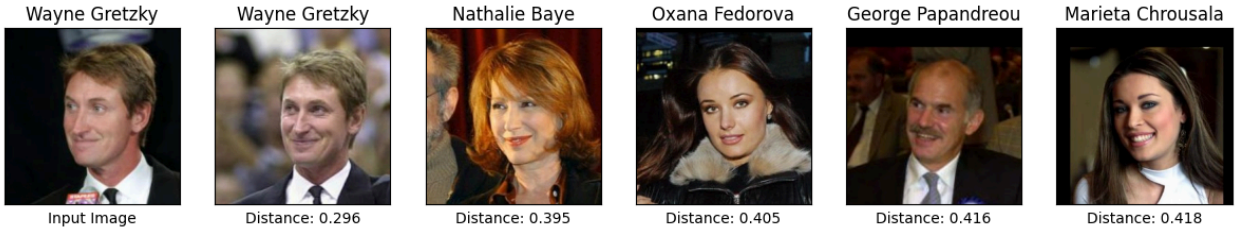

Figure 19. Good matching in sampled LFW, closest match is the reference image

Figure 20. Poor matching in sampled LFW, appears to be heavily influenced by hairstyle

In the first row of images, Wayne Gretzky positively matched against the reference image of himself as the strongest match. The next four closest matches are suitably more distant, although they all appear to share similar poses. In the second row of images Sarah Michelle Gellar fails to match against the reference image of herself. The closest four matches are all of women with shoulder-length hair, matching with the input image, whereas her reference image has short or pulled-back hair, indicating that there is likely some set of features that are being measured which should not be measured such as hair length.

For even greater evidence of better separation on controlled images, and a discussion on potential factors causing this overlap, see Section 5.1.4.7 using a small dataset of personal images. For sections 5.1.4.1 through 5.1.4.6, this sample dataset will be used. An added benefit of using this sample dataset is that the memory requirements of the Pandas DataFrames used for analysis are much lower: on the whole dataset, a simple DataFrame consumed about 6.5 GB of memory, while on the sample dataset, the same simple DataFrame consumed about 9.8 MB of memory.

The next step for evaluating the strength of the facial recognition pipeline is to choose a specific threshold for classifying whether two images match. The threshold that is best to maximize matching is not entirely clear; however, and to determine possible threshold choices requires an investigation into classification metrics.

5.1.4. Threshold Choices

5.1.4.1. The Confusion Matrix

Any binary classification task has four possible results per data point. It can either predict a data point as positive when it belonged to the positive class, predict a data point as positive when it

belonged to the negative class, predict a data point as negative when it belonged to the positive class, or predict a data point as negative when it belonged to the negative class. These four possibilities are called True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN), respectively. These values can easily be summarized using a confusion matrix.

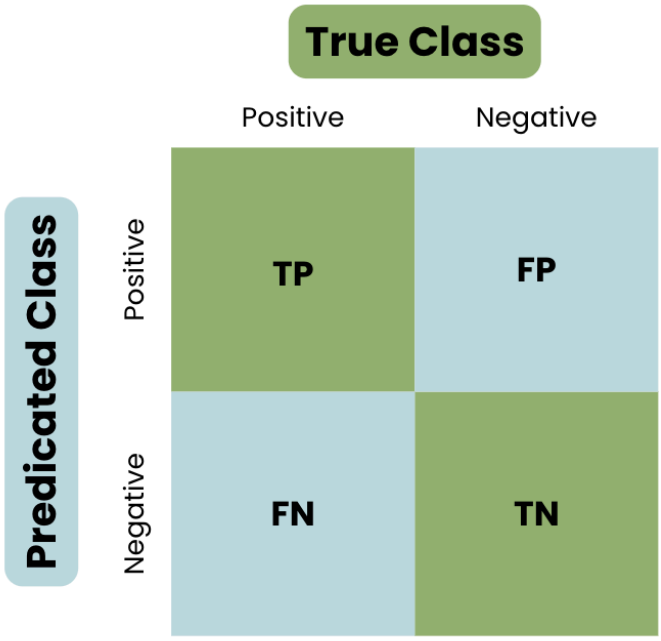

Figure 21. The Confusion Matrix (Ahmed, 2023)

The confusion matrix is simply a way of visually grouping these four possibilities together. However, a wide variety of metrics can be generated using these four numbers. A wide variety of these metrics will be referenced going forward; for the mathematical definitions of these metrics, see Appendix B. In the case of facial recognition, the positive class, otherwise known as class 1, will always be when the input image and reference image are of the same person, while the negative class, or class 0, will always be when the input image and reference image are not of the same person.

5.1.4.2. Accuracy

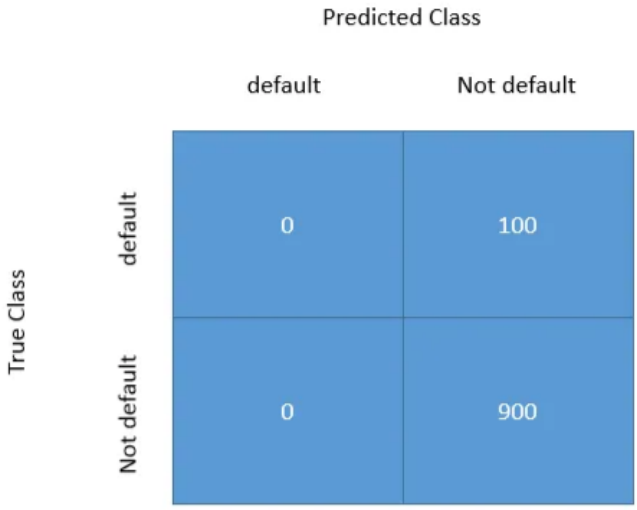

Figure 22. A High Accuracy Confusion Matrix with Poor Predictive Capability (Nagesh, 2022)

The classic layperson's preferred metric is accuracy (Eq. 2.1). However, accuracy is a notoriously poor metric for imbalanced datasets, as it is possible to obtain a very high accuracy while having a very bad classifier. For example, in the above confusion matrix, there are 100 data points belonging to the positive class and 900 belonging to the negative class, yet by always guessing the majority negative class, an accuracy of 90% is able to be achieved. This is a highly accurate, but very bad classifier. Using the sampled LFW data described earlier, it would be trivial to obtain a 99.6% accuracy by always guessing that two images do not match the same person, which is not very helpful. Facial recognition as a classification task will always result in highly imbalanced classes, so accuracy is a poor metric.

5.1.4.3. Recall and Precision

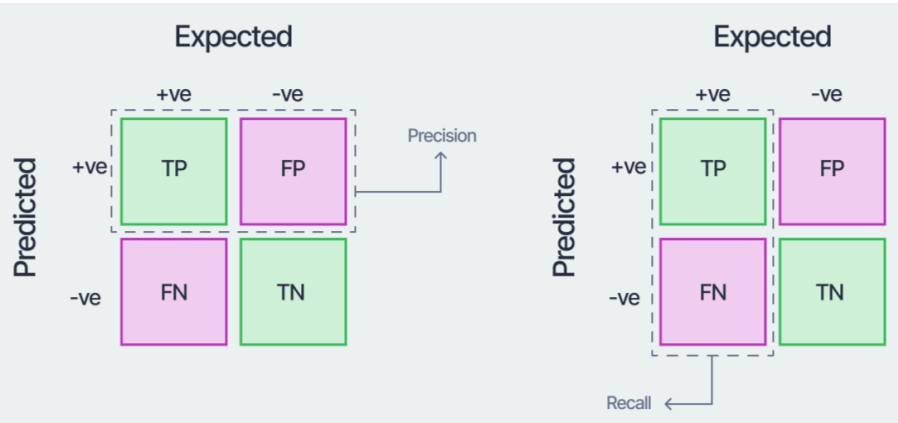

Figure 23. Recall and Precision on Confusion Matrices (Kundu, 2022)

Precision (Eq. 2.4) and recall (Eq. 2.1) are common metrics to evaluate on classifiers. Precision evaluates the positive predictive power of a classifier. In other words, it answers the question, "When the model predicts for the positive class, how effective is that prediction?" Recall evaluates the correctness of the classifier in classifying the positive class. It answers the question, "When the model is given an instance of the positive class, how correct is its classification?" Both precision and recall are necessarily at odds, and the trade-off is explained well in Kundu's blog post (Kundu, 2022). In order to increase precision, recall must be reduced, and vice-versa.

5.1.4.3.1. A Brief Detour to the Precision-Recall Curve

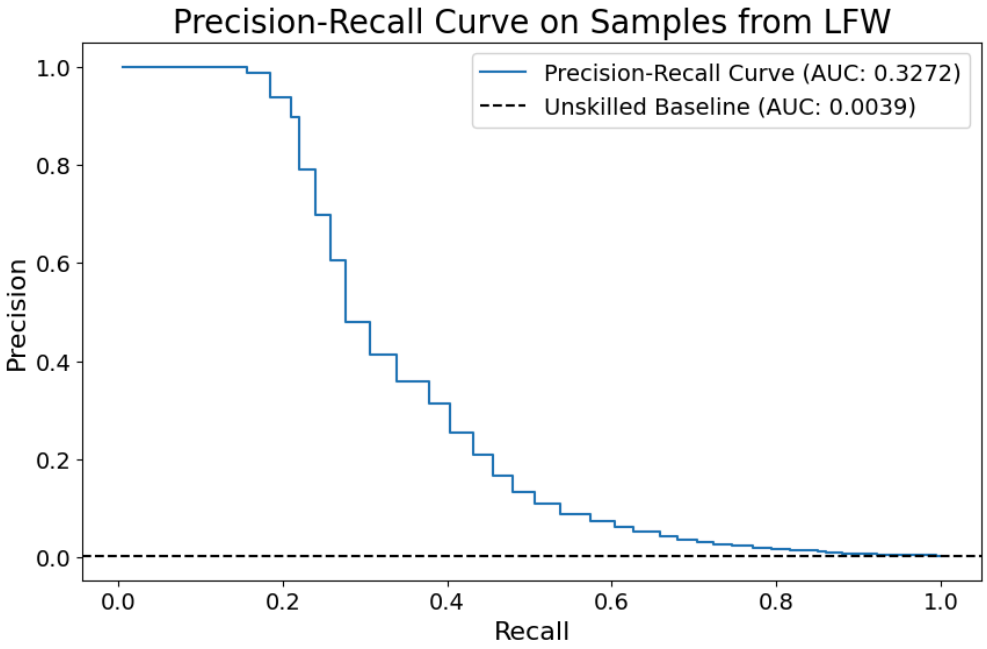

Figure 24. The Precision-Recall Curve on Sampled LFW

To demonstrate the precision-recall tradeoff, the precision-recall curve (PRC) is sometimes helpful to consider. A PRC illustrates how precision and recall shift across thresholds for a given classifier. The better the classifier is at separating the positive class, the closer to the top right (1, 1) the PRC curve is. Every PRC has an unskilled baseline measured by always predicting the positive class, where precision is minimized and recall is maximized (Saito & Rehmsmeier, 2015). The above PRC is calculated from the sampled LFW dataset. The classifier is not particularly strong at separating the positive class but is significantly better than an unskilled baseline.

The goal of the PRC is to help determine which threshold values give the largest change in precision or recall. Deciding which metric is more important for the classification task determines whether to try to maximize precision or recall. Precision is favored in tasks where false positives are undesirable, while recall is favored in tasks where false negatives are undesirable. In a facial recognition task such as building security, a false positive means an unauthorized person is incorrectly allowed in, representing a security risk, while a false negative means an authorized

person is incorrectly left out, which can be mitigated with further interaction with human security personnel. Therefore, precision is often preferred over recall for facial recognition tasks. However, emphasizing precision over everything else has an unfortunate consequence of preventing the classifier from ever making a positive prediction, so it is worthwhile to investigate other metrics.

5.1.4.4. F-Beta Measure (Score) and F1-Score

The F-Beta measure (Eq. 2.6) or score is a way to balance recall and precision. Not to be confused with bias (Eq 3.2) in Appendix B, the Beta value in the F-Beta measure represents the ratio of recall importance to precision importance ("Fbeta_score", n.d.). A lower beta value puts more emphasis on precision, while a higher beta value puts more emphasis on recall (Steen, 2020). As beta goes to zero, only precision is considered, while as beta goes to infinity, only recall is considered. The F_1 score (Eq. 2.7) is a special instance of the F-Beta measure which places equal emphasis on precision and recall and represents the harmonic mean between precision and recall. One of the benefits of using the F_1 score is that extreme measures of either precision or recall are heavily penalized, making the F_1 score ideal for trying to find a middle ground between precision and recall (Kanstrén, 2020).

5.1.4.5. Balanced Accuracy, Informedness, Markedness, and MCC

Recall, precision, and F_1 score are common metrics for classification tasks, but they all share a common weakness. They focus only on the classifier's ability to perform on positive class predictions and ignore the classifier's ability on negative class predictions. On the surface, facial recognition is solely focused on finding positive matches; however, it is also important that a facial recognizer can correctly determine when two images do not match, as a facial recognizer that can ascertain both positive and negative classes is stronger at separating the classes overall without trying to trade one type of mistake for another.

Balanced accuracy (BA) and informedness (BM) are related (Eq. 3.3, 3.4) and are ways to extend the idea of accuracy to highly imbalanced classes. They measure the correctness of both positive and negative classifications as combinations of positive and negative recall (sensitivity and specificity). BA and BM only differ in scale, with BA ranging from 0 to 1 and BM ranging from -1 to 1. A classifier with a high BA/BM exhibits strong classification ability but does not guarantee strong predictive power on both classes; however, BA/BM are the best metrics for determining if a classifier is similar to random guessing (Chicco et al., 2021).

Markedness (MK) is a way to extend the idea of precision to both classes (Eq. 3.5). It measures the positive and negative predictive power of a classifier as a combination of precision and negative precision, and ranges from -1 to 1 like BM. A classifier with high MK demonstrates strong predictive power, but may run into the problem of poor BM or recall, much like precision.

Matthew's Correlation Coefficient (MCC) is defined as the geometric mean between BM and MK, and as such, is the only metric discussed in this paper which considers all of the parts of the confusion matrix (Eq. 3.6, 3.7). MCC ranges from -1 to 1, and a classifier with an MCC close to zero is both uninformed and imprecise on both classes, meaning it tends to predict the

majority class. A classifier with an MCC close to 1, on the other hand, is both informed and precise on both classes, denoting a very strong separator of both classes. This makes MCC very useful for checking the overall health of a classifier, although middling values of MCC (~0.5) do not imply which of MK or BM are high (Chicco et al., 2021).

5.1.4.6. Which Metric to Maximize?

Now that a robust understanding of different classification metrics has been established, it is necessary to figure out which metric is the most useful to maximize for this project.

Threshold	Balanced Accuracy	Precision	Recall	F1 Score	Informedness	Markedness	Matthew's CC
0.253535	0.584996	**0.988372**	0.170000	0.290102	0.169992	**0.985096**	0.409217
0.269697	0.604952	0.897436	0.210000	0.340357	0.209905	0.894317	**0.433268**
0.293939	0.628667	0.605634	0.258000	**0.361851**	0.257335	0.602702	0.393822
0.447475	**0.812796**	0.020455	**0.772000**	0.039853	**0.625591**	0.019398	0.110160

Prediction Statistics at Various Thresholds on LFW Sample

Figure 25. Classification Metrics on Sampled LFW (Maximum Values in Black Boxes)

Four specific metrics were chosen as potential options: BA, precision, F_1 Score, and MCC. Using 100 evenly spaced thresholds from 0.1 to 0.9, the threshold with the highest performance in each category was extracted into the table above, with the caveat that the threshold chosen to maximize precision needed to have a recall of at least 0.15 to be considered useful. Note that all of the above metrics besides BA and BM are influenced by the dataset used and relative class imbalances and are thus not transferable across datasets (Chicco et al., 2021). For evidence of how the underlying dataset dramatically affects the decision thresholds of these statistics, see Section 5.1.4.7. However, it is still possible to use other metrics to choose thresholds for classifying on the LFW sample data.

Maximizing precision results in the most strict threshold of 0.2535; however, recall is very low at 0.17 and BA is also fairly poor at 58.5%. Using this threshold, the facial recognizer would be very confident when it predicts an image match, but it would reject many actual image matches. This strict threshold could be useful for use-cases where security is of the utmost importance at the cost of convenience.

Maximizing MCC produces the most well-rounded classifier. It happens that in this case, the MCC is picking up the high MK of the classification at this threshold. This means that the classifier is quite precise, but poorly informed about which comparisons belong to which class. Some precision is traded for a slight boost to recall, and so maximizing MCC would likely be best in similar situations as when maximizing precision.

Maximizing F_1 score produces a much harsher loss of precision but also gives a further boost to recall. F_1 score was chosen over any other F-beta measure, as it weighs both precision and recall fairly, and other metrics already cover heavier preference towards one or the other of

precision or recall. The notable loss in precision for a small amount of recall may be preferable in some scenarios for a less careful but less precise classifier.

Maximizing BA leads to a low precision classifier with high recall. The classifier is effective at dividing the classes confidently but does so in a way that disregards precision in predicting the positive class, because of the low prevalence (Eq. 3.1) of the positive class. In order to be more correct about the positive class, the classifier becomes overconfident about its positive predictions. Choosing to maximize BA would be useful for cases where true positive matches should be counted as positive, and false positive matches are less important, perhaps for a task such as marking attendance.

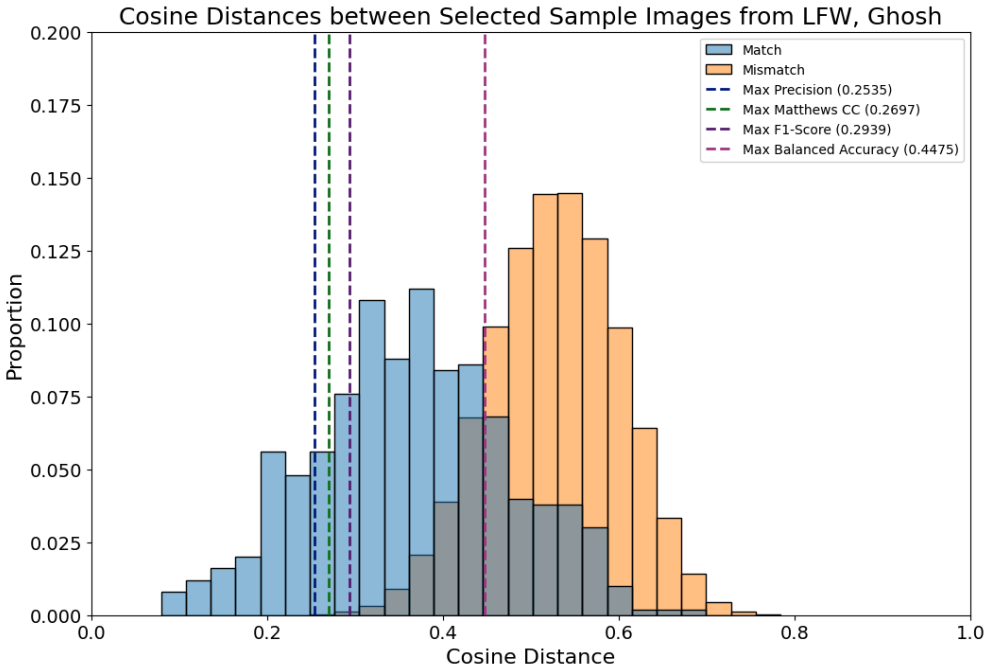

Figure 26. Cosine Distances on Sampled LFW with Thresholds

Taking a look at these decision thresholds on the LFW sample histograms provides a useful illustration of their effects. Remember that the histograms are normalized to have relatively similar sizes, and as such, any areas of overlap of the positive and negative class actually represent a large number of negative class instances over positive class instances.

The maximum precision threshold excludes the negative class almost entirely, while the max MCC and max F_1 score thresholds try to capture more of the positive class while allowing some overlap with the negative class. The max BA threshold splits the overlap of classes roughly down the middle, allowing greater capture of the positive class but about half of the overlapping negative class instances to be captured as well.

```
# Set thresholds based on selected strategy
if threshold_strategy == 'matthews_cc':
    thresholds = [0.269697, 0.277778, 0.285859, 0.293939, 0.302020]
elif threshold_strategy == 'f1_score':
    thresholds = [0.293939, 0.310101, 0.318182, 0.326263, 0.334343]
elif threshold_strategy == 'balanced_acc':
    thresholds = [0.447475, 0.455556, 0.463636, 0.471717, 0.479798]
elif threshold_strategy == 'precision':
    thresholds = [0.253535, 0.261616, 0.269697, 0.277778, 0.285859]
```

Code Snippet 38. Expanded Threshold Strategies Implemented

For this project, MCC was the chosen default metric to maximize, as it was the best metric at trying to evaluate overall classifier strength. Rather than use only the maximizing threshold, the project allowed for a specified threshold_strictness to be set, ranging from 1 to 5 in decreasing levels of strictness. Thresholds were determined by choosing the top 5 thresholds that maximized each strategy, with the caveat that threshold 2 had to be greater than threshold 1, and so on. This allowed the project to have a better chance of finding matches, given the uncontrolled nature of the LFW dataset.

5.1.4.7. Using a Controlled Dataset of Personal Images

One of the underlying issues with a simple preprocessing pipeline of detect, align, crop, and resize is that these basic operations are unable to compensate for bad data inputs. There are many factors which can lead to poor facial recognition performance, including inconsistent white balance, illumination, and poor frontalization of faces. Using a more consistent dataset can produce dramatically different results, even when using the same processing steps. To explore this concept, a small dataset of personal images was constructed, consisting of 95 images across 25 different individuals (Gottlieb et al., 2024). These images are of much higher resolution than the LFW images and exhibit far greater, although not perfect, control of external variables. Some sets of images representing the same person during different time periods were split into separate labels (i.e., Dan1, Dan2 both represent Dan during different years). These images were processed using the Ghosh pipeline. Evaluations of the facial recognizer on this personal dataset are below.

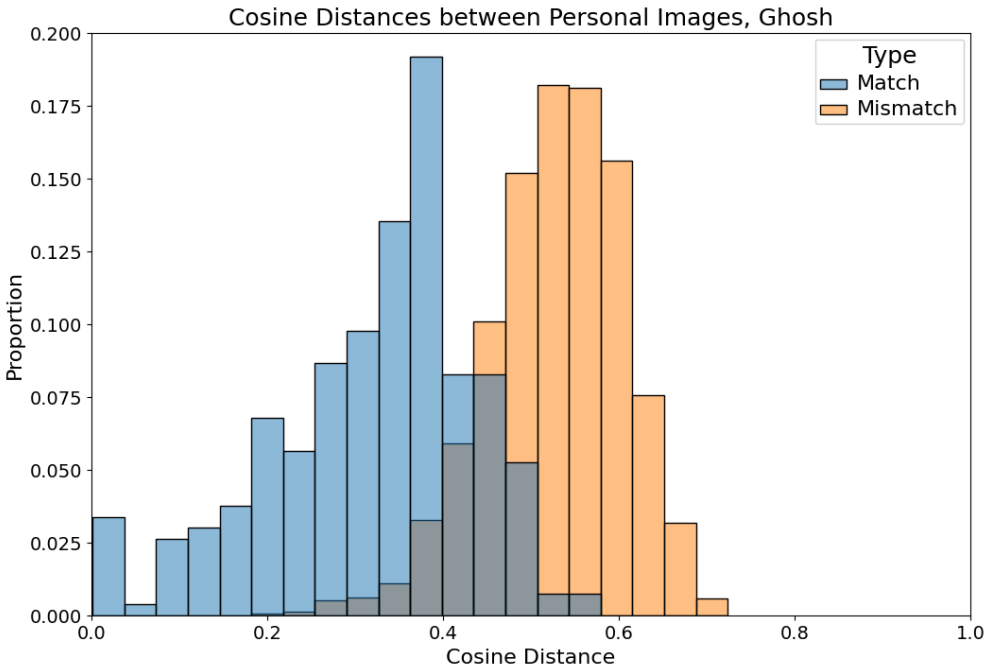

Figure 27. Cosine Distances on Personal Dataset

The histogram comparison between this personal dataset and the sample LFW dataset is revealing. With finer control over external variables, there is a stronger separation between the positive and negative classes. Positive matches exhibit much lower overall distance.

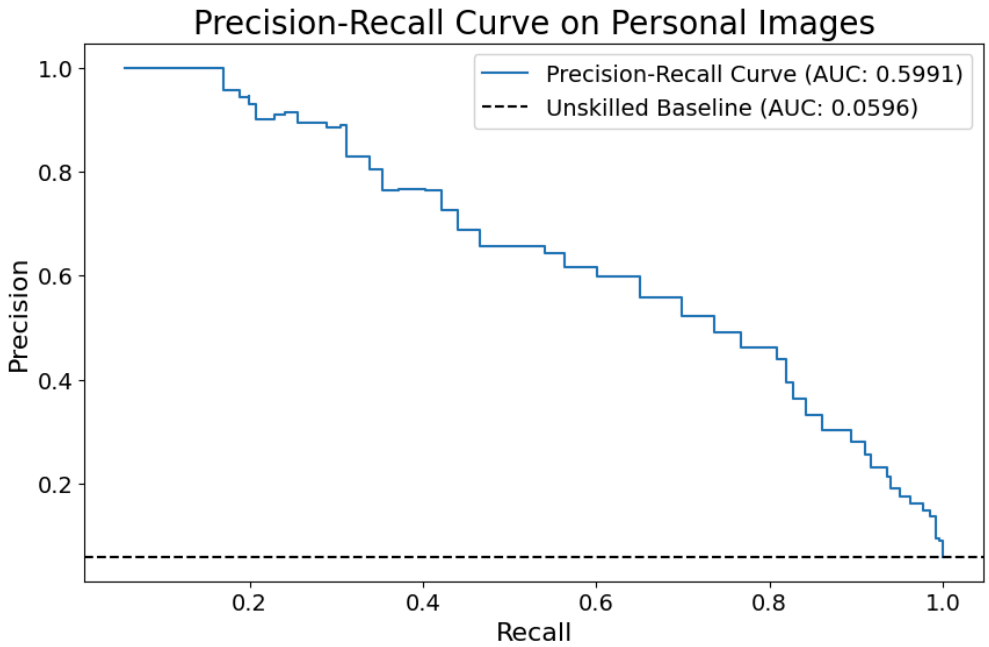

Figure 28. The Precision-Recall Curve on Personal Dataset

The PRC on the personal dataset showcases that the facial recognizer is much stronger at identifying positive class images on the personal dataset. It is possible to maintain ~50-60% precision, even at ~50-60% recall on the personal dataset, whereas on the sample LFW dataset, precision dropped to ~15% at the same level of recall.

Threshold	Balanced Accuracy	Precision	Recall	F1 Score	Informedness	Markedness	Matthew's CC
0.213131	0.599267	**0.946429**	0.199248	0.329193	0.198534	**0.898118**	0.422264
0.374747	0.808875	0.558065	0.650376	**0.600694**	0.617749	0.535682	0.575254
0.382828	0.829381	0.522472	0.699248	0.598071	0.658762	0.503002	**0.575638**
0.447475	**0.874970**	0.281324	**0.894737**	0.428058	**0.749940**	0.273587	0.452961

Prediction Statistics at Various Thresholds on Personal Images

Figure 29. Classification Metrics on Personal Dataset (Maximum Values in Black Boxes)

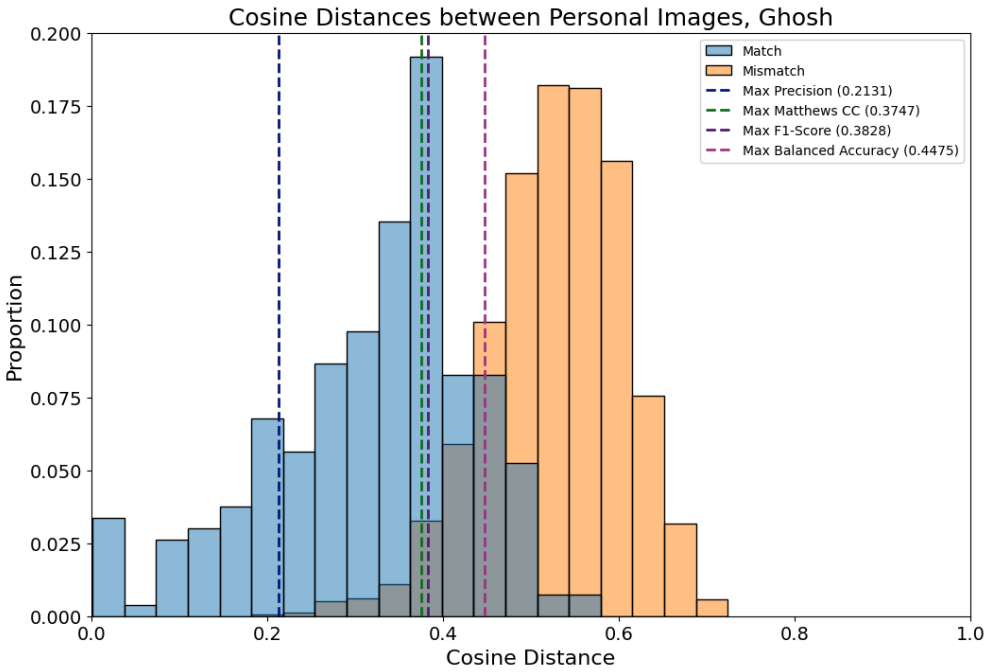

Figure 30. Cosine Distances on Personal Dataset with Thresholds

Looking at the extended set of classification metrics, it is clear that the same facial recognizer performs much better on the personal data set. The classifier is overall much better informed about the distributions of the positive and negative classes, causing a great increase in recall, BA, and BM, and both F_1 score and MCC corroborate this observation.

Statistics and metrics already provide strong evidence that a controlled dataset greatly influences results, but it is still useful to directly evaluate some images from the dataset. Consider the following batches of images.

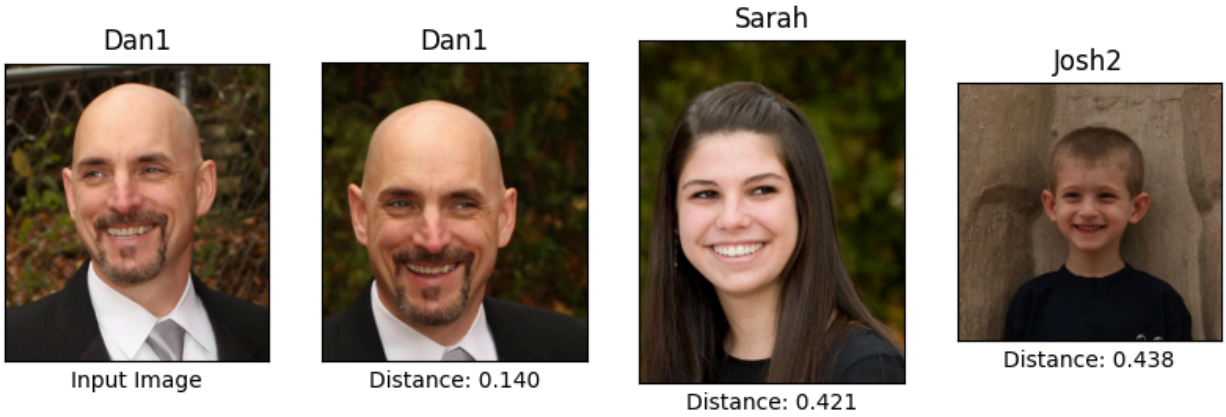

Figure 31. Good matches; the input image is close to the reference and far from the next closest

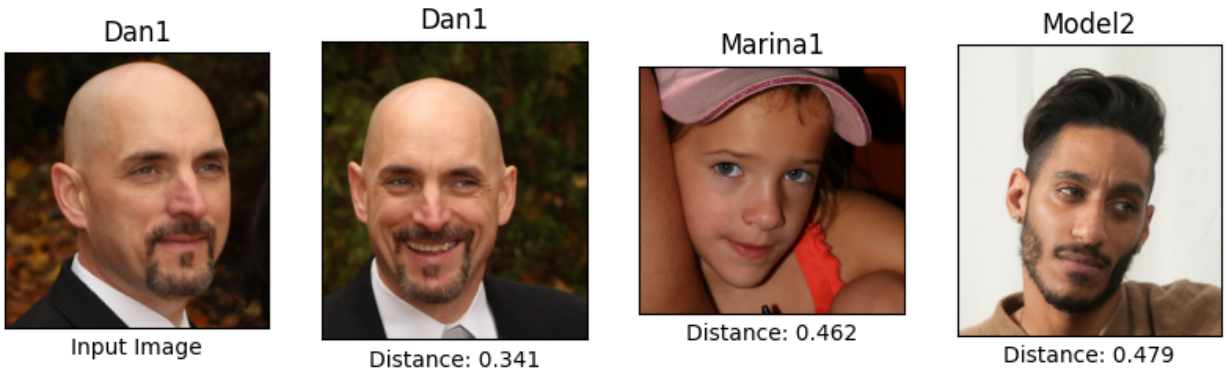

Figure 32. Mediocre matches; the input image is closest to the reference but somewhat distant.

Figure 33. Poor matches; Dan2 and Dan1 are labeled separately but represent the same person

In the first row of images, the facial recognizer is performing well. The input image is closest to the reference image, and the next two closest images are quite distant, in comparison. In the second row of images, the facial recognizer is still working correctly, as the reference image is the closest to the input image; however, the distance is large enough that the facial recognizer is likely to miss the match depending on the selected threshold.

In the third row of images, the recognizer is technically working correctly, as Dan2 and Dan1 are separated in the data set as two different labels. Dan2 did match on its own reference image with a distance of 0.003, not shown in the third row. However, Dan1 and Dan2 do represent the same person in reality, and the distance between the Dan2 input image and the Dan1 reference image is extreme. This is likely due to the large difference in face frontalization between the two images; Dan2 is facing away from the camera while Dan1 is facing forwards. Interestingly, the closest non-Dan matches in the second and third rows are Luke and Marina, who happen to be Dan's son and daughter, respectively.

Even though the personal dataset is not perfectly controlled and thus has some errors, this evaluation shows that having increased control of external variables when generating the reference dataset can cause significant performance increases even with sub-optimal preprocessing.

In conclusion, careful curation of data is essential for evaluating the effectiveness of a facial recognition system. Messy images taken under a wide range of conditions tend to cause a facial recognition system to perform poorly, while images taken under controlled conditions are handled correctly, even without specifically training on those images. For most use-cases such as building security, the image capture conditions can be strongly controlled, which grants a noticeable increase in facial recognition effectiveness.

5.2. UI Testing

Testing the UI involves verifying that all components function correctly:

- Camera View: Ensure the live feed is displayed.
- Capture Button: Verify that clicking the button captures an image and successfully runs the find_image_match() function from the Facial Recognition Layer, using vectors retrieved from the Database Access Layer. Check that if a match is found, the reference image is displayed, and if no match is found, a match details string is displayed to the user.

5.3. Database Testing

Testing the database involves ensuring data is correctly stored and retrieved:

- Seeding Vectors: Verify that vectors are correctly stored in Firestore.
- Retrieving Vectors: Ensure vectors are accurately retrieved and passed into the find_image_match() function during facial recognition.
- Retrieving Reference Images: Check that reference images are properly returned to the UI layer if a match is found during the find_image_match() function.

6. User Guide and Setup

6.1. Python and Jupyter Notebooks Versions

The Jupyter Notebook demo and all notebooks in the Github repo (Gottlieb et al., 2024) were successfully run under Python 3.8.10. The Kivy UI application was tested under Python 3.9.

The Jupyter Notebook Demo was successfully tested using the following versions:

```
IPython          : 8.7.0
ipykernel        : 6.19.4
ipywidgets       : 8.0.4
jupyter_client   : 7.4.8
jupyter_core     : 5.1.0
jupyter_server   : 2.0.4
jupyterlab       : not installed
nbclient         : 0.7.2
nbconvert        : 7.2.7
nbformat         : 5.7.1
notebook         : 6.5.2
qtconsole        : 5.4.0
traitlets        : 5.8.0
```

6.2. Sample requirements.txt file

Use pip install to install the following packages.

# Libraries needed for Jupyter Notebooks and Facial Recognition Engine
dlib==19.24.4
mediapipe==0.10.11
numpy==1.23.0
opencv_python==4.8.0.76
pandas==1.5.2
retina_face==0.0.17
tensorflow==2.13.1

# Libraries needed for Kivy-based UI and Firebase DB
firebase_admin==6.5.0
Kivy==2.3.0
opencv_contrib_python==4.10.0.84

6.3. Directory Structure for Jupyter Notebook Demo

Note that dummy_dataset and personal_images_batch2 datasets are pre-built sets for demonstration purposes. Any folder name can be placed into ./data for convenience of access. For preprocessing of custom data, consider using the Preprocessing notebook in ./src (not shown below) or by using the functions in ./src/modules/dataset_ops.py, image_preprocessor.py and image_vectorizer.py.

Pre-trained models must be obtained from the Github sources specified in Section 6.4, unpacked and placed in ./pretrained_models.

```
.
|── Demo-and-Test.ipynb
|── data/
|   ├── dummy_dataset/
|   |      ├──input_images/
|   |      ├── reference_images/
|   |      └── reference_vectors/
|   └── personal_images_batch2/
|          ├──input_images/
|          ├── reference_images/
|          └── reference_vectors/
|── pretrained_models/
|   ├── shape_predictor_5_face_landmarks.dat
|   └── VGGFace2_DeepFace_weights_val-0.9034.h5
|── src/
|   └── modules/
|          ├── data_manipulation.py
|          ├── dataset_ops.py
|          ├── face_api.py
|          ├── face_utils.py
|          ├── image_preprocessor.py
|          ├── image_vectorizer.py
|          ├── __init__.py
|          ├── stat_utils.py
|          └── utils.py
```

6.4. External Pre-trained Models Needed

The shape_predictor_5_face_landmarks.dat.bz2 file from davisking's Github (davisking, 2024) is used by the dlib library for detecting faces and must be unpacked and placed into ./pretrained_models. The VGGFace2_DeepFace_weights_val-0.9034.h5.zip from Ghosh's Github (Ghosh, 2019) is the file containing the model weights used by the DeepFace neural net and must be unzipped and placed into ./pretrained_models.

Example unpacking Linux commands using a terminal (from inside the ./pretrained_models directory):

- bzip2 -d shape_predictor_5_face_landmarks.dat.bz2
- unzip VGGFace2_DeepFace_weights_val-0.9034.h5.zip

6.5. How to Use Demo and Test Jupyter Notebook

The Demo and Test notebook is set to be run with very few required inputs from the user.

```
# Specify threshold strategy: default matthews_cc, least strict is balanced_acc, see face_api.py for more details
# Specify threshold strictness: 1 to 5 from most to least strict
# Specify maximum number of matches to return if distances are less than threshold, positive integer
threshold_strategy = 'balanced_acc'
threshold_strictness = 5
match_num = 3

# threshold_strategy = 'matthews_cc'
# threshold_strictness = 2
# match_num = 1
```

Code Snippet 34. Facial Recognition Parameter Settings

Cell 3 is where the user specifies the threshold strategy, threshold strictness, and maximum number of matches they wish for the facial recognizer to return. For definitions of threshold strictness and threshold strategy, see Section 5.4.6 or the code snippets for the facial recognition layer in Section 4.6.3. The balanced_acc strategy with strictness 5 is the most lenient strategy and is the best set of parameters for manually inspecting the effectiveness of the facial recognition engine by allowing a wider variety of matches.

```
# Specify reference directories for images and vectors to compare against.
# Specify number of digits as suffix for images
# reference_dir = './data/dummy_dataset/reference_images'
# vector_dir = './data/dummy_dataset/reference_vectors'
# num_digits = 4

reference_dir = './data/personal_images_batch2/reference_images'
vector_dir = './data/personal_images_batch2/reference_vectors'
num_digits = 2

vector_dict = face_api.load_vector_dict(vector_dir, num_digits = num_digits)
```

```
# Specify input image directory and name to check against.
# image_dir = './data/dummy_dataset/input_images'
# image_name = 'Wayne_Gretzky_0002.jpg'
# image_name = 'Sarah_Michelle_Gellar_0002.jpg'

image_dir = './data/personal_images_batch2/input_images'

# image_name = 'Jack-04.jpg'
# image_name = 'Adam-02.jpg'
# image_name = 'Ed-02.jpg'
image_name = 'Dan1-02.jpg'

image_path = os.path.join(image_dir, image_name)
```

Code Snippet 35. Reference and Vector Directories, Numbering Schema, and Input Image Path

Cells 4 and 5 are where the user specifies the directories containing the reference images, reference vectors, and input images they wish to use. The number of digits should be specified and is an indicator of how many digits post-pend the image name. For example, "Wayne_Gretsky_0002.jpg" has 4 digits (0002) while "Dan1-02.jpg" has 2 digits (02). All vectors in the reference vector directory must follow the same digit schema.

```
# Load in base image for inspection
base_image = cv2.imread(image_path)[:, :, ::-1]
```

```
# Show base image
fig, ax = plt.subplots(figsize = (4, 3))

ax.imshow(base_image);

ax.set_title(f'{image_name.split(".")[0]}')
ax.set_xlabel('Input Image');
ax.set_xticks([]);
ax.set_yticks([]);
ax.grid(False);

plt.tight_layout();
```

Code Snippet 36. Load in base image and display for verification.

Cells 6 and 7 display the base image for inspection of the image prior to matching. An example is shown below:

Figure 34. Base image display for Demonstration Jupyter Notebook

```
# Perform Matching
match_details = face_api.find_image_match(image_path, vector_dict, model = model,
                                          threshold_strategy = threshold_strategy,
                                          threshold_strictness = threshold_strictness,
                                          match_num = match_num)
```

```
# If no matches are found, print match details, else show matches found
if type(match_details) == str:
    print(match_details)
else:
    # Display results
    # Split results into fields
    matches, threshold, metric = match_details

    # Number of subplots
    n = len(matches) + 1

    # Load matched images
    matched_images = [cv2.imread(os.path.join(reference_dir, x[0] + '.jpg'))[:, :, ::-1] for x in matches]

    # Create subplots for base image and all matched images
    fig, ax = plt.subplots(1, n, figsize = (round(n * 2.5), 4))

    # Display base image
    ax[0].imshow(base_image);
    ax[0].set_title(f'{image_name.split(".")[0]}');
    ax[0].set_xlabel('Input Image');

    # Turn off axis numbers
    for a in ax:
        a.set_xticks([]);
        a.set_yticks([]);
        a.grid(False);

    # Display matched images
    for i in range(len(matched_images)):
        ax[i + 1].imshow(matched_images[i]);
        ax[i + 1].set_title(f'{matches[i][0]}');
        ax[i + 1].set_xlabel(f'Distance: {matches[i][1]:.03f}');

    plt.tight_layout();
```

Code Snippet 37. Run find_image_match() and Display Matching Results

Cell 8 is where find_image_match() is called, and cell 9 automatically visualizes the results. Note that cell 9 shows the input image and all matched images in one row, and for large match_num parameters, this may result in small images and poor visualization. A maximum match_num of 5 is recommended for this reason. An example of the display for 5 matches is shown below. If the user wishes to test more images or directories, they can simply change the parameters in cells 3, 4 and 5, and rerun the rest of the notebook to retrieve new results.

Figure 35. Match results display for Demonstration Jupyter Notebook

7. Conclusion and Lessons Learned

7.1. Project Summarization

The demand for facial recognition software continues to grow (Shashkina, 2024), and thus there is great value in the development of a user-friendly and simple to implement facial recognition architecture. By integrating off-the shelf libraries, tools, and components, the project demonstrates the viability of the architecture as an effective approach to achieving this goal.

As primarily a proof-of-concept prototype, the application demonstrates that a three-layered design with simple and clearly defined interfaces can form a robust and flexible foundation for the development of a secure, modular, and potentially scalable facial recognition framework.

A more in-depth exploration of potential future enhancements to the software are described in Appendix A.

7.2. Lessons Learned

7.2.1. Curate the Input and Reference Images

This paper has demonstrated that reasonable levels of effectiveness can be obtained using a pre-trained facial recognition model. In order to keep implementation simple and lightweight, minimal preprocessing is required, which means that one of the key elements for improving performance of a facial recognition engine is to carefully curate the input and reference data. Although this project began with the goal of training a model from the ground up, by controlling the posing and illumination of the individuals within each image, impressive results can be obtained without resorting to the costly and time-consuming process of training a custom neural network.

7.2.2. Establish a Uniform Development Environment

One of the challenges encountered during the development process was environment setup conflicts when moving from one developer’s machine to another’s. This included package and version conflicts, different operating systems, and different directory structures between the various machines. A uniform development environment that was shared by all developers could have mitigated these problems. This could be achieved either by using identically configured virtual machines or a common Docker image for use by all developers.

8. Appendix A: Future Enhancements

As a prototype, the application demonstrates the viability of the three-layer architecture for implementing a facial recognition application. A number of future enhancements should be considered, addressing the following areas:

- functionality
- scalability
- speed
- memory usage
- error handling

8.1. Functionality Enhancements

As the application currently stands, it works with a fixed data set. This is acceptable for the purposes of a prototype. However, the application lacks certain functions that should be included in future versions.

8.1.1. Different Levels of Authority

In a real-world usage of the application, at least 3-5 different types of users are envisioned, each with varying degrees of access to the database:

- Scenario 1: This person would be authorized to use the system to take a picture and search for a match in the database. This might be a security guard who uses the system to authorize access to a building, for example.
- Scenario 2: This person would be able to browse the database in view-only mode, but would not be allowed to make any changes. This is useful for searching to see if someone is in the database, in cases where perhaps the camera is unavailable or a photo match is not found.
- Scenario 3: This person would be an administrator or HR person who onboards new employees. They would only be allowed to add new entries to the database, e.g., when a new person is hired.
    - Scenario 3a: This would be an onboarding photographer, who is only allowed to add photos to an existing database entry that does not yet have a photo associated with it. For example, in the case of a new hire, the scenario 3 HR person might enter all the basic information for the new hire and then send them to another room where another user with 3a access would take the new hire's photo and add it to the database record for the new hire.
- Scenario 4: This would be an administrator or HR person who has full access to the database. This person would be able to browse the database, make changes to entries, delete people from the database, and add new people to the database.

8.1.2. Browsing, Adding, Editing, and Deleting Database Entries

Authorized users should be able to browse the contents of the database, select individual records for viewing, and be able to edit all of the details for each person whose image and personal data is stored in the database. Authorized users should also be able to delete records from the database. Provision should also be made for replacing the photo associated with an individual record in the database. An audit trail log should be maintained, documenting all changes made and who made them.

8.1.3. Selecting Cameras

Most systems that use a camera might default to the system-defined default camera, but also allow the user to select a specific camera. This enhancement would allow the user to select a camera from the cameras attached to their local machine.

8.1.4. Testing the Camera

In its current configuration, when the user clicks “Take Photo,” a photo is taken and a search for a match is initiated. This does not provide any way to simply take a picture and preview it, without also having to undertake the search for a match in the database. This enhancement would decouple taking the picture from searching, so the user could click a button to take the picture and then click another button to initiate the search. This would allow the user to ensure that the photography subject is properly aligned facing forward, that their eyes are not closed, etc. and confirm the photo is suitable before matching.

8.1.5. Integration with an Existing HR or Personnel System

Given that most companies already have a system for tracking their employees and storing the employees’ personal data, the system could be modified to retrieve people’s personal details from the existing system, rather than storing potentially duplicate data in the facial recognition system’s own database.

8.2. Scalability Enhancements

For the purposes of this application, scalability will be considered for the following scenarios:

- growth in the number of entries in the database
- growth in the number of end user nodes (e.g., using the system at multiple entry points in multiple buildings)

8.2.1. Managing a Growing Number of Entries

In the current prototype, the database is relatively small, holding 175-200 entries. The facial recognition engine currently loads all the image vectors into memory and stores them in a dictionary keyed on personID that is used for the image matching. Each image vector is 4 KB in

size. As the number of database entries grows, this approach of holding all the vectors in memory at once may no longer be viable.

Two approaches have been identified as possible paths forward to handles this:

- partitioning and loading the data in chunks, each chunk containing some reasonable number of entries that can fit into memory
- moving the facial recognition to the server, where it would run as a service

8.2.1.1. Partitioning the Data into Chunks

Suppose that the maximum number of entries that can be held in memory at once is N. In this scenario, the facial recognition engine would load the first batch of N entries, perform the search against them, then load the next batch of N entries, perform the search against them, and so on, until all the entries have been searched.

If the facial recognition engine component is kept on the local machine, all the vectors could be cached locally in pickle files, with each pickle file representing a batch of N entries. These batches could be loaded and searched one after another until the search is complete. The benefit of caching them locally is that the facial recognition engine would not need to make calls to the server database each time it needs to load a new batch. This is viable because the vectors cached locally are relatively static; they do not need to be changed except when an existing database entry has a new photograph loaded. Provision would need to be made to update the cache when changes are made to the server database, including changes to existing photos, deletions of existing entries, and addition of new entries.

8.2.1.2. Deploying the Facial Recognition Engine as a Service

It makes sense to move the facial recognition engine off of the local machine and onto the server, where it would run as a service. This puts the facial recognition engine on the same physical machine as the database, and greatly speeds queries from the facial recognition engine to the database access layer.

In this scenario, the facial recognition engine might not need to ever store more than one database entry's worth of vector data in memory at a time. If the access speed is sufficient, the facial recognition engine can sequentially read the vectors from the database for each image search.

If database access in this scenario proves too slow, then the facial recognition engine can load and store batches of image vectors in chunks, as described above under Section 8.2.1.1. As a process running on the server, the facial recognition engine could be allocated much more working memory than is available on the local machine, and it could therefore hold much larger chunks of data in memory.

8.2.2. Managing a Growing Number of User Nodes

The current prototype is designed to run on a single local machine, where the UI and facial recognition engine both reside on the local machine, and the database resides on Firebase in the cloud. As such, it implements what is known as a thick client architecture, where most of the processing is performed locally rather than on a remote server (Gillis, 2020).

In a real-world scenario, it is likely that the application would need to be deployed on multiple machines in multiple locations. For example, if the application is being used to grant access to a company's offices, the application might need to be deployed at multiple entrances to a single building and possibly in multiple buildings as well.

The current design and implementation supports this usage. However, it suffers from the some of the same challenges as other thick client applications, for example:

- Application updates need to be deployed to multiple machines.
- As a thick client application, a certain minimum hardware specification is required, precluding the possibility of deploying on cheaper commodity hardware like chromebooks or raspberry pi devices.

These challenges could be addressed by moving the entire application to a server- or cloud-based solution. The UI would have to be reimplemented as a web application or use some other solution, but the facial recognition engine and database access layer would be able to be repurposed with few changes.

8.3. Availability Enhancements

The current prototype uses Firebase, a cloud-hosted object/document database. If the application ever needed to run in disconnected mode, i.e., with no internet or intranet connectivity, this could be a problem.

If standalone, unconnected operation were to become a requirement, then possible solutions include:

- Hosting the database locally
- Caching the contents of the database locally, as described in Section 8.2.1.1. In this scenario, an accompanying csv file or local database replicated from the server could host the remaining person data (name, phone number, etc.)

8.4. Speed Enhancements

The current prototype performs a sequential search of the entire database of image vectors for every image match request. This could become a bottleneck as the database grows in size.

Anything that can be done to eliminate the need for an O(N) sequential search would speed up the performance.

One possible method would be to use unsupervised or semi-supervised clustering algorithms to group the vectors into batches of similar types. There are numerous types of clustering algorithms, including BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) and K-Means Clustering (Brownlee, 2020). Determining which clustering algorithm best suits the application's needs would require further investigation. The purpose of clustering would be to partition the database in an informative way to speed up searches. Instead of having to search the entire database, the facial recognizer can instead perform an initial comparison to a small sample of representative vectors, one from each cluster, choose the cluster whose representative vector is most similar to the input vector, and then perform a linear search on the vectors within that cluster. This brings down the search algorithm from O(N), where N is the number of vectors in the database, to O(M) where M is the number of vectors in the chosen cluster, which is, by definition, less than or equal to N.

8.5. Error Handling Enhancements

The current prototype has almost no error handling. This is fine for a proof-of-concept prototype, but it is unacceptable for production deployment. Below is a list of possible errors that could occur. In future versions of the software, these errors should be tested for and appropriately handled. Test cases should be created simulating these error conditions and incorporated into a system testing plan.

8.5.1. Environment/Setup Errors

- The database is unavailable.
- No camera is installed.
- The installed camera is not accessible to the application.
- The local disk cannot be written to.

8.5.2. Internal Errors

- The image file passed from the UI to the facial recognition engine either:
    - cannot be opened or does not exist
    - cannot be read
    - is empty
    - does not contain a valid image format
- facial recognition errors
    - The model cannot be loaded from disk.
    - Image vectorization fails to locate a face in the image.
    - get_image_vector returns an empty vector.
    - There is insufficient memory to store the vectors in memory.
- Database access layer errors
    - get_all_vectors_and_ids returns

        - an empty vector
        - an incomplete subset of all vectors in the database
    - get_person_data
        - receives an invalid personID as input
        - returns no data, despite having received a valid personID as input

8.6. Facial Recognizer Enhancements

One of the original goals of this project was to train a model from the ground up. This turned out to be infeasible in the time allotted for the class, so the project utilized the pre-trained weights developed by Swarup Ghosh for the DeepFace neural network (Ghosh, 2019). However, there were some issues in Ghosh's preprocessing of the VGG-Face2 dataset which could be addressed and could be used to retrain the DeepFace network with better inputs. Some potential enhancements are discussed below.

8.6.1. Using a Stricter Preprocessing Pipeline to Reduce Bad Inputs

As demonstrated in Section 5.1, when using a pre-trained model, it is best to mimic the exact preprocessing steps used when training the model. However, the preprocessing steps used by Ghosh were overly simplistic and introduced the potential for large noise influences due to generous cropping from the dlib package.

Figure 20. Poor matching in sampled LFW, appears to be heavily influenced by hairstyle

This example was brought up in Section 5.1.3.1, and it is clear that a facial recognizer should be focused on extracting actual facial features which are relatively immutable, rather than features such as hairstyle which are subject to change over time. To figure out a way to remedy this issue, it is necessary to return to the preprocessing pipeline.

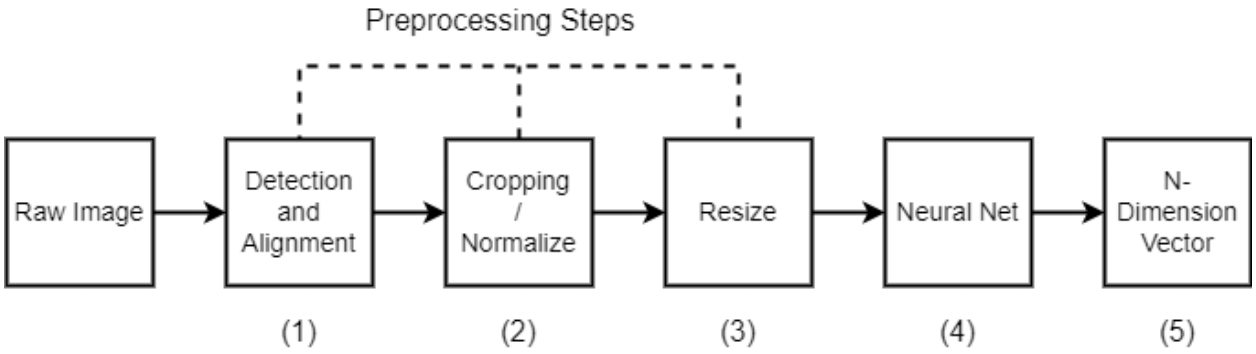

Figure 9. Preprocessing pipeline visualization

One method of enhancing the preprocessing pipeline is to evaluate different options for the Detection and Alignment, Cropping, and Resizing steps.

8.6.1.1. Facial Preprocessing Backends

In addition to dlib, there are two other preprocessing packages that were considered for this project before training the model from scratch was abandoned due to time constraints.

RetinaFace (Deng et al., 2019) is the face detection module of the insightface project (Insightface, 2024). The facial recognition engine that could be used is the tensorflow-based reimplementation that was made pip-compatible by Sefik Serengil (Serengil, 2024). RetinaFace is slower to execute but is effective at finding faces in an image. It does not produce as straight of an alignment as dlib, but it tends to do a better job at alignment than MediaPipe. For bounding boxes, RetinaFace tends to capture foreheads in its area, which leads to a larger facial area. This is sometimes problematic when a person is wearing clothing that encroaches on the forehead, such as a hat.

MediaPipe (Google AI, n.d.) is a machine-learning tool created by Google. The source code is not readily available, and instead of creating a bounding box, it instead generates a set of 478 landmarks on each face it detects. From their tutorials, MediaPipe's primary use-case is to create 3D face-mesh models for use in real-time virtual editing, but the facial recognition engine could use it simply as part of the process of aligning and normalizing images. MediaPipe tends to perform poorly for alignment, but creates a very accurate face mesh, making it the preferred choice for normalization.

A facial recognition engine was designed that used dlib, RetinaFace, and MediaPipe as potential backends for alignments and normalization.

8.6.1.2. Detection and Alignment

Facial detection is the process of finding a bounding box for a face, while facial alignment is the process of aligning a face so that ideally, the eyes are roughly level and the nose is straight. This is not always a perfect process, as faces can be significantly rotated in three dimensions, whereas the image representation can only be rotated in two dimensions.

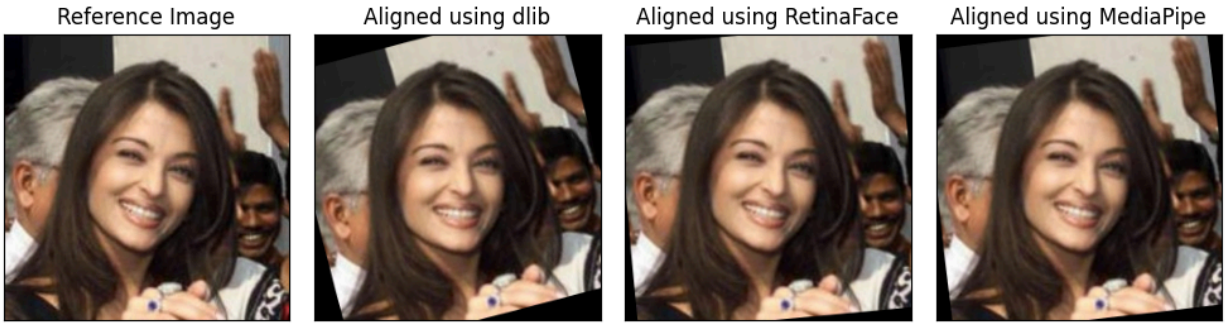

Figure 36. Image alignment backend comparisons

Above is an example of the differences between all three libraries in the task of facial alignment. dlib's alignment process tends to produce the best results and has the side effect of centering the image around the detected facial region, whereas RetinaFace and MediaPipe often produce similar results and rotate around the center of the image. The preferred backend for facial alignment is still dlib, as used by Ghosh.

8.6.1.3. Normalization / Cropping

Facial normalization is the process of extracting only the important areas of an image representing a face, often called the region of interest (ROI). The purpose of this step is to remove extraneous background information from the image, so that the input image to the neural net conveys maximal meaningful information about the detected face. In practice, facial normalization consists of cropping the image around the detected facial area.

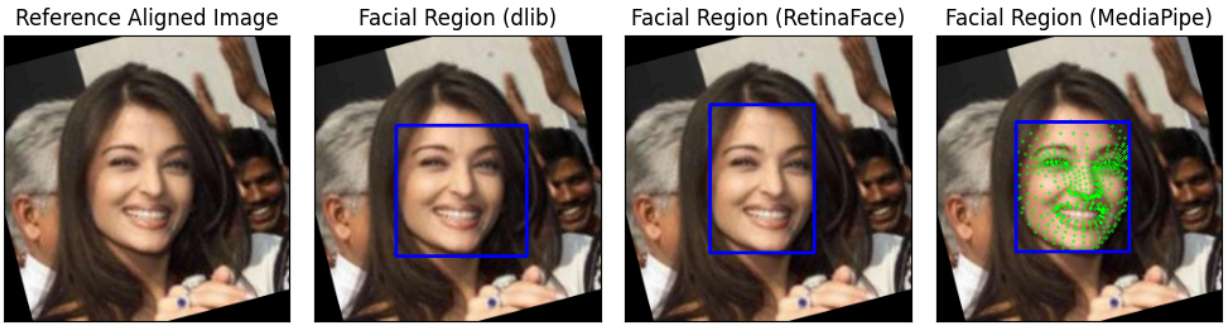

Figure 37. Face detection for image normalization backend comparisons

The image above shows the difference in detected facial regions for each library. As can be seen, dlib detects a very generous facial region, which captures a lot of unnecessary information, which is a strong reason why the pre-trained model by Ghosh is sensitive to less relevant information. RetinaFace produces a much better bounding box for a detected face and is closest to what a human would consider the entirety of the face. However, RetinaFace captures foreheads which is not a region with a large amount of facial information and can sometimes introduce more noise into the final image by capturing objects such as the brims of

hats. MediaPipe does not produce a bounding box, and instead produces the facial landmarks denoted in green on the image. A bounding box can be constructed by taking the minimal and maximal x- and y- coordinates of the landmarks found. MediaPipe produces the most restrictive bounding box of the backends, which minimizes noise and focuses the image only on regions with lots of discriminatory value; in other words, it captures facial regions with high information density. Thus, normalization using MediaPipe would result in much more feature-rich inputs into the neural network.

8.6.1.4. Resizing the Image

The final step for preprocessing images requires resizing images to a constant size in order for images to have the appropriate shape for input into the neural network. Zero padding, that is, padding the image with black pixels that have a value of zero, is preferable to stretching the image to fit the desired size. Image resizing requires interpolation methods which can add noise artifacts to the image, which padding avoids.

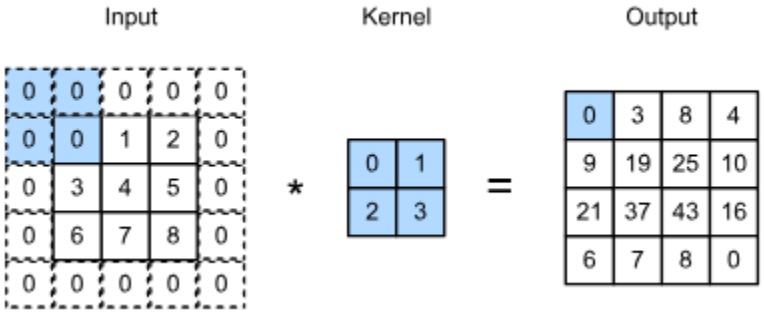

Figure 38. Convolution Operation (Zhang et al., n.d.)

Because convolution is functionally a composite operation of addition and multiplication, pixels with a value of zero effectively contribute nothing to the convolution calculation. This is useful, as it does not directly influence the training of the neural net, unlike interpolated values.

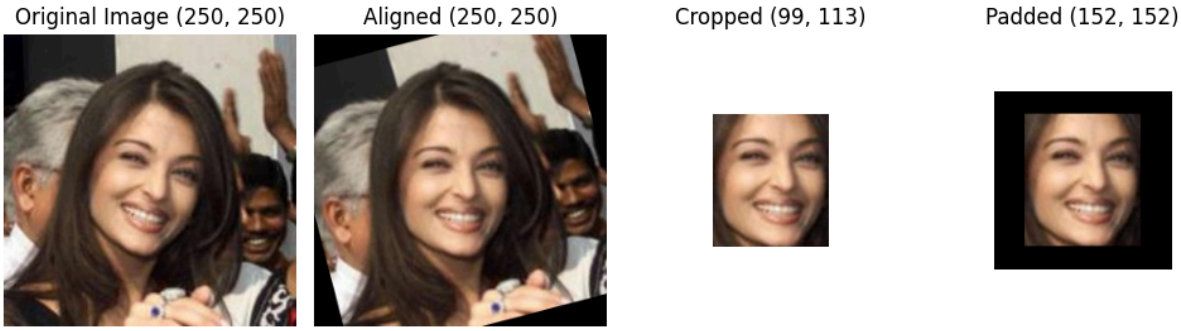

Figure 39. Full image preprocessing pipeline using dlib, MediaPipe, and zero padding

Ghosh's original preprocessing used the default settings of the dlib library, which produces images of shape (200, 200, 3) using bilinear interpolation. By using padding, no interpolation would be required for increasing the size of the image to the desired end shape. In all cases,

downsizing an image requires interpolation. The benefits of using a better cropping backend and resizing strategy would be to reduce the variance between images by eliminating extraneous information and preserving pixel data with as much fidelity as possible.

8.6.2. Generating a Normalized Dataset

Another method of obtaining a controlled dataset is to generate a normalized dataset from a non-normalized dataset such as LFW. Two of the biggest challenges in a non-normalized dataset are pose normalization and illumination normalization. Although beyond the scope of this paper to discuss, various methods for frontalization using 3D facial shapes have been shown to be effective in improving facial recognition accuracy (Hassner et al., 2014a), and a frontalized version of the LFW data set is available (Hassner et al., 2014b). Normalized illuminance has also been shown to greatly increase the effectiveness of facial recognition tasks (Shan et al., 2003). One or both of these methods would be useful in creating a normalized dataset for better training of the facial recognizer.

It is worth recognizing that these types of preprocessing steps may not be feasible on low-resource systems, and thus may not be viable for common use-cases. These normalization processes may also be unnecessary if the input data is already relatively clean, as is often the case when using ID photos for building security, one of the primary use-cases outlined for this project. However, normalized datasets have been shown to train better facial recognizers, and can be of great use for facial recognition in unconstrained conditions, such as in surveillance scenarios; so, these processes are worth mentioning despite being beyond the scope of this paper.

9. Appendix B: Mathematics and Equations

$$1.1\ \text{Euclidean, } L_2 \text{ distance: } d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

Euclidean distance formula. (n.d.)

The Euclidean or L_2 distance is the traditional measurement used for distances in the Riemann coordinate system. In the two-dimensional case, the distance can be represented with a triangle and the formula collapses to the familiar $a^2 + b^2 = c^2$.

$$1.2\ \text{Cosine distance: } d(p,q) = 1 - \frac{p \cdot q}{\|p\|\,\|q\|}$$

(Kumar, 2020)

Cosine distance is a measurement of the similarity of direction between two vectors. A low distance indicates a small angle between the vectors, which means they lie in the same general direction of coordinate space. Note that cosine distance is not a true distance from a mathematical standpoint, as it does not meet the triangle inequality.

$$1.3\ \text{cosine similarity, } \theta \text{ interior angle, } \cos(\theta) = \frac{p \cdot q}{\|p\|\,\|q\|}$$

(Kumar, 2020)

Cosine similarity is a measurement of the similarity of direction between two vectors. The only difference between cosine similarity and distance is the interpretation; two vectors pointing in the same direction exhibit high cosine similarity and low cosine distance, while two vectors pointing in perpendicular directions exhibit low cosine similarity and high cosine distance.

$$2.1\ \text{accuracy} = \frac{TP + TN}{TP + FP + FN + TN} = \frac{TP + TN}{N}$$

(Ahmed, 2023)

Accuracy is defined as the ratio of true positives (TP) and true negatives (TN) to the total number of samples N. On datasets with balanced classes, accuracy is a useful measurement of model effectiveness, but it produces misleading results on datasets with large class imbalances. Accuracy is measured on a scale from 0 to 1.

$$2.2\ \text{true positive rate (TPR), recall, sensitivity} = \frac{TP}{TP + FN}$$

(Chicco et al., 2021)

Recall is defined as the ratio of true positives (TP) to the sum of true positives and false negatives (FN). Recall is a measurement of the correctness of a model in capturing the underlying positive class. Recall is measured on a scale from 0 to 1.

$$2.3\ \text{true negative rate (TNR), negative recall, specificity} = \frac{TN}{TN + FP}$$

(Chicco et al., 2021)

Negative recall is the same as recall but for the negative class.

$$2.4\ \text{positive predictive value (PPV), precision} = \frac{TP}{TP + FP}$$

(Chicco et al., 2021)

Precision is defined as the ratio of true positives (TP) to the sum of true positives and false positives (FP). Precision is a measurement of the effectiveness of the model's positive predictions. Precision is measured on a scale from 0 to 1.

$$2.5\ \text{negative predictive value (NPV), negative precision} = \frac{TN}{TN + FN}$$

(Chicco et al., 2021)

Negative precision is the same as recall but for negative predictions.

$$2.6\ \mathrm{F}_\beta\ \text{measure/score} = \frac{(1 + \beta^2) \cdot \text{Precision} \cdot \text{Recall}}{\beta^2 \cdot (\text{Precision} + \text{Recall})}$$

(Steen, 2020)

F-Beta score is a weighted harmonic mean of precision and recall. Values of Beta greater than 1 place more emphasis on recall, while values of beta less than 1 place more emphasis on precision. For example, a Beta of 2 places twice the emphasis on recall, while a Beta of 0.5 places twice the emphasis on precision. By extension, as Beta goes to zero, recall is ignored, and as Beta goes to infinity, precision is ignored. F-Beta score is measured on a scale from 0 to 1.

$$2.7\ \mathrm{F}_1\ \text{measure/score} = \frac{2 \cdot \mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

(Steen, 2020)

F_1 score is a special case of the F-Beta score which places equal weight on both precision and recall and is the exact harmonic mean of precision and recall. The harmonic mean penalizes the higher of the two values and is maximized when both values are equal.

$$3.1\ \text{prevalence}\ (\phi) = \frac{TP + FN}{N}$$

(Chicco et al., 2021)

Prevalence is the total existence of the positive class within the data. Prevalence is a large influencer on a number of classification metrics, including precision, markedness, and MCC. Prevalence is classifier-independent and is only reliant on the underlying dataset. Datasets with low prevalence can struggle to achieve high values of these metrics, and in the context of facial recognition, prevalence is naturally very low. Prevalence is measured on a scale from 0 to 1.

$$3.2\ \text{bias}\ (\beta) = \frac{TP + FP}{N} = TPR \cdot \phi + (1 - TNR) \cdot (1 - \phi)$$

(Chicco et al., 2021)

Bias is the preference of a classifier to predict the positive class. Classifiers with high bias exhibit higher capture of the positive class (recall) but typically have low precision, markedness, and MCC scores as a result. Bias is measured on a scale from 0 to 1.

$$3.3\ \text{bookmaker informedness, informedness}\ (\mathrm{BM}) = TPR + TNR - 1$$

(Chicco et al., 2021)

Informedness is a measurement of a classifier's ability to capture the underlying positive and negative class. It is a balance of sensitivity and specificity, and classifiers with high levels of informedness are more "correct" but may not be particularly effective in their predictions. Informedness is measured on a scale from -1 to 1, where magnitude indicates level of correctness, and direction indicates proper labeling. Thus, an informedness of -1 indicates the classifier perfectly captures the classes as if their labels were reversed (positive identified as negative and vice versa).

$$3.4\ \text{balanced accuracy (BA)} = \frac{TPR + TNR}{2} = \frac{BM + 1}{2}$$

(Chicco et al., 2021)

Balanced Accuracy is equivalent to informedness on a different scale. Balanced Accuracy is measured on a scale from 0 to 1.

$$3.5\ \text{markedness (MK)} = PPV + NPV - 1$$

(Chicco et al., 2021)

Markedness is a measurement of the effectiveness of a classifier's positive and negative predictions. It is a balance of precision and negative precision, and classifiers with high levels of markedness tend to make more "trustworthy" predictions but may not be particularly correct in their predictions. Markedness is measured on a scale from -1 to 1, identically to informedness. Thus, a markedness of -1 indicates the classifier perfectly predicts the classes as if their labels were reversed, in a similar fashion as the scale for informedness.

$$3.6\ \text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN + FN)}}$$

$$3.7\ \text{MCC} = \sqrt{\frac{\phi - \phi^2}{\beta - \beta^2}} \cdot BM = \sqrt{\frac{\beta - \beta^2}{\phi - \phi^2}} \cdot MK = \pm\sqrt{MK \cdot BM}$$

(Chicco et al., 2021)

The Matthew's Correlation Coefficient, or MCC, is a measurement of the correlation between observed and predicted classifications. It ranges from -1 to +1, where positive levels of MCC indicate the observations and predictions are correlated, and negative levels of MCC indicate anti-correlation. The MCC is also the geometric mean of markedness and informedness. MCC is a balanced measure of the effectiveness of a classifier, although intermediate values with magnitude 0.1 to 0.9 are not strong indicators of whether markedness or informedness is high. MCC is useful for predicting overall classification health but not for diagnosing where the classifier struggles. MCC can also be described in terms of only markedness or informedness, along with prevalence and bias.

$$3.8\ \text{MK} = \frac{\phi - \phi^2}{\beta - \beta^2} \cdot BM$$

(Chicco et al., 2021)

Markedness can also be described in terms of prevalence, bias, and informedness.

10. Appendix C: Images and Visualizations

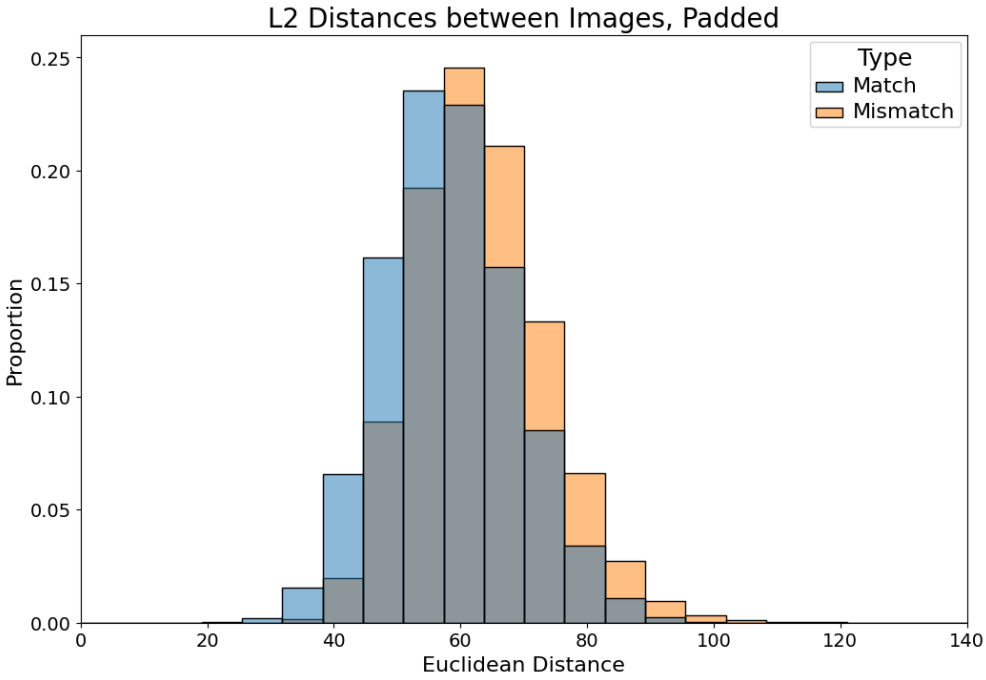

Figure 16b. Euclidean Distances between Images using MediaPipe + Padding Preprocessing

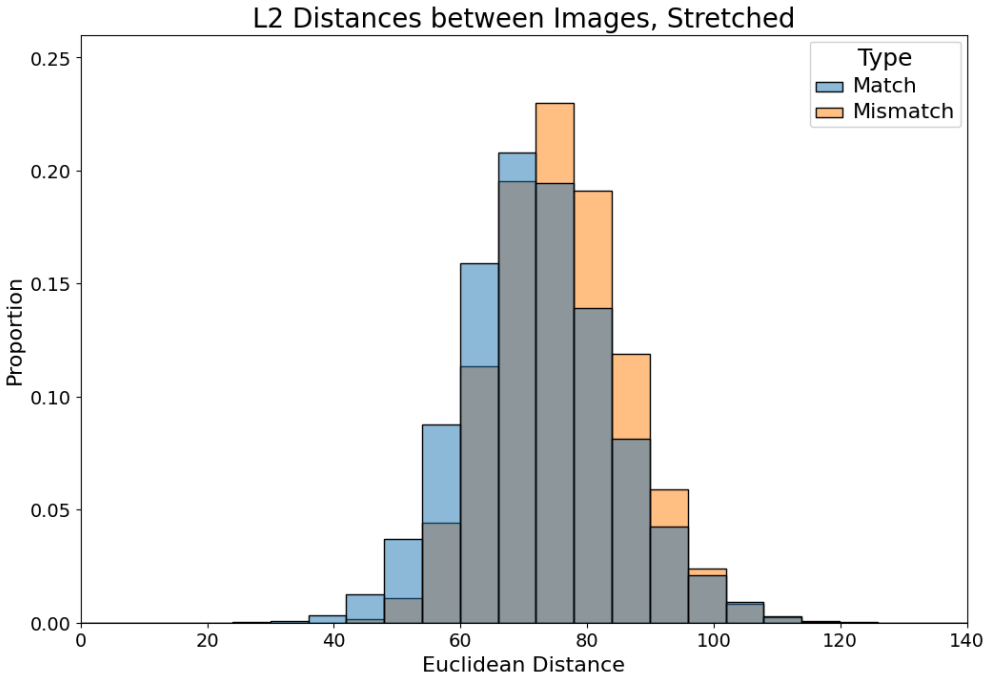

Figure 16c. Euclidean Distances between Images using MediaPipe + Stretching Preprocessing

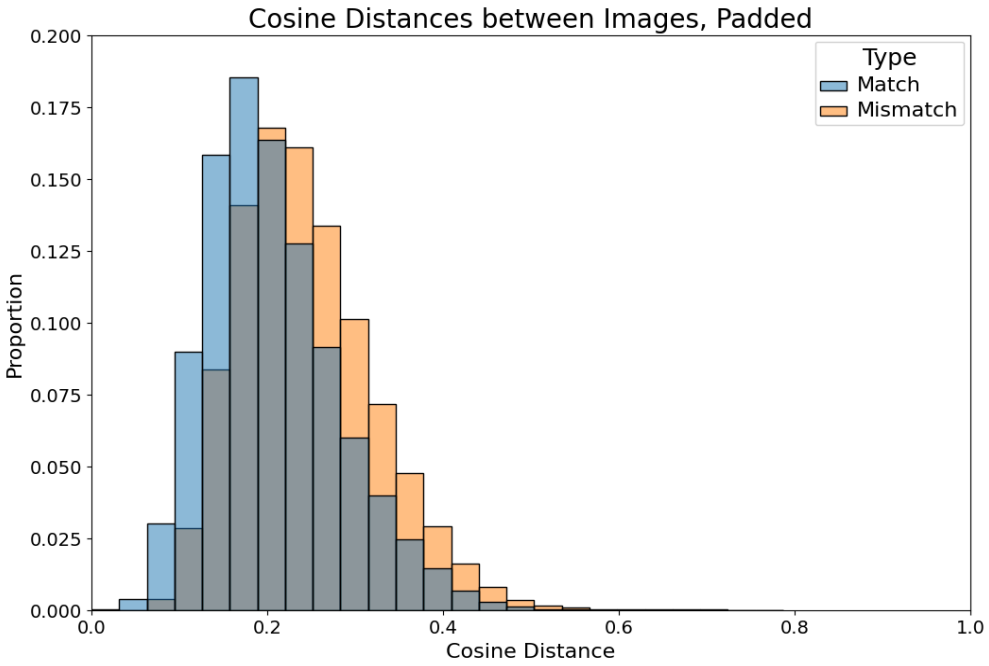

Figure 17b. Cosine Distances between Images using MediaPipe + Padding Preprocessing

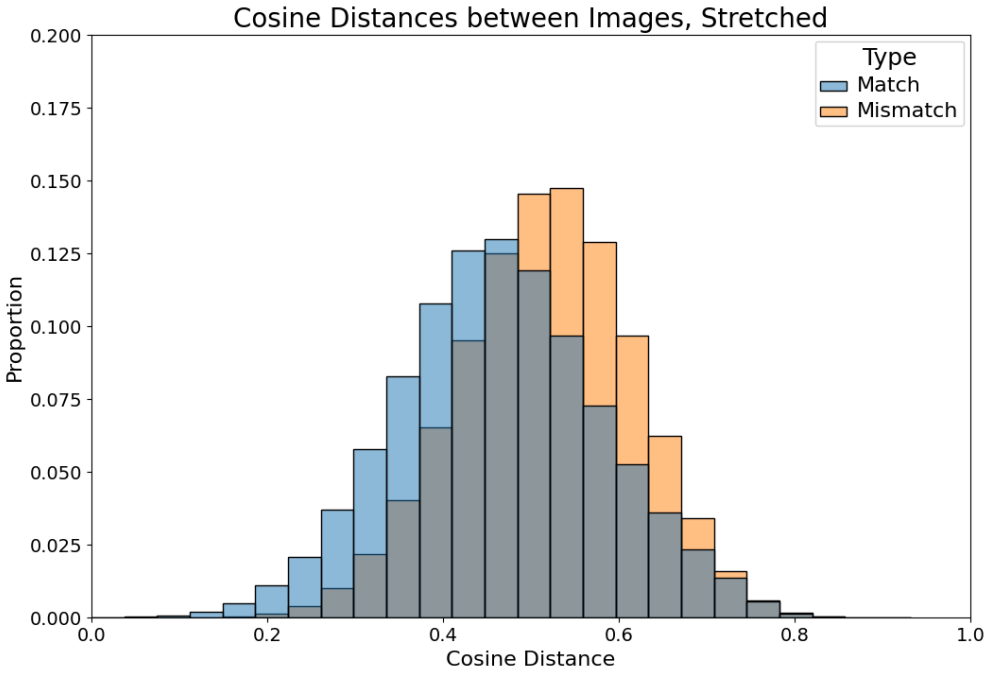

Figure 17c. Cosine Distances between Images using MediaPipe + Stretching Preprocessing

11. References

Ahmed, N. (2023, November). What is a confusion matrix in machine learning? The model evaluation tool explained. *DataCamp*. https://www.datacamp.com/tutorial/what-is-a-confusion-matrix-in-machine-learning

Brownlee, J. (2020, August 20). 10 clustering algorithms with python. *Machine Learning Mastery*. https://machinelearningmastery.com/clustering-algorithms-with-python/

Cao, Q., Shen, L., Xie, W., Parkhi, O., & Zisserman, A. (n.d.). VGGFace2 dataset [Data set]. University of Oxford. Retrieved July 19, 2024 from https://www.robots.ox.ac.uk/~vgg/data/vgg_face2/

Chicco, D., Tötsch, N., & Jurman, G. (2021). The matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. BioData Mining. Retrieved August 9, 2024 from https://biodatamining.biomedcentral.com/articles/10.1186/s13040-021-00244-z

davisking (2024). dlib-models. Retrieved July 19, 2024 from https://github.com/davisking/dlib-models

Deng, J., Guo, J., Zhou Y., Yu, Jinke., Kotsia, I., & Zafeiriou S. (2019). RetinaFace: Single-stage dense face localisation in the wild. Cornell University. Retrieved July 19, 2024 from https://arxiv.org/abs/1905.00641

dlib C++ library homepage (n.d.). dlib. Retrieved July 19, 2024 from http://dlib.net/

Emmery, C. (2017, March 25). Euclidean vs. cosine distance. *Chris Emmery*. https://cmry.github.io/notes/euclidean-v-cosine

Euclidean distance formula. (n.d.) Google. Retrieved July 19, 2024 from https://www.gstatic.com/education/formulas2/553212783/en/euclidean_distance.svg

Fbeta_score (n.d.). Scikit-learn. Retrieved August 9, 2024 from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html

Firebase. (n.d). Cloud Firestore | Firebase. Retrieved July 19, 2024, from https://firebase.google.com/docs/firestore

Ghosh, S. (2019). Pre-trained weights of vggface2 dataset. Retrieved July 19, 2024 from https://github.com/swghosh/DeepFace/releases

Gillis, A. (2020, September). Thick client (fat client). *TechTarget*. https://www.techtarget.com/whatis/definition/fat-client-thick-client

Google AI. (n.d.). Face landmark detection guide. Google. Retrieved July 19, 2024 from https://ai.google.dev/edge/mediapipe/solutions/vision/face_landmarker

Gottlieb, J., Aadhyanth, R., & Bolanos, C. (2024). Facial recognition with deepface github repository. Retrieved August 9, 2024 from https://github.com/JoshuaGottlieb/Facial-Recognition-with-Deepface

Hassner, T., Harel S., Paz E., & Enbar, R. (2014a). Effective face frontalization in unconstrained images. *IEEE Conf. on Computer Vision and Pattern Recognition*. Retrieved August 9, 2024 from https://arxiv.org/pdf/1411.7964

Hassner, T., Harel S., Paz E., & Enbar, R. (2014b). LFW3D. *IEEE Conf. on Computer Vision and Pattern Recognition*. Retrieved August 9, 2024 from https://talhassner.github.io/home/publication/2015_CVPR_1

Huang, G., Ramesh M., Berg T., & Learned-Miller E. (2007). Labeled faces in the wild: A database for studying face recognition in unconstrained environments [Data set]. University of Massachusetts. Retrieved July 19, 2024 from http://vis-www.cs.umass.edu/lfw/

Insightface (2024). Insightface: 2D and 3D face analysis project. Retrieved July 19, 2024 from https://github.com/deepinsight/insightface

Kanstrén, Teemu. (2020, September 11). A look at precision, recall, and f1-score. *TowardsDataScience*. https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec

Kivy homepage. (n.d.) Kivy. Retrieved July 19, 2024 from https://kivy.org/

Kumar, A. (2020, June 17). Cosine similarity & cosine distance. *DataDrivenInvestor*. https://medium.datadriveninvestor.com/cosine-similarity-cosine-distance-6571387f9bf8

Kundu, R. (2022, September 19). Precision vs. recall: Differences, use cases & evaluation. *V7*. https://www.v7labs.com/blog/precision-vs-recall-guide

Nagesh, B. (2022, September 4). Why accuracy is a bad metric for imbalanced datasets. *Medium*. https://binginagesh.medium.com/why-accuracy-is-bad-metric-for-imbalanced-datasets-7c1aad4cf12a

Saito, Takaya & Rehmsmeier M. (2015). The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalance datasets. PLOS ONE. Retrieved August 9, 2024 from https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0118432

Serengil, S. (2018, August 6) Deep face recognition with keras. *Sefik Ilkin Serengil*. https://sefiks.com/2018/08/06/deep-face-recognition-with-keras/

Serengil, S. (2020, May 22) Fine tuning the threshold in face recognition. *Sefik Ilkin Serengil*. https://sefiks.com/2020/05/22/fine-tuning-the-threshold-in-face-recognition/

Serengil, S. (2024). RetinaFace: Pip compatible implementation. Retrieved July 19, 2024 from https://github.com/serengil/retinaface/tree/master

Shan, S., Gao W., Cao B., & Zhao D. (2003). Illumination normalization for robust face recognition against varying lighting conditions. *IEEE International Workshop on Analysis and Modeling of Faces and Gestures*. Retrieved August 9, 2024 from https://www.cin.ufpe.br/~voa/Artigos/Illumination%20Normalization%20for%20Robust%20Face%20Recognition.pdf

Shashkina, V. (2024, February 16). Business guide to facial recognition: Benefits, applications, and issues to consider. *itrex Group*. https://itrexgroup.com/blog/facial-recognition-benefits-applications-challenges/

Steen, D. (2020, October 11). Beyond the f-1 score: A look at the f-beta score. *Medium*. https://medium.com/@douglaspsteen/beyond-the-f-1-score-a-look-at-the-f-beta-score-3743ac2ef6e3

Taigman, Y., Yang, M., Ranzato, M., & Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. *Meta Research*. Retrieved July 19, 2024 from https://research.facebook.com/file/266870805034649/deepface-closing-the-gap-to-human-level-performance-in-face-verification.pdf

What is facial recognition?. (n.d.). Amazon Web Services. Retrieved August 9, 2024 from https://aws.amazon.com/what-is/facial-recognition/

Zhang, A., Lipton, C., Li, Mu., & Smola, A. (n.d.) Chapter 7.3. padding and stride. *Dive into deep learning*. [eBook edition]. https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html