HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Capstone Project

Course: Biometric-IT4432E
2D Face Authentication System

CHU TRUNG ANH(Team leader)
anh.ct225564@sis.hust.edu.vn

BÙI DUY ANH
anh.bd225563@sis.hust.edu.vn

PHM MINH TIN
tien.pm225555@sis.hust.edu.vn

Supervisor:	Dr. Tran Nguyen Ngoc	______________
	Dr. Ngo Lam Trung	______________
		Signature
Department:	Computer Engineering	
School:	The School of Information and Communications Technology	

HANOI, 12/2024

ACKNOWLEDGMENT

The successful completion of this report would not have been possible without the invaluable contributions of several individuals and groups. I extend my heartfelt appreciation to my academic advisors, Dr. Tran Nguyen Ngoc and Dr. Ngo Thanh Trung, for their expert guidance and constructive feedback throughout the project and writing process. Their support was instrumental in shaping this work.

Additionally, I am grateful for the unwavering support of my family and friends. Their encouragement and understanding were essential in maintaining my motivation throughout this project.

The project may still contain mistakes or misunderstandings. If you encounter any, please don't hesitate to share them with us. We are always open to feedback and eager to listen.

ABSTRACT

This report presents the development and implementation of a cutting-edge 2D face recognition system, named VerifyMe, designed for accurate and efficient authentication. The project was developed for the Biometric Course IT4432E at HUST and explores two primary approaches: using a pre-trained model (FaceNet) for feature extraction followed by training a Support Vector Machine (SVM) for classification, and training a Siamese network architecture with an L1 Distance layer from scratch. The system is built with a comprehensive technology stack including Python, OpenCV, TensorFlow, PyTorch, and GUI frameworks such as PyQt6 and Kivy. The report details the design and architecture considerations, the data collection and preprocessing pipelines, model training processes, and the integration of the models into user-friendly applications. Additionally, it discusses the challenges faced during the development process, such as handling diverse datasets and optimizing model performance. The 2D Authentication System aims to provide secure and convenient access to applications or corporate systems, demonstrating the potential of advanced machine learning techniques in real-world biometric authentication scenarios.

For more details on implementation and installation guide, please visit our project Github repository: `https://github.com/chutrunganh/Biometric_IT4432E.git`

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

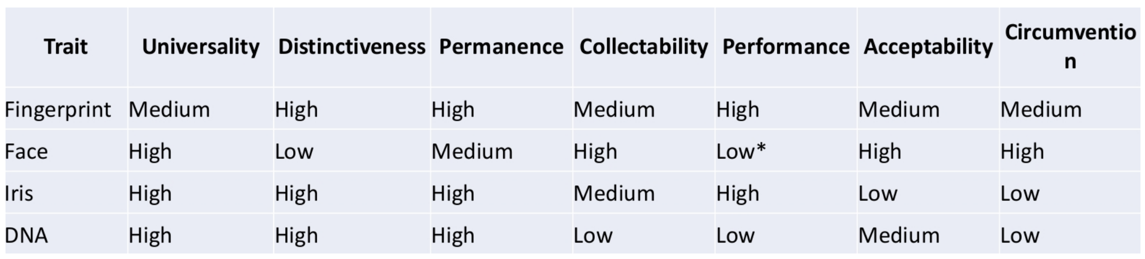
1.1 Brief Introduction

1.1.1 Why Choose Face Recognition?

Biometrics refers to the measurement and statistical analysis of people's unique features to recognize the identity, or verify the claimed identity. Biometric characteristics can be divided into two main categories:

- **Physiological Characteristics**: These are related to the shape of the body. Examples include fingerprints, face, DNA, iris, etc.
- **Behavioral Characteristics**: These are related to the behavior of a person. Examples include typing, gait, voice, signature,.

Each has its own advantages and disadvantages, with trade-offs between them. Here are some criteria to evaluate different types of biometric characteristics:



Trait	Universality	Distinctiveness	Permanence	Collectability	Performance	Acceptability	Circumvention
Fingerprint	Medium	High	High	Medium	High	Medium	Medium
Face	High	Low	Medium	High	Low*	High	High
Iris	High	High	High	Medium	High	Low	Low
DNA	High	High	High	Low	Low	Medium	Low

Figure 1.1: Biometric traits comparison

Although facial recognition has lower accuracy compared to other traits, it offers several advantages that align with our goals. It is easy to capture, can be captured from a large distance, and can be scaled for large-scale operations.

Facial recognition is the process of identifying and verifying individuals based on their facial features. For humans, this task is straightforward, even under varying environmental conditions, changes in age, or when wearing hats and glasses. However, for computers, it has been a challenging problem for decades.

In the era of artificial intelligence, leveraging the power of deep learning algorithms and vast amounts of data, we can create advanced models that represent faces as feature vectors in a high-dimensional space. This allows computers to recognize individuals with remarkable accuracy, sometimes even surpassing human capabilities.

Some common applications of facial recognition include

- **For Individuals**: 2D facial unlocking on mobile or PC devices (eg. Windows

Hello, Apple Face ID), attendance systems, security cameras, etc

- **For Government and Organizations**: access control system, tracking down criminals, managing and surveilling citizens, ...

Facial recognition technology has become an essential tool in various fields, providing both convenience and enhanced security.

There are two main tasks in face biometrics field:

- **Face Authentication / Verification**: Verify if the user is who they claim to be. In simple terms, answer the question ”Are you who you say you are?”
- **Face Identification**: Determine the identity of the user. In simple terms, answer the question ”Who are you?”

In this project, **we focus solely on the face verification task**.

1.2 Project Objective

Our primary objective in this project is to build a 2D biometric system. To achieve this, we have researched various documents to understand the concepts and implementation methods. From that knowledge , we approach the problem using two pipelines:

1. Using a pre-trained Model (**FaceNet**) for features extraction, then use those features to train a **Support Vector Machine** model for classification
2. Training a **Siamese network architecture + L1 Distance layer** from scratch

These approaches combine the convenience and accuracy of pre-trained models with the educational value of training custom architectures from scratch. After training, we use the models to build applications with a GUI application

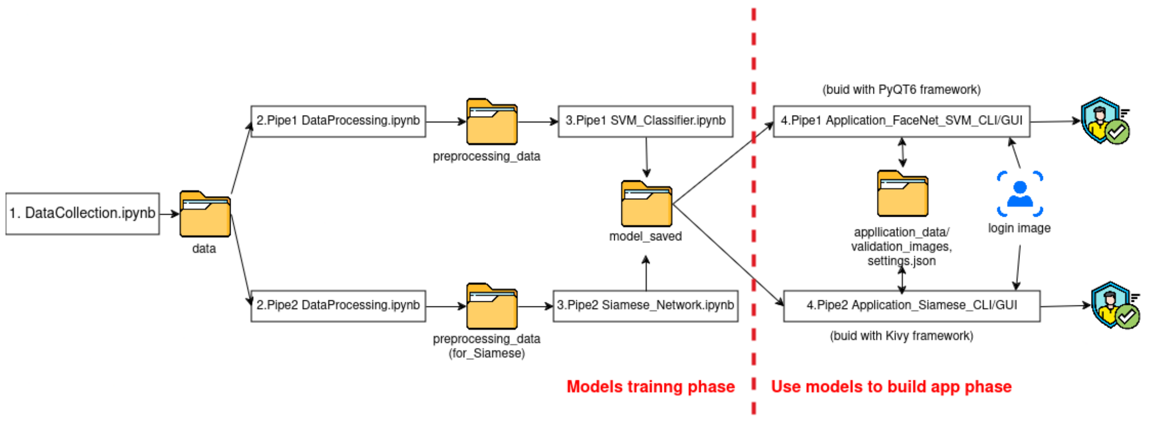
About technologies/ frameworks we used:

- For training AI model: numpy, pandas, matplotlib, seanborn, scikit-learn, keras, tensorflow, Pytorch
- For image processing: OpenCV
- For building GUI: PyQT6, Kivy
- Platform: tested for compatibility with Windows and Linux (requires Python 3.12 on system)

CHAPTER 2. PROJECT WORKFLOW

2.1 Intro to project workflow

Here is a summarized project workflow presented in the diagram below. We will delve into the details of each component:



1. DataCollection.ipynb
data
2.Pipe1 DataProcessing.ipynb
preprocessing_data
3.Pipe1 SVM_Classifier.ipynb
2.Pipe2 DataProcessing.ipynb
preprocessing_data (for_Siamese)
3.Pipe2 Siamese_Network.ipynb
model_saved
(buid with PyQT6 framework)
4.Pipe1 Application_FaceNet_SVM_CLI/GUI
appllication_data/ validation_images, settings.json
login image
4.Pipe2 Application_Siamese_CLI/GUI
(buid with Kivy framework)
Models trainng phase
Use models to build app phase

Figure 2.1: Project Workflow

The workflow consists of three primary phases:

1. Building the dataset
2. Training the model
3. Utilizing the trained model to develop the biometric system

In this Chapter 2, we will explore the two first sections, which aim to build and train AI models

2.2 Dataset

For the detail code implementation of this part, see `1.DataCollection.ipynb` *file*

When working on this face recognition project, one of our primary goals was to select a high-quality and diverse dataset. This step is critical for effectively training and evaluating our models. To achieve this, we employed three methods to gather data for our dataset.

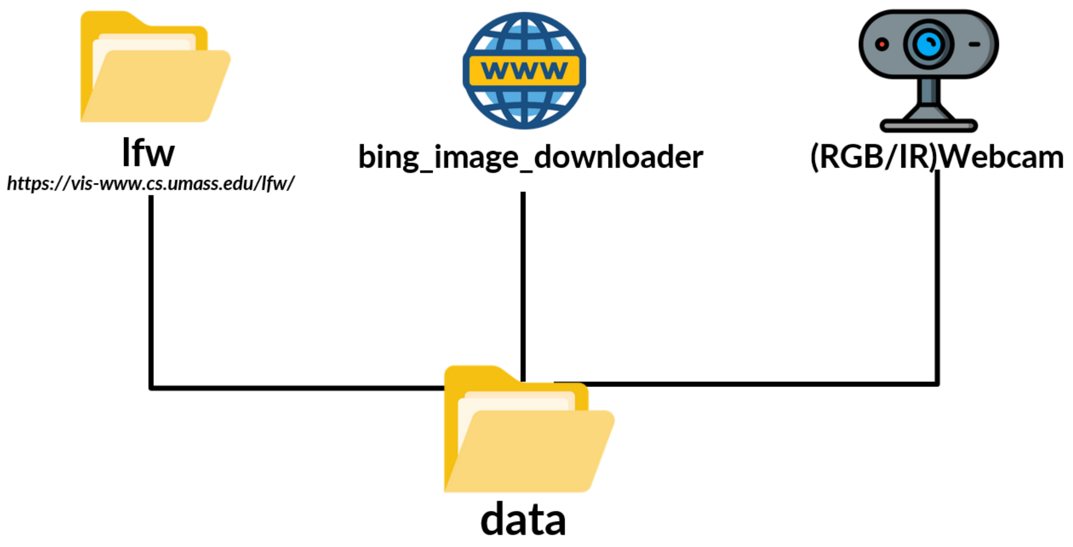


Figure 2.2: Three sources to get images for our dataset

The three methods used for data collection include: utilizing a prebuilt dataset, performing web scraping, and capturing real-world images using webcams.

This multi-source approach ensured that the dataset was both comprehensive and diverse, meeting the requirements for robust model training and evaluation. To maintain a consistent structure, the `data` folder contains several subfolders, each named after a person. Inside each subfolder are the images of that specific individual.

This hierarchical folder structure ensures clear organization and efficient access to images associated with each person.

```
data
└─person_1
│  └─person_1_001.jpg
│  └─person_1_002.jpg
│  └─...
└─person_2
   └─person_2_001.jpg
   └─person_2_002.jpg
   └─...
```

All images from these sources were standardized to a uniform format. Specifically, each image was resized to 250x250 pixels and saved in JPG format.

2.2.1 Prebuilt Dataset: Labeled Faces in the Wild (LFW)

We utilized the widely known *Labeled Faces in the Wild* (LFW) dataset, which contains a diverse collection of labeled face images. The dataset consists of 13,233 images of 5,749 individuals, size 205 MB after extraction. The dataset is organized into multiple subfolders, with each subfolder containing images of a specific person, named after that person. Below is an overview of how the dataset is structured:

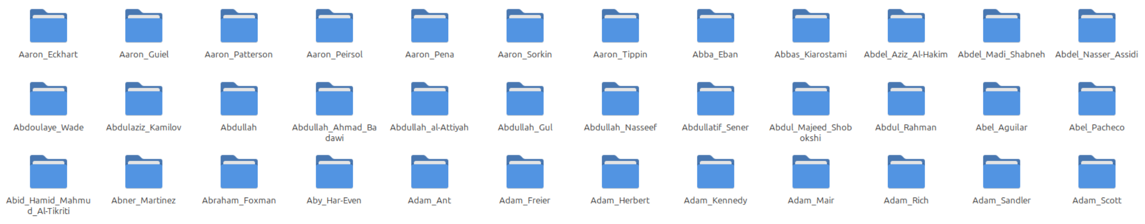


Figure 2.3: LFW Dataset Overview

The dataset can be downloaded via this link, then use this command to extract `tar -xvzf lfw.tgz`

2.2.2 Collecting data from the internet using web scraping

We utilized the `bing_image_downloader` library to automatically search for and download images based on specified keywords. By using a person's name as the keyword, the library searches for images of that individual and downloads them into a folder named after the person's name.

It is important to manually verify the downloaded images to ensure their quality and relevance. These additional images from the internet enhance the dataset by increasing its variety and providing updated data.

```python
persons = ["Robert Downey Jr", "Will Smith", "Tom Cruise",
    "Miu Shiromine"]

for person in persons:
    try:
        # Find and download images of that person
        downloader.download(person, limit=8, output_dir='
            crawl_data', adult_filter_off=True, force_replace
            =False, timeout=60)

        person_path = os.path.join('crawl_data', person)
```

Then we check these images manually again to ensure the data quality, resize them then drag and drop those folders in the main data folder `data`

2.2.3 Collecting Data from the Real World Using Webcam Devices

Our system is capable of detecting cameras connected to your computer. Typically, the built-in laptop camera has a `CAM_ID = 0`, the IR webcam has `CAM_ID = 2`, and an external webcam is assigned `CAM_ID = 4`. However, these IDs may vary depending on the system configuration.

We captured data from real-world sources by utilizing webcams to capture real images or by processing input from video streams. For external webcams, we used a smartphone camera connected to the laptop via *DroidCam*. Full setup details for this process can be found in the file `1.DataCollection.ipynb`.

2.2.4 Data Arrangement

To improve the model's ability to adapt to and recognize a person under various conditions, we perform data augmentation. This process helps enhance the robustness of the model by generating diverse variations of images, which makes the model more resilient to changes in lighting, angle, and other environmental factors.

For each person's folder within the dataset, we first count the number of images it contains. If a folder has more than three images, it is considered to have a high image density. In such cases, we apply fewer augmentations to avoid overfitting. On the other hand, if the image count is low, we perform more aggressive augmentation to increase the variety and volume of data, thereby improving the model's generalization capability. After the augmentation process, the size of training data increase nearly twice.

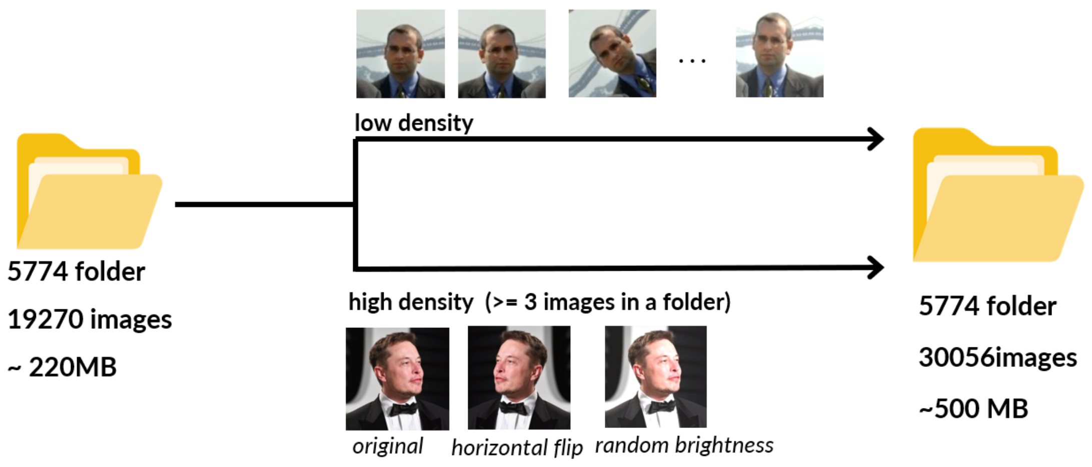


Figure 2.4: Data Argumentation

About all the augmentation operations we used:

```python
def create_augmentations(density='high'):
    """Create list of augmentations based on density"""
    if density == 'high':
        # Limited augmentations with high density
        augmentations = [
            RandomBrightnessContrast(p=1.0),
            HorizontalFlip(p=1.0),
        ]
    else:
        # More augmentations: Brightless,Horizontal Flip,
           Rotate, Zoom, Vary color, ...
        augmentations = [
            RandomBrightnessContrast(p=1.0),
            HorizontalFlip(p=1.0),
            Rotate(limit=60, p=1.0),
            ShiftScaleRotate(shift_limit=0.3,scale_limit
               =0.3,rotate_limit=20,p=1.0),
            HueSaturationValue(p=1.0),
        ]
    return augmentations
```

2.3 Pipeline 1: FaceNet + SVM

2.3.1 Data preprocessing for Pipeline 1

For the detail code implementation of this part, see `2.Pipeline1 DataPreprocessing.ipynb` file

In this part, we take the images from the `data` folder for processing. The output of this step is the `preprocessed_data` folder, which will later be used to train the SVM model. By using an intermediate file like this, we aim to maintain the modularity and maintainability of the code, such that changes in one file will not affect the others.

Create a `preprocessed_data` folder in the same location as the `data` folder. The `preprocessed_data` folder should contain two subfolders: `face_detect` and `face_embeddings`.

The pipeline look something like this:

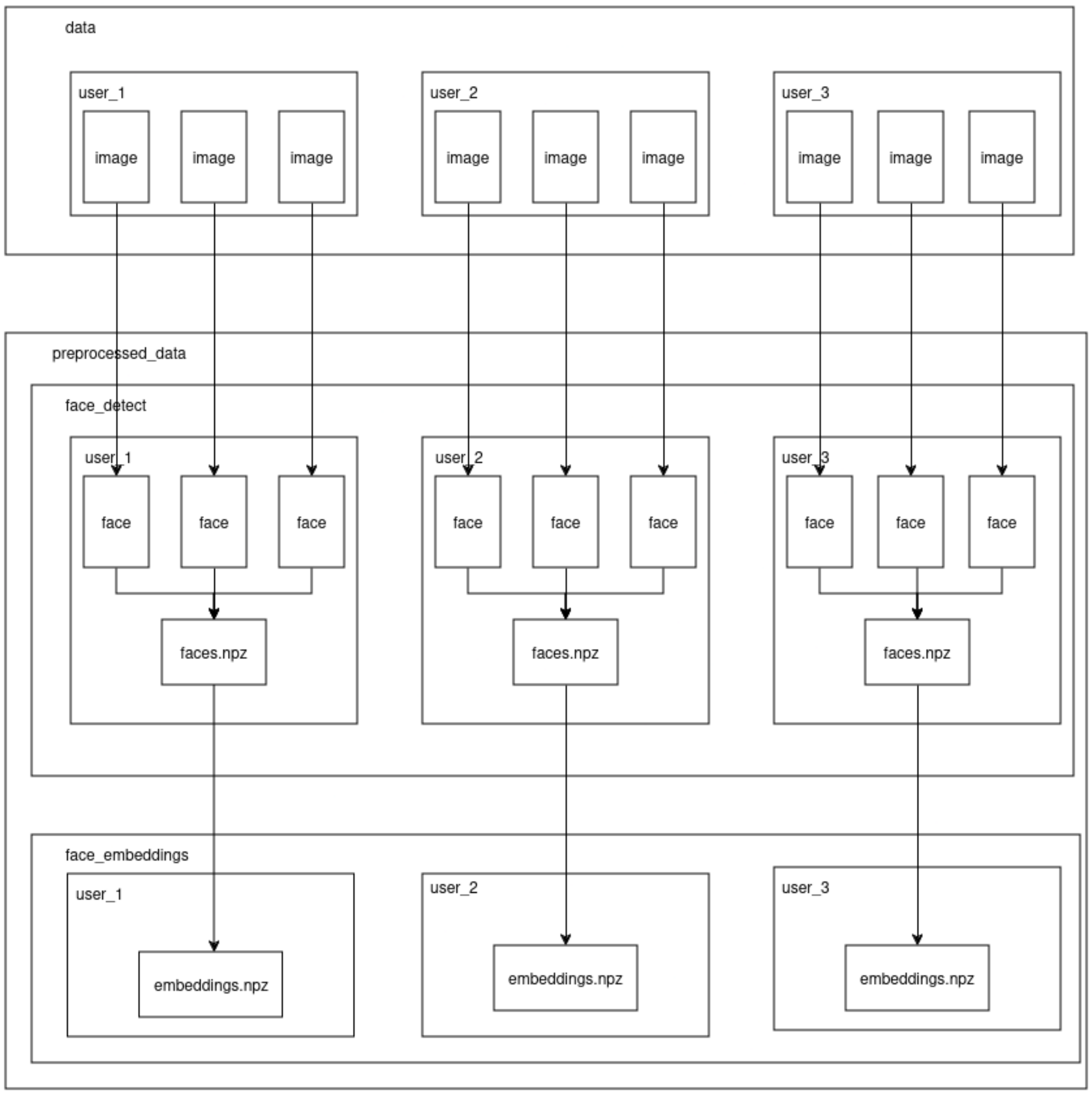
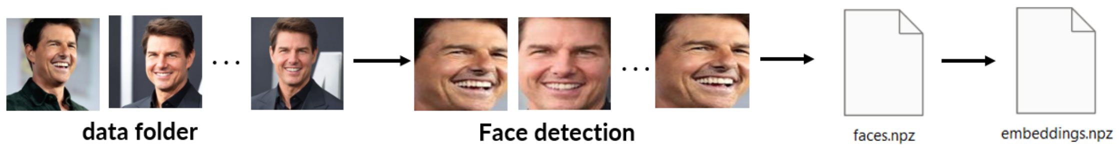


Figure 2.5: Pipeline1 Workflow



a, face_detect folder

The *face_detect* folder will contain the images with the faces extracted from the original images. Inside, there are many subfolders named after each person. Each subfolder contains the images of faces detected then cropped from that person's original images. Additionally, all face images inside each subfolder are compressed into a single `faces.npz` file. (Note: The face images are primarily for visualization; the model works with the numpy arrays in the `.npz` file.).

For this detection task in our system, we will be employing the MTCNN algorithm, which stands for Multi-Task Cascaded Convolutional Neural Network. This deep learning algorithm is commonly used for face detection and facial recognition tasks. MTCNN uses a cascading approach to detect faces and facial features such as the eyes, nose, and mouth.

Some other models are usually used for this task can be considered: Dlib (CNN), Dlib (HOG), Haar cascade,...See detail in this link

A .npz file is just a compressed file format used to store multiple NumPy arrays. We use this format instead of working directly with images to improve storage efficiency and reduce data access time.

b, face_embeddings folder

The*face_embeddings* folder will contain the `embeddings.npz` files that store the embeddings of the faces. Each `.npz` file contains the embeddings of a specific person's faces. The embeddings are generated by a pre-trained model, in this case, we use the **FaceNet** model from PyTorch

FaceNet is a deep learning model developed by Google in 2015 for face recognition. It maps facial images to a compact Euclidean space where the distance between points corresponds to their similarity. FaceNet generates embeddings, which are numerical representations of faces, enabling tasks like face recognition, verification, and clustering. The model is trained using a triplet loss function, which ensures that faces of the same person are closer in the embedding space, while faces of different people are farther apart. FaceNet is known for its high accuracy, efficiency, and scalability in face-related tasks.

Other models to consider include VGGFace, DeepFace, and DeepID, among others. For more details, check out the following links: this article, FaceNet, and VGGFace.

2.3.2 SVM (Support Vector Machine) Classifier Model

For the detailed code implementation of this part, see the `3.Pipeline1 SVM_Classifier.ipynb`*file or the online Kaggle version of this notebook.*

After we got extracted face features from `2.Pipeline1 DataPreprocessing.ipynb` (in form of vector), we have many way to deal with face vitrifaction problem. Some several methods are:

- Compute directly the distance between two face vectors (from a input image when login and the registered images). This way is simple without any training step, but it is not robust and not general.

- Train a classifier model to classify if two vectors are considered as similiar or not (meaning the input image is the same person with the registered image). This way is more robust and general, but it requires a training step. We seen it as a binary classification problem. Given input as a pair of face vectors, the model will output 1 if the two vectors are considered as similar, and 0 otherwise. Some common models for this problem are SVM, K Nearest Neighbors, Naive Bayes, or even more robust deep learning models.

In this task, we try **SVM (Support Vector Machine)** model to solve the problem. In reality scenario, SVM often used for this kind of task since it is particularly effective in high-dimensional spaces, which is typical for face feature data.

Here's the pipeline for preparing the data for training a classifier in the face verification process:

1. **Load Embeddings**:
   - Iterate through the subfolders in the `face_embeddings` directory.
   - Load the facial embeddings from the `embeddings.npz` files in each subfolder.
2. **Create Positive Pairs**:
   - For each person's embeddings, create pairs of embeddings by combining all possible pairs inside that person's embeddings.
3. **Create Negative Pairs with Random Sampling**:
   - Randomly select a limited number of embeddings from other subfolders to create negative pairs.
   - Limit the number of samples to pair in each subfolder and the number of subfolders to sample from.
   - This limits the number of negative pairs to consider, as trying all combined pairs would be approximately $n \cdot (n-1)/2$, where n is the number of embeddings in the dataset, which could overwhelm computational resources.
4. **Generate Labels**:
   - Assign label 1 to positive pairs (same person).
   - Assign label 0 to negative pairs (different persons).
5. **Prepare Training Data**:
   - Combine the pairs and their corresponding labels into arrays.

- Shuffle the data.
- Divide the data into training and testing sets.

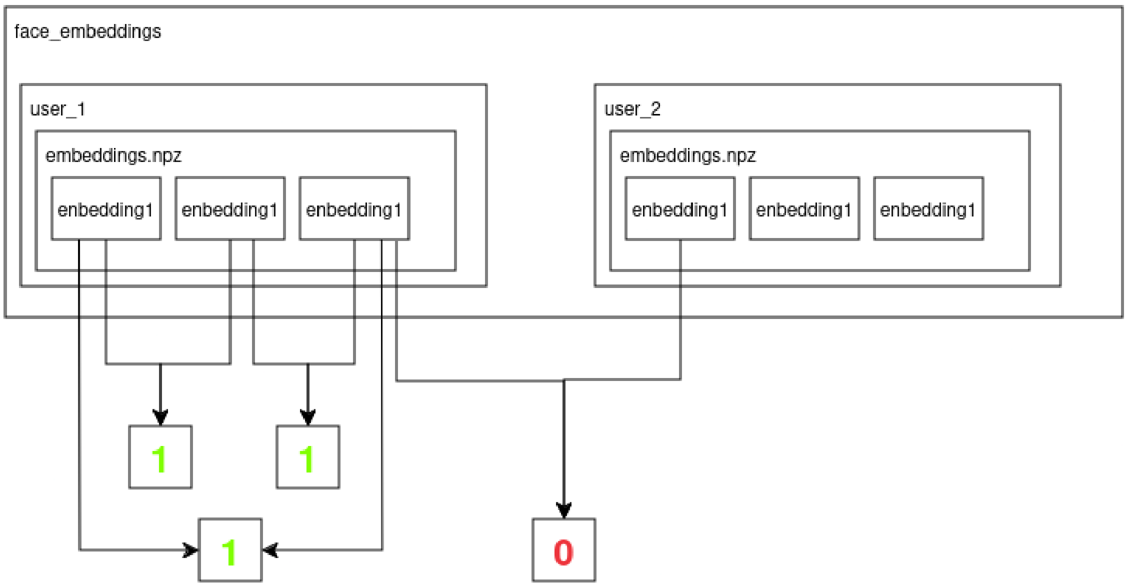


Figure 2.6: Create training pairs for SVM

Balancing Positive and Negative Pairs in Training Data

To ensure effective training, pay close attention to the ratio of positive to negative pairs. A balanced dataset is generally ideal for good training performance. Here's how different imbalances can affect the model:

- **Positive Pairs** $>>$ **Negative Pairs:**
  - Results in lower precision (more false positives) and higher recall (better at finding matches).
  - The system becomes more lenient, favoring user convenience but reducing security.
- **Positive Pairs** $<<$ **Negative Pairs:**
  - Results in higher precision (fewer false positives) and lower recall (misses some matches).
  - The system becomes stricter, favoring security but making matches harder to accept.

Practical Considerations: In real-world face verification applications, negative pairs often far outnumber positive pairs. However, the balance can be adjusted based on specific requirements. In our code implementation: the number of positive pairs is fixed, but the number of negative pairs is adjustable via a parameter to control the balance as needed.

a, Envaluation

After trainig the model, here is the reulst:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.73	0.53	0.61	36
1	0.90	0.95	0.92	153
Accuracy	0.87 (Overall)			
Macro Avg	0.81	0.74	0.77	189
Weighted Avg	0.86	0.87	0.86	189

Table 2.1: Classification Report for the Face Verification Model

However, due to time constraints and limited computational resources, we were only able to train the SVM model on a very small subset of the dataset we built. Additionally, we could not perfectly balance the ratio between positive and negative pairs as mentioned, resulting in overall low performance. This implementation is intended purely for demonstration purposes.

2.4 Pipeline 2: Siamese network architecture + L1 distance

2.4.1 Data preprocessing for Pipeline 2

For the detail code implementation of this part,
see `2.Pipeline2 DataPreprocessing.ipynb` file

After this preprocessing step, result in `preprocessed_data(for_Siamese)` folder, we use these data to train a the Siamese network architecture with L1 distance

The process is quite similar to the `2.Pipeline1 DataPreprocessing.ipynb` notebook; however, there are notable differences after the face detection step:

- In `2.Pipeline1 DataPreprocessing.ipynb`, we used a pre-trained model to extract face embeddings. The pre-trained model is directly called to predict embeddings without additional training since it is already trained.
- While in the Siamese Network approach, we aim to train the model to extract features from scratch. Here, we only detect faces and pass them to the Siamese Network, which learns to generate feature embeddings during training.

Face Detection Process:

The process for detecting ten cropped faces from an image still uses the MTCNN model (as discussed in the previous section). However, there are minor differences in this step:

- The image is resized to **100x100 pixels** to match the architecture of the Siamese Network (details discussed in the next section).
- A **Gaussian filter** is applied to the image to smooth it.

2.4.2 Siamese Network

For the detailed code implementation of this part, see the `3.Pipeline1 Siamese_Network.ipynb` *file or the online Kaggle version of this notebook.*

a, Short explain how Siamese architecture work

Training Process:

- Pairs of images are fed through identical twin networks (twin, that is why it is called Siamese).
- The network learns feature extraction from scratch and generates an embedding (feature vector) for each image.
- The distance between two feature vectors is computed to determine the similarity score.
- Based on the label:
  - **Label = 1 (Same person):** The similarity score should be high.
  - **Label = 0 (Different people):** The similarity score should be low.
- If the similarity score does not match the expected label, the network back-propagates and updates its weights to improve performance.

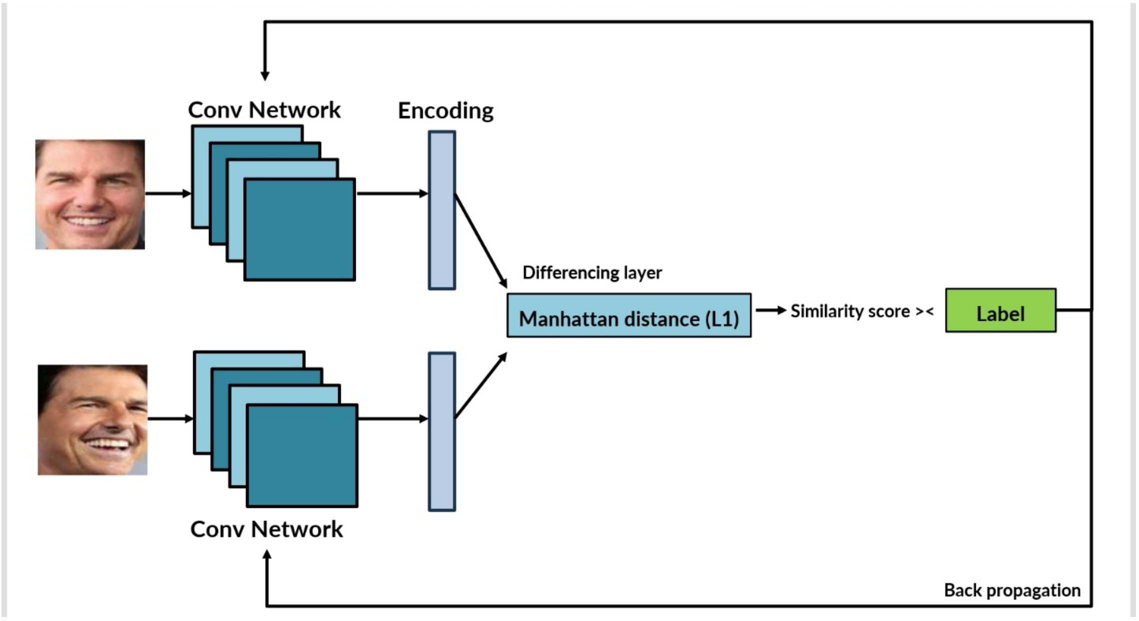


Figure 2.7: How Siamese network architecture work

b, Create trainng pairs

First, from the preprocess data, we create training pairs for Siamese:

- **Positive pairs:** Two images of the same person.
- **Negative pairs:** Two images of different people.

The method for creating pairs is similar to the approach described in the previous section. Additionally, to improve efficiency, we group every 16 pairs into a single batch for processing.

(Still need to pay a close attention to the ratio between positive and negative pairs as mentioned)

c, Building layers

Here is the Siamese Network architecture we are going to build:

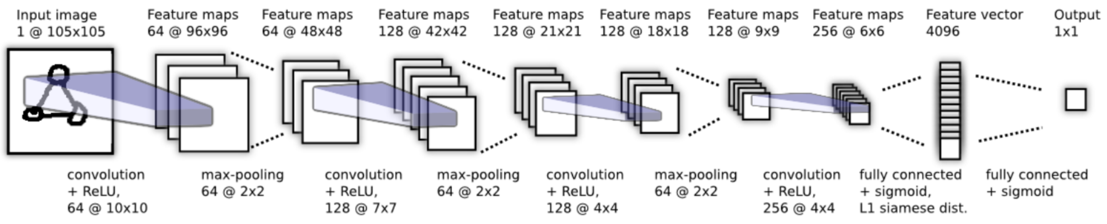


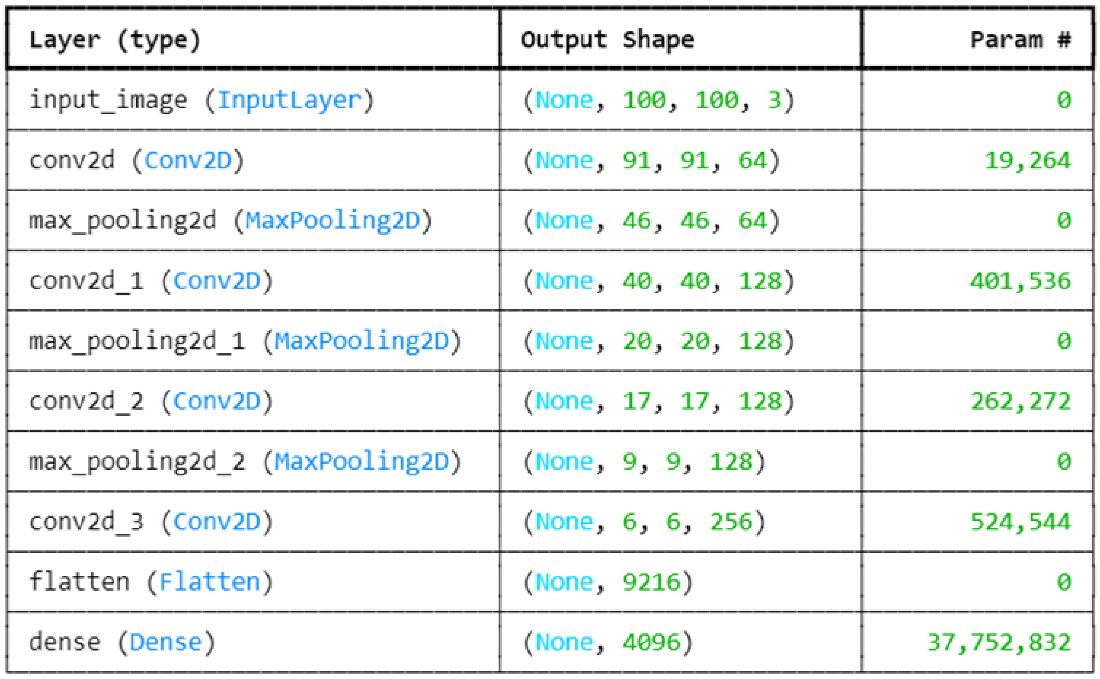
Figure 2.8: Enter Caption

This architecture is taken from this paper

We build this architecture using base layers offered in TensorFlow

```
def make_embedding() :
    #Create the input layer
    inp = Input(shape=(100, 100, 3), name='input_image') # 100x100 image size, 3 channels color

    ### First Block ###
    # Next layer is a convolutional layer 64 filters, kernel size of 10x10, and ReLU activation
    c1= Conv2D(64, (10,10), activation='relu')(inp) # c1 stands for convolutional layer 1
    # Next layer is a max pooling layer with a pool size of 2x2
    m1 = MaxPooling2D(64, (2,2), padding='same')(c1)

    ### Second Block ###
    # Next layer is a convolutional layer 128 filters,
kernel size of 7x7, and ReLU activation
    c2 = Conv2D(128, (7,7), activation='relu')(m1)
    # Next layer is a max pooling layer with a pool size of 2x2
    m2 = MaxPooling2D(64, (2,2), padding='same')(c2)

    ### Third Block ###
    c3 = Conv2D(128, (4,4), activation='relu')(m2)
    m3 = MaxPooling2D(64, (2,2), padding='same')(c3)

    ### Fourth Block ###
    c4 = Conv2D(256, (4,4), activation='relu')(m3)
    f1 = Flatten()(c4) # Flatten the output of the convolutional layer to feed it to the dense layer
    d1 = Dense(4096, activation='sigmoid')(f1) # Dense layer with 4096 neurons and sigmoid activation

    return Model(inputs=[inp], outputs=[d1], name='Siamese__embedding_layer')
```

Here is the summary of the model after build:



Layer (type)	Output Shape	Param #
input_image (InputLayer)	(None, 100, 100, 3)	0
conv2d (Conv2D)	(None, 91, 91, 64)	19,264
max_pooling2d (MaxPooling2D)	(None, 46, 46, 64)	0
conv2d_1 (Conv2D)	(None, 40, 40, 128)	401,536
max_pooling2d_1 (MaxPooling2D)	(None, 20, 20, 128)	0
conv2d_2 (Conv2D)	(None, 17, 17, 128)	262,272
max_pooling2d_2 (MaxPooling2D)	(None, 9, 9, 128)	0
conv2d_3 (Conv2D)	(None, 6, 6, 256)	524,544
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 4096)	37,752,832

Figure 2.9: Model layers summary

The model will take input as a image size 100x100, color channel 3 (Color in

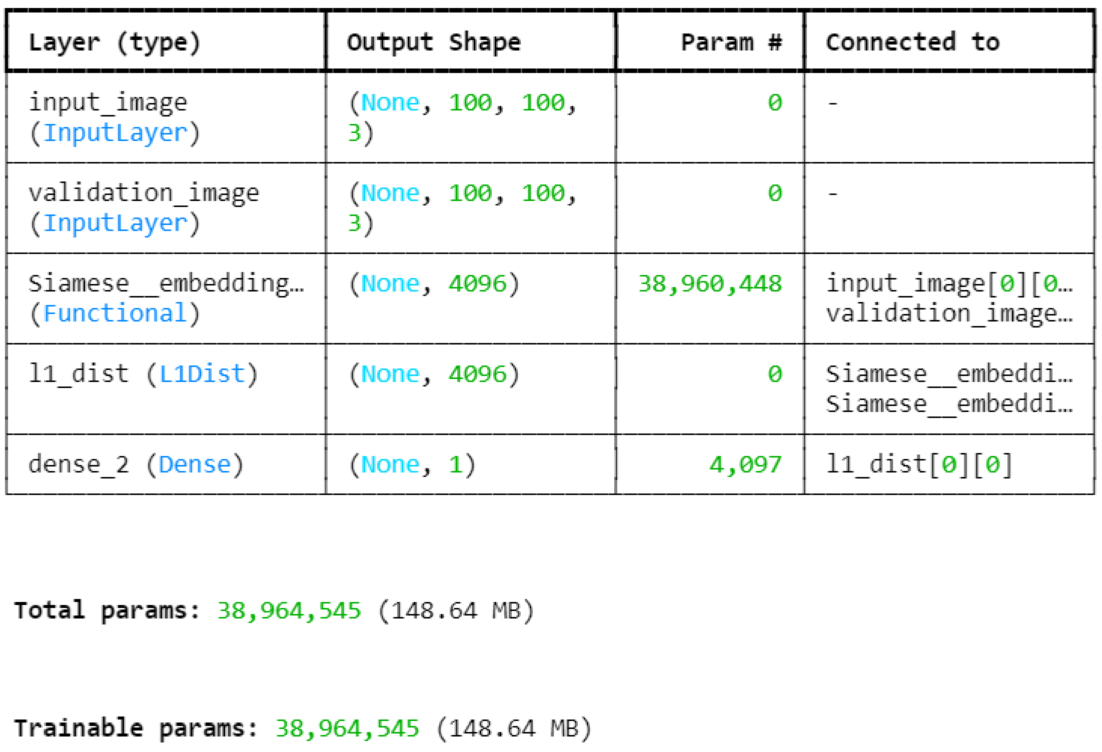
3 channels, can be RGB, BRG or even just grayscale but duplicated 3 times to match the format), then from that image, it will extract features, return a features vector with 4096 dimensions. (the **None** parameter indicates that model can accept batches of any size

After that, to compare two feature vectors to see how similar our two images are using L1 distance (or also called Manhattan distance). The L1 distance is the sum of the absolute differences between the two feature vectors.

Since there is no L1 distance layer in Keras, we will create a custom layer to calculate the L1 distance. The custom layer will take two feature vectors as input and output the L1 distance between them.)

```python
class L1Dist(Layer):
    def __init__(self, **kwargs):
         super(L1Dist, self).__init__(**kwargs)

    def call(self,input_embedding, validation_embedding):
        # Convert inputs to tensors otherwise will meet
           error: unsupported operand type(s) for -: 'List'
           and 'List'
        input_embedding = tf.convert_to_tensor(
           input_embedding)
        validation_embedding = tf.convert_to_tensor(
           validation_embedding)
        input_embedding = tf.squeeze(input_embedding, axis
           =0)  # Remove potential first dimension
        validation_embedding = tf.squeeze(
           validation_embedding, axis=0)

        # Calculate and return the L1 distance
        return tf.math.abs(input_embedding -
           validation_embedding)
```

Fully Siamese + L1 layer summary after build:



Layer (type)	Output Shape	Param #	Connected to
input_image (InputLayer)	(None, 100, 100, 3)	0	-
validation_image (InputLayer)	(None, 100, 100, 3)	0	-
Siamese__embedding… (Functional)	(None, 4096)	38,960,448	input_image[0][0… validation_image…
l1_dist (L1Dist)	(None, 4096)	0	Siamese__embeddi… Siamese__embeddi…
dense_2 (Dense)	(None, 1)	4,097	l1_dist[0][0]

```
Total params: 38,964,545 (148.64 MB)

Trainable params: 38,964,545 (148.64 MB)
```

Figure 2.10: Siames + L1

Look at the summary table, we see that the input of the model is two images size 100,100,3. The feature extraction vector (output of the Embedding layer) is **4096-dimensional**. The output is a **single value**, which is the distance between the two images, then pass to sigmoid function to get binary output

d, Training

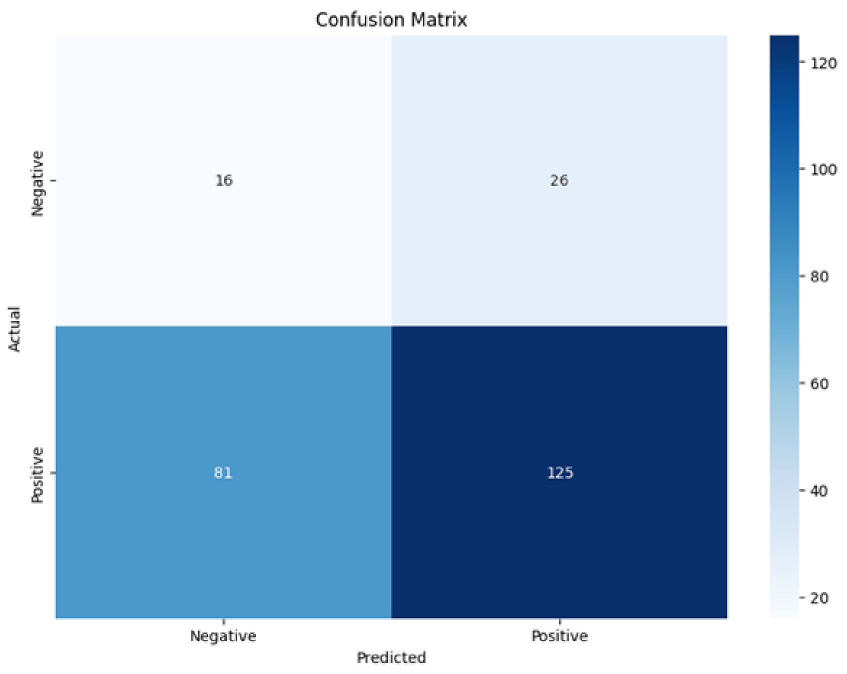
In this step, we defined actual training steps. We train on one batch of data, one batch of data come through our training step, we go on making prediction → calculate our loss function → calculate gradient then apply back propagation (calculate new weights and apply) through our neutral network to get the best possible model

```python
# Initialize early stopping parameters
patience = 4
best_val_loss = float('inf')
patience_counter = 0

@tf.function  # Annotation to indicate that complies to
    TensorFlow graph execution
def train_step(batch):
    with tf.GradientTape() as tape:
        # Get the pairs and labels from the batch
        pairs, labels = batch
        input_images, validation_images = pairs[:, 0], pairs[:, 1]
        # Forward pass
        predictions = fully_siamese_network([input_images, validation_images], training=True)
        # Set training=True for is important since some layers will only activated when this is set to True
        # Calculate the loss
        loss = binary_cross_loss(labels, predictions)
    # Calculate the gradients
    gradients = tape.gradient(loss, fully_siamese_network.trainable_variables)
    # Update the weights
    optimizer.apply_gradients(zip(gradients, fully_siamese_network.trainable_variables))
    # Adam is a variant of stochastic gradient descent, it applies the learning rate and gradient to slightly reduce the loss function, unit ll
    # it really near the minimum value.

    return loss
```

After that, we build the training loop:

```
def train(train_data, val_data, EPOCHS):
    global best_val_loss, patience_counter
    for epoch in range(1, EPOCHS + 1):
        print(f'\nEpoch {epoch}/{EPOCHS}')
        progressBar = tf.keras.utils.Progbar(len(train_data))
        epoch_loss = 0
        # Training loop
        for idx, batch in enumerate(train_data):
            loss = train_step(batch)
            epoch_loss += loss.numpy()
            progressBar.update(idx + 1, [('loss', loss.numpy())])
        avg_train_loss = epoch_loss / len(train_data)
        # Validation loop
        val_loss = 0
        for batch in val_data:
            loss = validate_step(batch)
            val_loss += loss.numpy()
        avg_val_loss = val_loss / len(val_data)
        print(f'Epoch {epoch} - Loss: {avg_train_loss:.4f}
           - Val Loss: {avg_val_loss:.4f}')
        # Check for early stopping
        # when the validation loss is not improving, we
           stop the training
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            patience_counter = 0
            # Save the best model
            checkpoint.save(file_prefix=checkpoint_prefix)
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print("Early stopping triggered.")
                break
```

e, Envaluation



Precision: 0.8278145685364238

Recall: 0.6067961165048543

However, due to time constraints and limited computational resources, we were only able to train the SVM model on a very small subset of the dataset we built.

Additionally, we could not perfectly balance the ratio between positive and negative pairs as mentioned, resulting in overall low performance. This implementation is intended purely for demonstration purposes.

3.1 Main idea

For the detail code implementation of this part, see the following source file

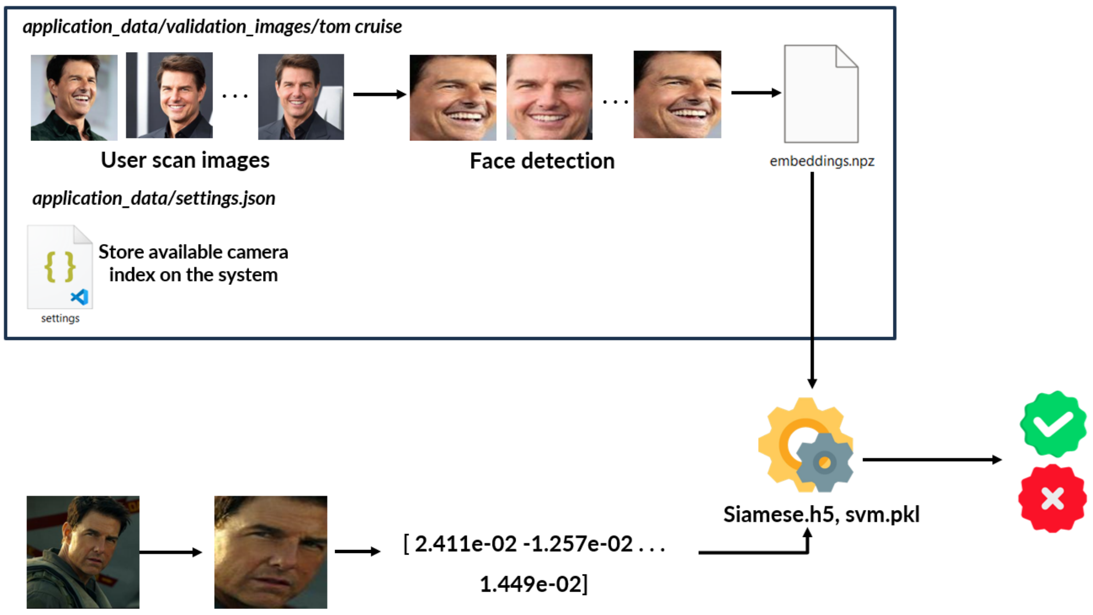
- `4.Pipeline1 Application_FaceNet_SVM_CLI.ipynb`
- `4.Pipeline1 Application_FaceNet_SVM_GUI.py`
- `4.Pipeline2 Application_Siamese_CLI.ipynb`
- `4.Pipeline2 Application_Siamese_GUI.py`

For each pipeline, we have both CLI and GUI version fot that

We have completed training some models and stored them in the `model_saved` folder. Now, we load these models and use them to verify the face of a person.

The process is as follows:

1. The user registers their face in the system through a scanning process. After scanning the images, we extract the face, then extract the face embeddings and store them in the system.
2. When the user wants to verify their face, we open the camera, capture the image, extract the face embeddings, and compare them with the embeddings stored inside the system (using the corresponding name provided during login).


application_data/validation_images/tom cruise
User scan images
Face detection
embeddings.npz
application_data/settings.json
Store available camera index on the system
settings
Siamese.h5, svm.pkl
[2.411e-02 -1.257e-02 . . . 1.449e-02]

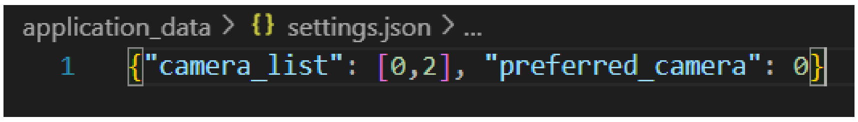
3.2 Enrollment

3.2.1 Camera detection

First of all, we need to discover the camera index of the user's system so that we can open the camera frame window using OpenCV. Each device has a different camera ID/index, so it is necessary to find the correct camera ID for our device. To achieve this, we loop through a range of camera IDs and ask the user to check if the camera is working. Since each device can have multiple webcams, resulting in multiple corresponding camera IDs, we also ask the user to select their preferred camera ID.

We store these configurations in the `application_data/settings.json` file. **The next time the user opens the app, these settings will be loaded automatically, without asking user again.**

Content inside `settings.json`:



application_data > {} settings.json > ...

```
{"camera_list": [0,2], "preferred_camera": 0}
```

Figure 3.1: Example content of the `settings.json` file.

Notably, during testing, we observed platform-specific behavior. On Windows, only the RGB camera is detected when performing camera detection. However, on Linux, both the RGB and IR cameras are detected, even though the tests were conducted on the same laptop equipped with two cameras.

3.2.2 Scan Validation Images

** Validation images can also be referred to as gallery, reference, or enrollment images in other documentation.*

The process involves collecting personal images of a person for the verification or enrollment process through a webcam. This is similar to the initial sign-in process in Windows Hello, where you need to scan your face for the first time. These images are stored and later used to compare with the input image each time the user logs in.

After successfully connecting to the host camera, the system requires the user to capture their images. The following steps are performed:

- Use MTCNN to detect and crop the face from the captured image.
- If no face is detected, display a message to the user.

- Store the cropped face images in `faces.npz`.
- Depending on the pipeline:
  - **Pipeline 1:** Use FaceNet to extract features and store them in `embeddings.npz`.
  - **Pipeline 2:** Skip feature extraction, as this task will be handled by the Siamese model in the next step.

All generated data is stored in `application_data/validation_images`. The directory structure is as follows:

```
application_data
├─ settings.json
└─ validation_images
   └─ user1
      ├─ face
      │  └─ faces.npz
      ├─ embeddings
      │  └─ embeddings.npz
      ├─ image1.jpg
      ├─ image2.jpg
      └─ ...
```

3.3 Login/Verify

** Login images can also be referred to as probe, query, or input images in other documentation.*

The process for user verification is as follows:

- **Access webcam:** Capture the input image of the user when they attempt to log in. This input image is processed **directly** as openCV frame object without storing in any file.
- **Verify against positive samples:** Compare the input image with a set of positive samples, which are the images collected during the enrollment process. These positive samples, also referred to as *validation images*, are stored in the `application_data/verification_images` folder. Each user has their own subfolder within the `verification_images` directory.

For each input image, the system iterates through all the positive samples. For instance, if there are 50 positive images in the `validation_images/user_name` folder, the verification function will output 50 predictions. Each comparison between a input image and a verification image (from the 50 images in the folder) generates a score between 0 and 1. To determine if that input image matches a positive sample, it must higher than a **detection threshold** we choose.

After obtaining the 50 scores, we apply a **verification threshold** to decide the minimum number of matching images required for valid authentication. For example, if the verification threshold is set to 0.8, then at least 80% of the 50 images must match the input image for it to be considered a successful login.

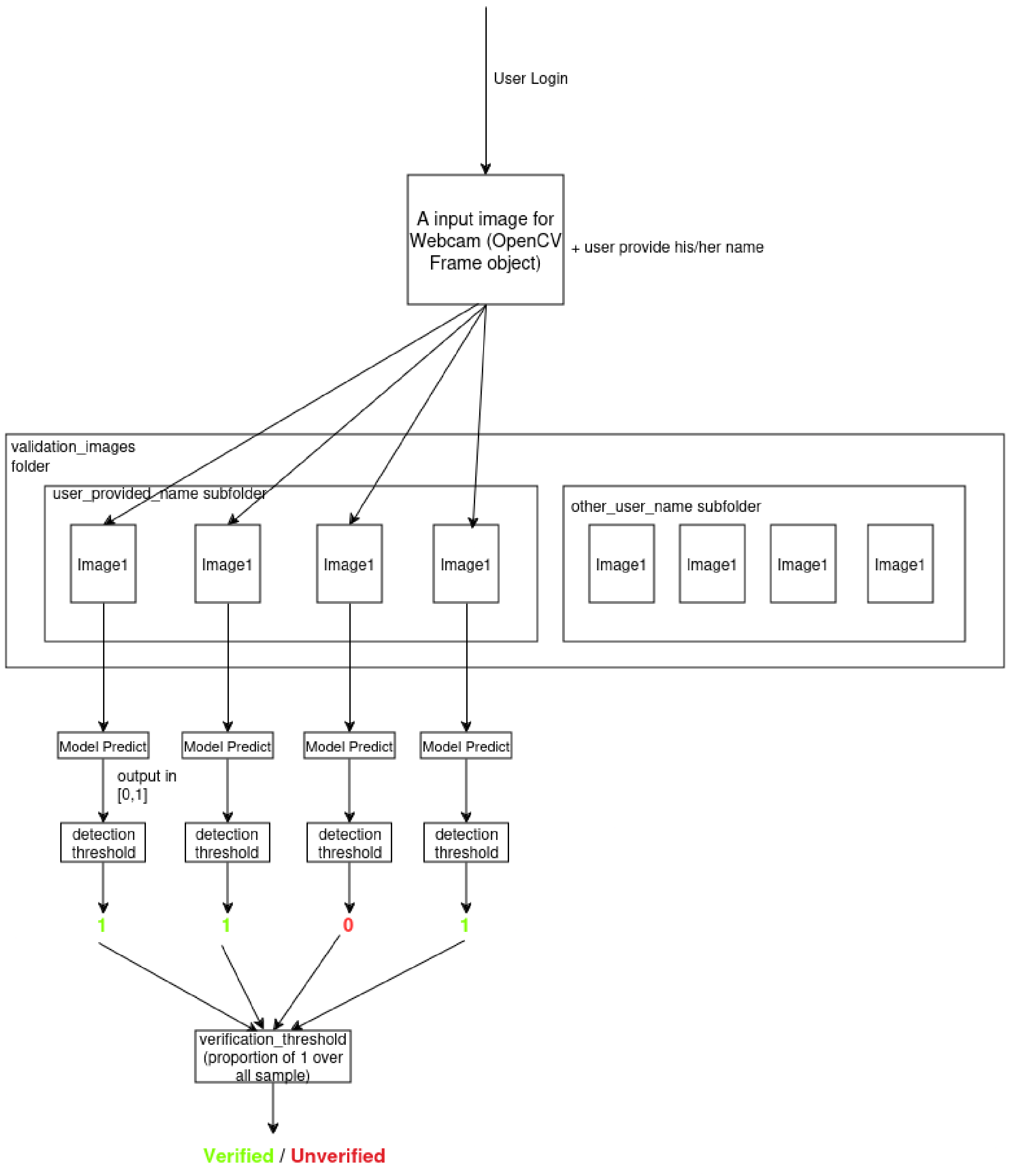


Figure 3.2: Verification process during user login.

CHAPTER 4. INSTRUCTION GUIDE

4.1 Instruction Guide

4.1.1 Only run the application file

Follow the instructions, see the project's GitHub release page. This file includes the GUI and the trained models necessary to run the application:

`https://github.com/chutrunganh/Biometric_IT4432E/releases/tag/v1.0.0`

4.1.2 Run entire project locally

To explore the full project including data preprocessing, training, and application. Follow our instruction below.

1. **Clone the repo**

```
git clone https://github.com/chutrunganh/Biometric_IT4432E.git
```

2. **Install dependencies**

   Navigate to the project folder:

```
cd REPLACE_WITH_YOUR_PATH/Biometric_IT4432E
```

With Linux

```
# Activate python virtual environment
python3 -m venv venv
source venv/bin/activate

# Install pip tool if you haven't already
sudo pacman -Syu base-devel python-pip # With Arch-based
# sudo apt update && sudo apt upgrade -y && sudo apt install build-essential python3-pip  # With Debian-based, use this command instead
pip install --upgrade pip setuptools wheel

# Install all required dependencies
pip install -r requirements_for_Linux.txt
```

With Windows

```
python -m venv venv
```

```
.\venv\Scripts\activate.bat # If execute in CMD
# .\venv\Scripts\activate.ps1 # If execute in PowerShell

# Install pip tool if you haven't already
python -m ensurepip --upgrade
pip install --upgrade pip setuptools wheel

# Install all required dependencies
pip install -r requirements_for_Windows.txt

# Install ipykernel in your virtual environment
pip install ipykernel
python -m ipykernel install --user --name=venv --display
    -name "Python (venv)" # Create a new kernel for
    Jupyter
```

Choose the kernel named `venv` when running Jupiter Notebook. It may take about 15-30 minutes to download all dependencies, depending on your internet speed.

> **Important:** This project requires **Python 3.12.x**. Some other versions, such as 3.10.x, have been reported to have compatibility issues with dependencies.

3. **Follow the code files**

   Follow the code files from 1 to 4 (you can choose to just follow Pipeline1 or Pipeline2) and read the instructions, run the code inside these files to generate and process data. Note that this is a pipeline, so do not skip any files; otherwise, errors will occur due to missing files.

4.2 Application UI

Here is the detail describe for the UI running from `Pipeline1 Application_FaceNet_SVM_GUI.py`. This UI is build with PyQT6 framework.

Another UI for `4.Pipeline2. Application_Siamese_GUI.py`, is developed using the Kivy framework. However, it currently supports only the Verify/Login process and is still unstable for deployment.

4.2.1 Camera detection window

This *Camera Detection* window will appear after running the code for the first time.

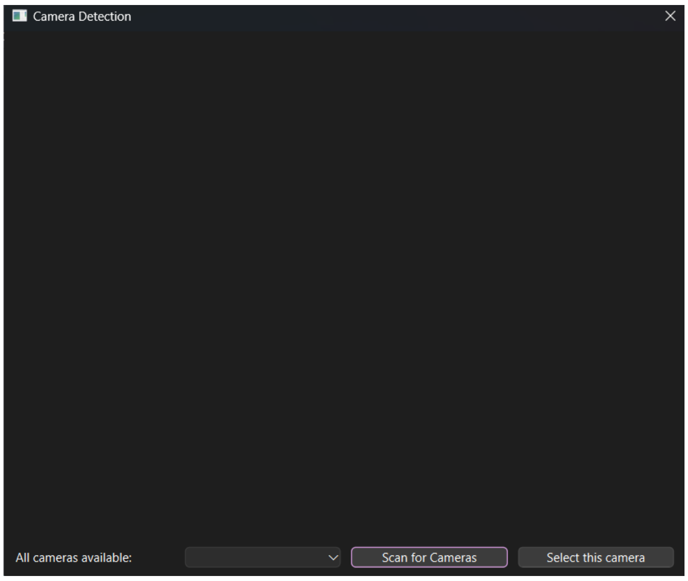
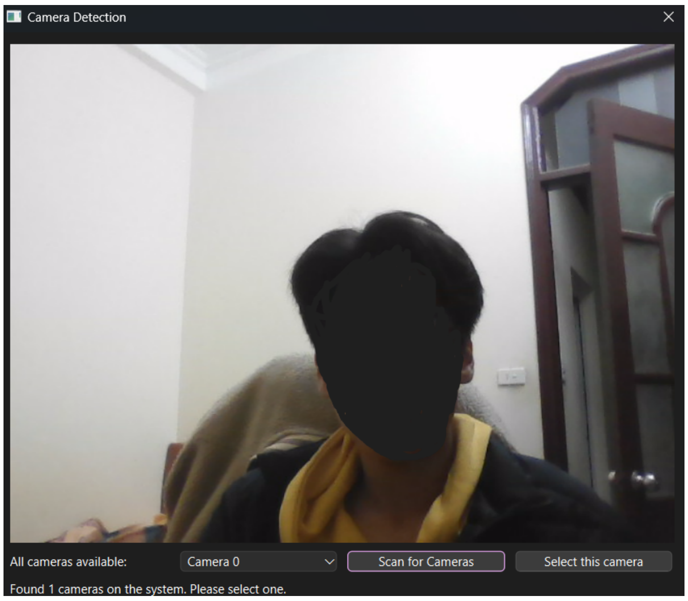


Figure 4.1: UI after running the code for the first time.

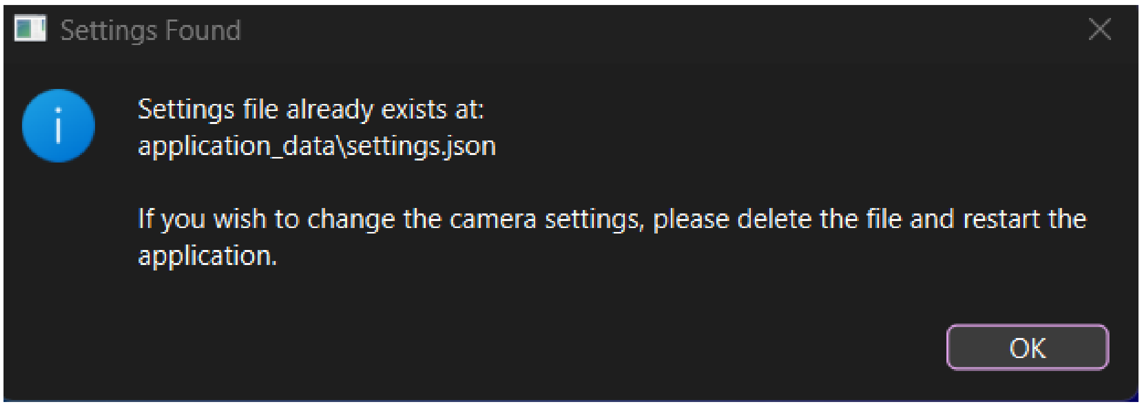
Click *Scan for Cameras* button, the program will detect any existed cameras you have on your computer, show valid camera options in the Combo box. You can switch between cameras to try. Next, click on the button *Select this camera* to select camera you want to use among all available cameras.



Found 1 cameras on the system. Please select one.

Figure 4.2: UI after detect camera

These settings will be saved in the file `application_data/settings.json`. The next time the user runs the application, if this settings file already exists in the system, the camera detection window will no longer appear. Instead, a popup window will be displayed.



Settings Found

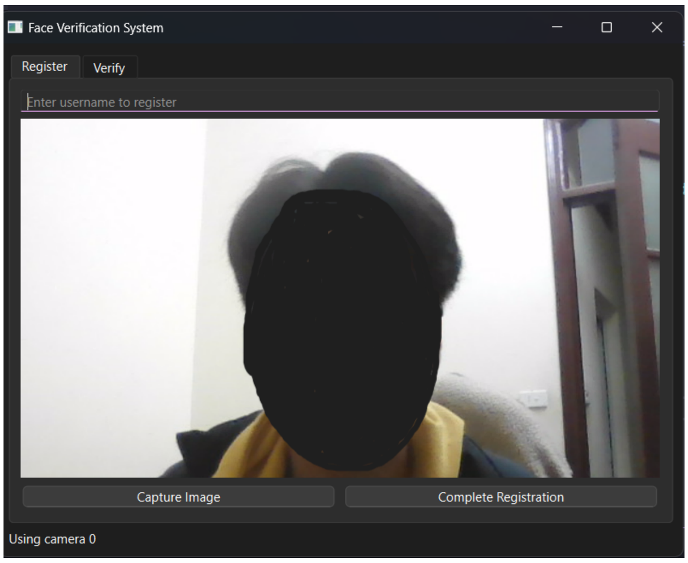
Settings file already exists at:
application_data\settings.json

If you wish to change the camera settings, please delete the file and restart the application.

Figure 4.3: Cameras are already recognized

4.2.2 Registration window

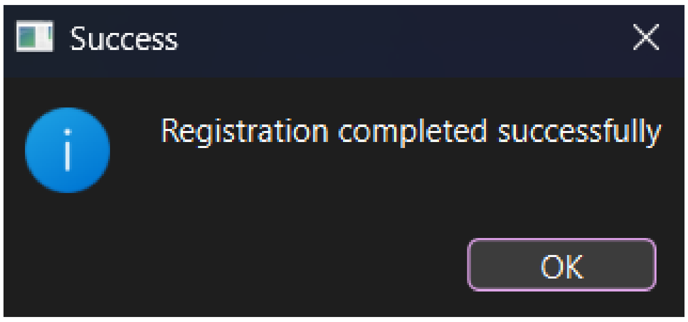
First, enter your username, then click the *Capture Image* button to take a picture of yourself. Each time you press the button, an image will be captured. We strongly recommend capturing these images under good lighting conditions.



[redacted]

Figure 4.4: UI during registration

Then click *Complete Registration* button to finish the registration process.



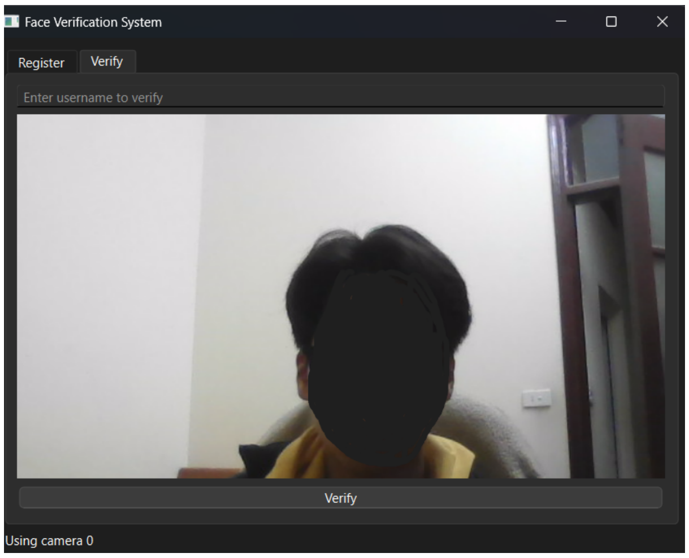
Success

Registration completed successfully

Figure 4.5: Registration successfully

4.2.3 Verification window

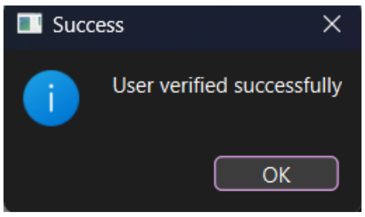
First, enter your username and click the *Verify* button to verify.



[redacted]

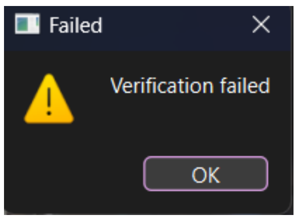
Figure 4.6: UI during verification

These are two notifications following as a successful and unsuccessful face recognition by the system.



User verified successfully

(a) Success



Verification failed

(b) Failed

Note that PyQT6 relies on the system settings to render graphics, so if your system is in light mode, it may appear differently, like this:

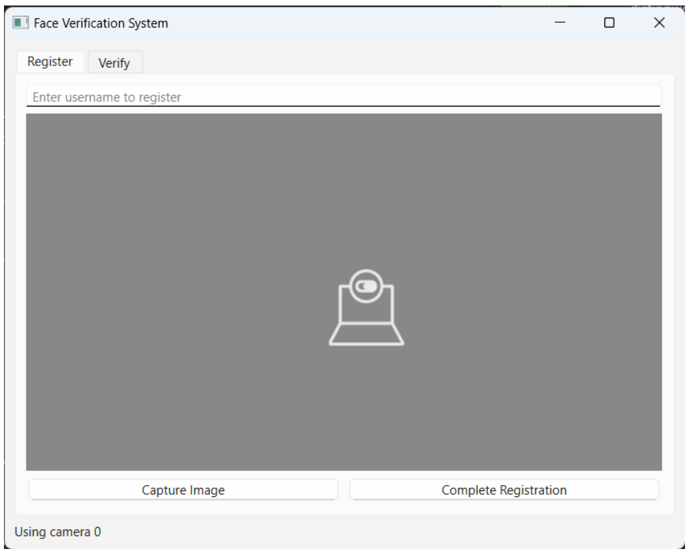


Figure 4.8: Enter Caption

CHAPTER 5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

In conclusion, the development and implementation of the VerifyMe 2D face recognition system have demonstrated the potential of advanced machine learning, deep learning, computer vision and image processing techniques in enhancing biometric authentication. By leveraging both pre-trained models and custom-trained Siamese networks, the project has achieved a robust and efficient authentication mechanism. Throughout the project, various challenges were encountered, including handling diverse datasets and optimizing model performance, which were systematically addressed to improve the system's accuracy and reliability. The VerifyMe application provides a secure and convenient solution for accessing applications or corporate systems, showcasing the practical applications of biometric authentication in real-world scenarios.

5.2 Future Improvements

Although the project has met the minimum goals, due to time constraints, we were unable to fully explore its potential. There is still room for future work.

- **Deploy Using Docker, Package in .exe, .deb for End-User:** Utilize Docker for containerization to simplify deployment across different environments. A test Docker file is already included in the project, but we have not yet resolved the issue of mapping the physical host camera to the camera inside the container. Additionally, package the application as an executable (.exe for Windows) or a Debian package (.deb for Linux) to make installation easier for end-users.
- **Trying Other Models to Optimize Price-Performance:** Experiment with different models to achieve a better balance between performance and cost. This could involve reducing processing time and utilizing less powerful resources such as cameras and processors. The goal is to make the system more efficient without compromising accuracy.
- **Build a More User-Friendly UI:** Focus on designing a UI that is easier for users to interact with. This could include improving navigation, simplifying controls, and providing clear feedback to enhance the overall user experience.
- **Collect More Quality Data to Reduce Biases:** Gather a more diverse and high-quality dataset to improve the model's accuracy and generalization. This will help reduce biases that may occur from a limited or skewed dataset, en-

suring fairer results across different demographics.

- **Handle Variations Like Glasses, Masks, Make-up, etc.:** Improve the model's robustness to variations in facial features, such as glasses, masks, make-up, and other accessories. This will help increase the system's reliability in real-world scenarios where these factors are common.
- **Quality Control: Reject Images that are Too Bright, Dark, or Unrecognizable:** Implement a quality control mechanism to automatically reject images that are too bright, too dark, or cannot be recognized properly by the system. This will ensure that only high-quality images are processed, improving overall performance.
- **Implement Multi-Instance Verification for Login Images:** Allow multiple instances of a user's image to be verified at once. This feature would help enhance accuracy, especially in cases where the initial image might have been blurry or partially obscured.
- **Implement Multi-Threading for Faster Response Time:** Introduce multi-threading to the application to improve response times. By processing multiple tasks simultaneously, the system will be able to handle requests more efficiently, resulting in a smoother user experience.

REFERENCE

- Jason Brownlee, *How to Perform Face Detection with Deep Learning*, Available: `https://machinelearningmastery.com/how-to-perform-face-detection-with-classical-and-deep-learning-methods-in-python-with-keras/`
- Adrian Rosebrock, *Face Recognition with OpenCV, Python, and Deep Learning*, Available: `https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/`
- Adrian Rosebrock, *Face Recognition with OpenCV, Python, and Deep Learning*, Available: `https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/`
- Adrian Rosebrock, *Face Detection with OpenCV and Deep Learning*, Available: `https://www.pyimagesearch.com/2018/02/26/face-detection-with-opencv-and-deep-learning/`
- Viblo, *Nhn din khuôn mt vi mng MTCNN và Facenet - Phn 1*, Available: `https://viblo.asia/p/nhan-dien-khuon-mat-voi-mang-mtcnn-va-facenet-phan-1-Qbq5QDN4lD8`
- Viblo, *Nhn din khuôn mt vi mng MTCNN và Facenet - Phn 2*, Available: `https://viblo.asia/p/nhan-dien-khuon-mat-voi-mang-mtcnn-va-facenet-phan-2-bJzKmrVXZ9N`
- Pandaml, *Nhn dng khuôn mt*, Available: `https://www.pandaml.com/nhan-dang-khuon-mat/`
- Shen D, *Documentation: Project Face Recognition System with OpenCV*, Available: `https://medium.com/@shendyeff/documentation-project-face-recognition-system-with-opencv-2d18793623d6`
- DataCamp, *Face Detection in Python with OpenCV*, Available: `https://www.datacamp.com/tutorial/face-detection-python-opencv`
- Analytics Vidhya, *Face Recognition System Using Python*, Available: `https://www.analyticsvidhya.com/blog/2022/04/face-recog`

`nition-system-using-python/`

- VinBigData, *One-Shot Learning with Siamese Networks*, Available: `https://vinbigdata.com/kham-pha/tong-quan-ve-one-shot-learning-trong-thi-giac-may-tinh.html`
- Ravi Salakhutdinov, *One-Shot Learning*, Available: `https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf`
- Tien Su, *Face Recognition: A Practical Guide*, Available: `https://tiensu.github.io/blog/52_face_recognition/`
- Huy Tran Van, *Face Recognition with OpenCV, Python, and Deep Learning*, Available: `https://github.com/huytranvan2010/Face-Recognition-with-OpenCV-Python-DL`