FACIAL RECOGNITION WITH EMOTION AND LIVENESS

COS30082 Applied Machine Learning

MAY 26, 2025

HAI NAM NGO
Student ID 103488515

Contents

1. Methodology

The model training was done in two phases using separate Jupyter notebooks. The first notebook used a Kaggle competition kernel as a starting point for data preparation, followed by training two CNN models: one using metric learning and the other using classification. No hyperparameter tuning was applied at this stage. In the second notebook, the same setup was used, but with added hyperparameter tuning to improve model performance. The results from both phases were compared to evaluate the impact of tuning and identify the best-performing model.

Two approaches were explored for building the GUI. The first used built-in webcam and detection functions from the DeepFace library, while the second combined multiple components: a Facial Expression Recognition (FER), separately trained age and gender models, and an open-source silent anti-spoofing module from GitHub. Although the DeepFace approach was quicker to set up, it suffered from performance issues like lag and lower accuracy. The second approach, while more complex and time-consuming to implement, delivered better accuracy and smoother real-time performance, making it the preferred choice for the final system.

1.1. Data Preparation

To prepare the data for training and evaluation, I first defined key parameters, including the number of training epochs (25), batch size (128), number of labels to use (500), embedding dimension (128), and a triplet loss margin of 0.3. These parameters guided both the training loop and the structure of the model. However these are just initial set up, the number will be changed in further implementation, which will be discussed later.

All images were resized to 64×64 pixels to ensure uniform input dimensions and reduce computational load. I applied standard image preprocessing using PyTorch transforms: resizing, converting to tensor format, and normalizing pixel values using the ImageNet mean and standard deviation. This helped stabilize training and ensure the model learned more effectively.

For the dataset itself, I used a custom FaceDataset class to load images from the training, validation, and test folders. To manage dataset size and focus the model on a more controlled classification task, I limited the data to the first 500 unique identities (classes). After loading, I filtered each dataset (train, validation, and test) to include only images that belonged to these selected 500 classes.

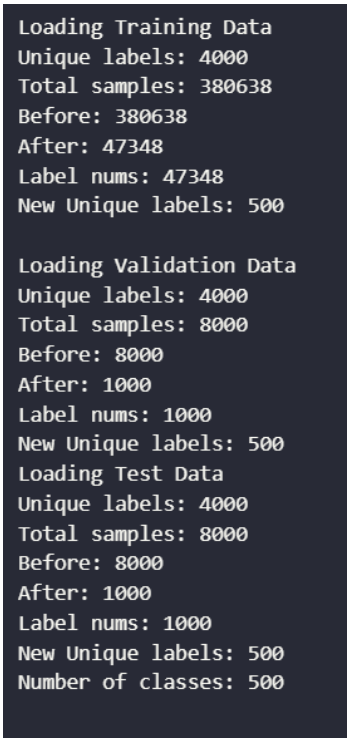

Figure 1 Sample and class after reducing the size of the dataset

This setup allowed for speeding up the training progress and still keeps the goal of knowing the algorithm of the implementation.

1.2. Architecture - Classification-based embedding model

This is the architecture of the classification-based embedding model that is used for both training notebook and used for load the trained model to the GUI.py

Component	Details
Input	Image tensor with 3 channels (RGB), size 64×64
Feature Extraction	3 convolutional blocks: each with 2 Conv layers (3×3 kernels), BatchNorm, ReLU, and MaxPooling
	- Block 1: 64 channels, MaxPool reduces size to 32×32
	- Block 2: 128 channels, MaxPool reduces size to 16×16
	- Block 3: 256 channels, MaxPool reduces size to 8×8
Flattening	Output features flattened (256 channels ×8 ×8 = 16,384 features)
Embedding Layer	Fully connected layers: 16,384 to 512 (with BatchNorm and ReLU), then 512 to 128 (embedding_dim)

Embedding Output	Normalized embedding vector of size 128 (for similarity comparison)
Classification Layer	Linear layer mapping embedding (128) to number of classes (num_classes)
Output	Returns classification logits and normalized embedding (or just embedding if requested)

The model is a CNN designed to classify faces and extract meaningful face embeddings. It has three convolutional blocks that progressively learn features while reducing the image size through pooling. After flattening, the features pass through fully connected layers to create a normalized embedding vector, which is used for both classification and face verification.

Below is the comparison of the parameter for classification models of two notebooks, the result will be compared in the Result section.

Parameter	Without Hyperparameter Tuning	With Hyperparameter Tuning
Learning Rate (lr)	0.01	0.05
Weight Decay	1e-4	0.001
Momentum	0.9	0.95
Step Size (LR Scheduler)	5 epochs	7 epochs
Gamma (LR Decay Rate)	0.1	0.1
Embedding Dimension	128	512
Batch Size	128	64
Validation Accuracy	N/A	0.0800
Epochs	20	25
Notes	Default hyperparameters, simpler setup	Best parameters found via tuning, improved accuracy but requires more effort

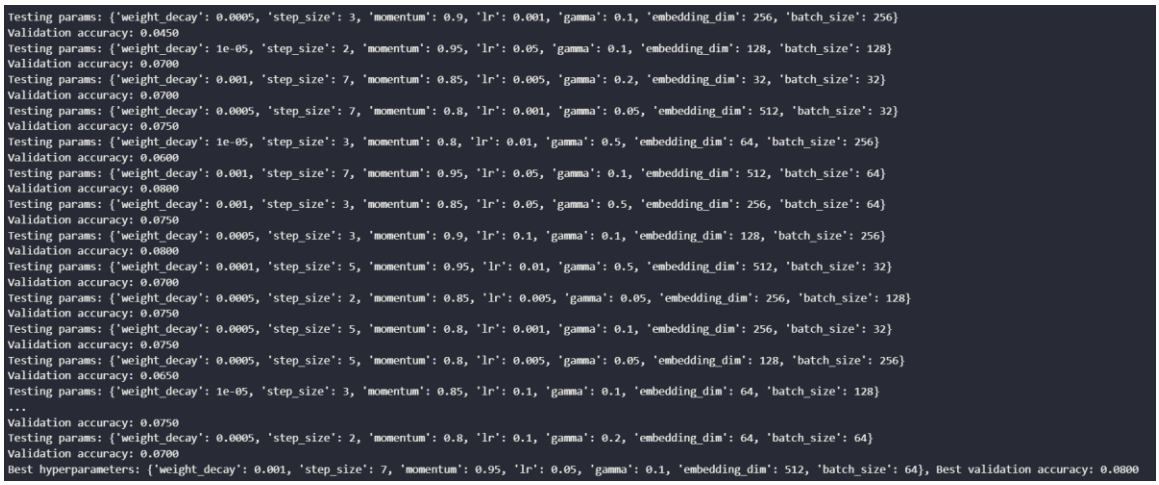

Figure 2 Proof of finding hyperparameter

1.3. Architecture - Metric-learning embedding model

This is the architecture of the metric-learning embedding model that is used for both training notebook and used for load the trained model to the GUI.py

Component	Description
Feature Extraction	Three convolutional blocks with Conv2d, BatchNorm, ReLU, and MaxPooling layers to extract image features
Flatten and embedding	Flatten features, then pass through two fully connected layers with BatchNorm and ReLU to create embeddings
Embedding Size	Outputs a fixed-length embedding vector (default 128 dimensions)
Normalization	Embeddings are L2-normalized to make similarity comparisons consistent
Loss Function	Uses triplet loss to train model to bring similar faces closer and dissimilar faces farther apart (margin = 1.0)

The model uses three convolutional blocks to extract features from input images. These features are flattened and passed through fully connected layers to produce a compact embedding vector. The embeddings are normalized and trained using triplet loss, which encourages the model to distinguish between similar and different faces.

Below is the comparison of the comparison of the parameter for triplet models of two notebooks, the result will be compared in the Result section.

Parameter	Old Triplet Parameters	New Tuned Triplet Parameters

Learning Rate (lr)	0.0001	5e-5
Weight Decay	1e-5	0.001
Margin	0.3	0.3
Number of Triplets	10,000	5,000
Batch Size	64	64 *(from classification best params)*
Embedding Dimension	128	512 *(from classification best params)*
Best Avg Training Loss	Not specified or higher	0.0351
Notes	Default setup	Tuned for lower loss and aligned with classification setup

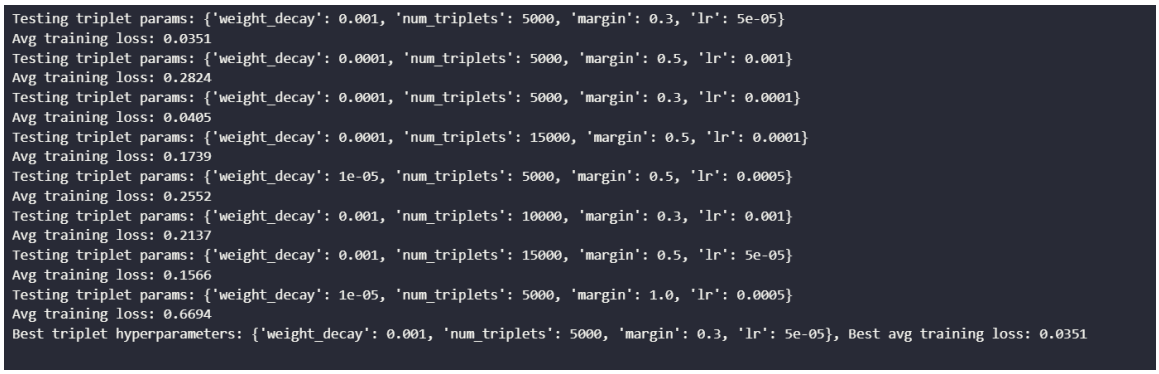

Figure 3 Proof of finding hyperparameter

2. Training Process

Overall, the classification-based model of both versions takes longer time compared to the metric learning model.

2.1. Training progress – Classification Based Model

2.1.1. Without hyperparameter

```
Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"c4ac88192c3a4e98b1113343fb108db0","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"be5192c29de84024ba2799d4582177bb","version_major":2,"version_minor":0}

Epoch 17/20 - Train Loss: 0.0226, Train Acc: 1.0000, Val Loss: 2.1016, Val Acc: 0.5460, Time: 49.50s

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"04816a083add426ca9e53988946de241","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"775967e3af6c4c26a4feb8aca6b0076b","version_major":2,"version_minor":0}

Epoch 18/20 - Train Loss: 0.0225, Train Acc: 1.0000, Val Loss: 2.1027, Val Acc: 0.5420, Time: 51.44s

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"5ecd380126744968bff2878e52d9e53a","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"9152887eda454cf28b95a81eae094b79","version_major":2,"version_minor":0}

Epoch 19/20 - Train Loss: 0.0225, Train Acc: 1.0000, Val Loss: 2.1066, Val Acc: 0.5520, Time: 48.94s

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"56ade903e0e04144b9a797a68e047f40","version_major":2,"version_minor":0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id":"31c155727d8342f4b173ab767ada4cfa","version_major":2,"version_minor":0}

Epoch 20/20 - Train Loss: 0.0225, Train Acc: 1.0000, Val Loss: 2.1034, Val Acc: 0.5440, Time: 47.79s
```

Figure 4 Result for each epoch (without hyperparameter)

2.1.2. With hyperparameter

```
Epoch 19/25 [Train]: 100% 740/740 [00:45<00:00, 16.45it/s, loss=0.0476, acc=1.0000]
Epoch 19/25 [Val]: 100% 16/16 [00:01<00:00, 15.49it/s, loss=2.0216, acc=0.5500]
 Epoch 19/25 - Train Loss: 0.0403, Train Acc: 1.0000, Val Loss: 1.7057, Val Acc: 0.6270, Time: 46.26s
Epoch 20/25 [Train]: 100% 740/740 [00:45<00:00, 16.52it/s, loss=0.0388, acc=1.0000]
Epoch 20/25 [Val]: 100% 16/16 [00:00<00:00, 21.19it/s, loss=2.0039, acc=0.5750]
 Epoch 20/25 - Train Loss: 0.0401, Train Acc: 1.0000, Val Loss: 1.7079, Val Acc: 0.6240, Time: 45.92s
Epoch 21/25 [Train]: 100% 740/740 [00:45<00:00, 16.51it/s, loss=0.0448, acc=1.0000]
Epoch 21/25 [Val]: 100% 16/16 [00:01<00:00, 17.87it/s, loss=2.0255, acc=0.5750]
 Epoch 21/25 - Train Loss: 0.0401, Train Acc: 1.0000, Val Loss: 1.7194, Val Acc: 0.6290, Time: 46.58s
Epoch 22/25 [Train]: 100% 740/740 [00:45<00:00, 16.51it/s, loss=0.0370, acc=1.0000]
Epoch 22/25 [Val]: 100% 16/16 [00:00<00:00, 22.51it/s, loss=2.0362, acc=0.5250]
 Epoch 22/25 - Train Loss: 0.0395, Train Acc: 1.0000, Val Loss: 1.7145, Val Acc: 0.6180, Time: 45.88s
Epoch 23/25 [Train]: 100% 740/740 [00:45<00:00, 16.49it/s, loss=0.0451, acc=1.0000]
Epoch 23/25 [Val]: 100% 16/16 [00:01<00:00, 17.92it/s, loss=2.0533, acc=0.5500]
 Epoch 23/25 - Train Loss: 0.0393, Train Acc: 1.0000, Val Loss: 1.7179, Val Acc: 0.6220, Time: 46.06s
Epoch 24/25 [Train]: 100% 740/740 [00:45<00:00, 16.49it/s, loss=0.0457, acc=1.0000]
Epoch 24/25 [Val]: 100% 16/16 [00:00<00:00, 22.48it/s, loss=2.0735, acc=0.5000]
 Epoch 24/25 - Train Loss: 0.0392, Train Acc: 1.0000, Val Loss: 1.7198, Val Acc: 0.6230, Time: 45.84s
Epoch 25/25 [Train]: 100% 740/740 [00:45<00:00, 16.52it/s, loss=0.0446, acc=1.0000]
Epoch 25/25 [Val]: 100% 16/16 [00:00<00:00, 18.24it/s, loss=2.0376, acc=0.5000]
 Epoch 25/25 - Train Loss: 0.0395, Train Acc: 1.0000, Val Loss: 1.7212, Val Acc: 0.6240, Time: 46.02s
```

Figure 5 Result for each epoch (with hyperparameter)

2.2. Training progress – Metric Learning Model

2.2.1. Without hyperparameter

```
Epoch 19/25 - Loss: 0.0027, Time: 14.34s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"9d006ed5b02845d1aea01fe6d08211f1"}

Epoch 20/25 - Loss: 0.0029, Time: 14.31s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"ba00a79249e44a849af29e7e361f5a14"}

Epoch 21/25 - Loss: 0.0030, Time: 14.27s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"4aa60c95763c42a787e28d0404c467cf"}

Epoch 22/25 - Loss: 0.0017, Time: 14.30s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"54a20fbef8154da69495dddb10b611cb"}

Epoch 23/25 - Loss: 0.0009, Time: 14.16s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"b2d553e67e114eaaa19e48757ff37211"}

Epoch 24/25 - Loss: 0.0029, Time: 14.38s

  Could not render content for 'application/vnd.jupyter.widget-view+json'
  {"version_major":2,"version_minor":0,"model_id":"5fc5185594e94793a84138c07c5a16d6"}

Epoch 25/25 - Loss: 0.0028, Time: 14.40s
```

Figure 6 Result for each epoch (without hyperparameter)

2.2.2. With hyperparameter

```
Epoch 11/25 - Loss: 0.0044, Time: 14.29s
Epoch 12/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.63it/s, loss=0.0195]
Epoch 12/25 - Loss: 0.0036, Time: 14.26s
Epoch 13/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.61it/s, loss=0.0276]
Epoch 13/25 - Loss: 0.0023, Time: 14.32s
Epoch 14/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0304]
Epoch 14/25 - Loss: 0.0026, Time: 14.25s
Epoch 15/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.63it/s, loss=0.0198]
Epoch 15/25 - Loss: 0.0016, Time: 14.25s
Epoch 16/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.61it/s, loss=0.0000]
Epoch 16/25 - Loss: 0.0018, Time: 14.32s
Epoch 17/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0214]
Epoch 17/25 - Loss: 0.0010, Time: 14.23s
Epoch 18/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0176]
Epoch 18/25 - Loss: 0.0020, Time: 14.34s
Epoch 19/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.64it/s, loss=0.0172]
Epoch 19/25 - Loss: 0.0027, Time: 14.34s
Epoch 20/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.61it/s, loss=0.0131]
Epoch 20/25 - Loss: 0.0029, Time: 14.31s
Epoch 21/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0147]
Epoch 21/25 - Loss: 0.0030, Time: 14.27s
Epoch 22/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.64it/s, loss=0.0000]
Epoch 22/25 - Loss: 0.0017, Time: 14.30s
Epoch 23/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0378]
Epoch 23/25 - Loss: 0.0009, Time: 14.16s
Epoch 24/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.59it/s, loss=0.0366]
Epoch 24/25 - Loss: 0.0029, Time: 14.38s
Epoch 25/25 [Triplet]: 100% 79/79 [00:14<00:00,  5.62it/s, loss=0.0005]
```

Figure 7 Result for each epoch (with hyperparameter)

3. Training Result

3.1. Comparison between two versions of Classification-Based model

The model was trained for 10 epochs using the Adam optimizer with a learning rate of 1e-4. The cross-entropy loss function was employed to evaluate performance at each step.

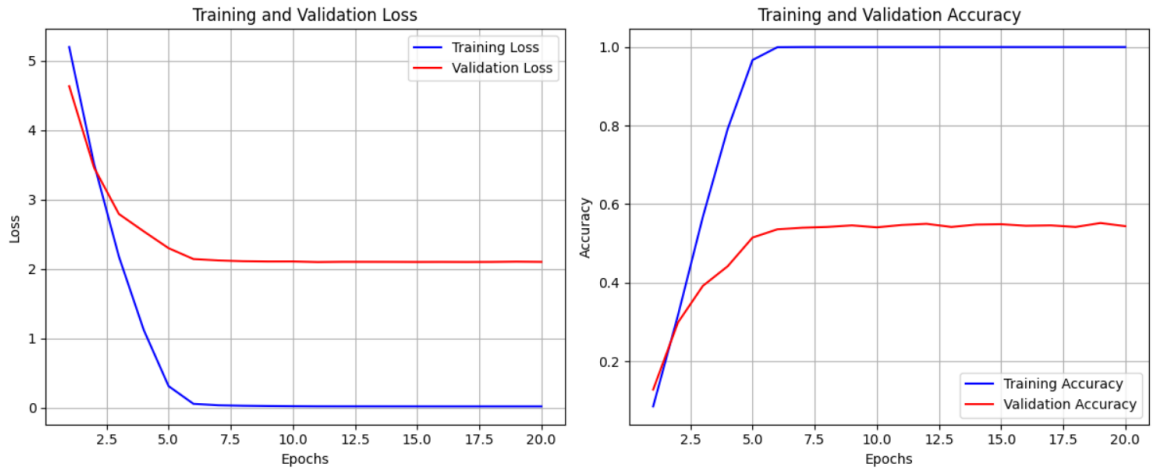

Figure 8 Training/validation Loss & Acc (without hyperparameter tuning)

The training and validation performance without hyperparameter tuning appears stable, with no major fluctuations. From the 6th epoch onward, the training loss levels off around 2.1, while the validation loss remains steady at approximately 0.02. Although the training accuracy stays consistently high (close to 1.0), the validation accuracy only reaches about 0.58, staying below 0.6.

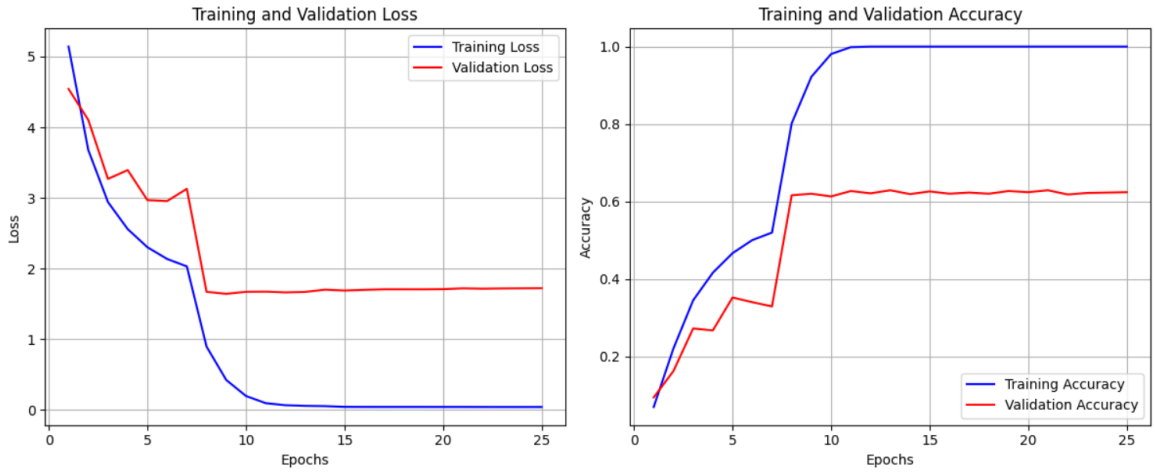

Figure 9 Training/validation Loss & Acc (with hyperparameter tuning)

On the other hand, with hyperparameter tuning, the training and validation performance shows more fluctuation during the first 10 epochs. However, from the 10th epoch onward, noticeable improvements are observed compared to the untuned version. The validation

loss drops below 2, settling around 1.8, and the validation accuracy increases, surpassing 0.6.

3.2. Comparison between two versions of metric learning model

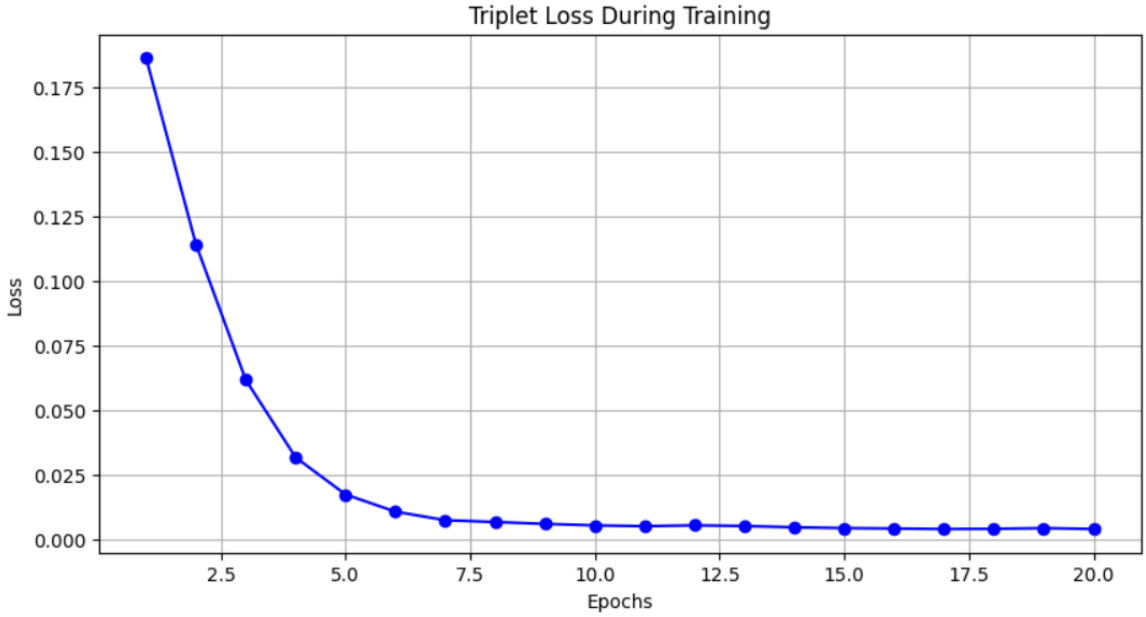

Figure 10 Triplet Loss during training (without hyperparameter tuning)

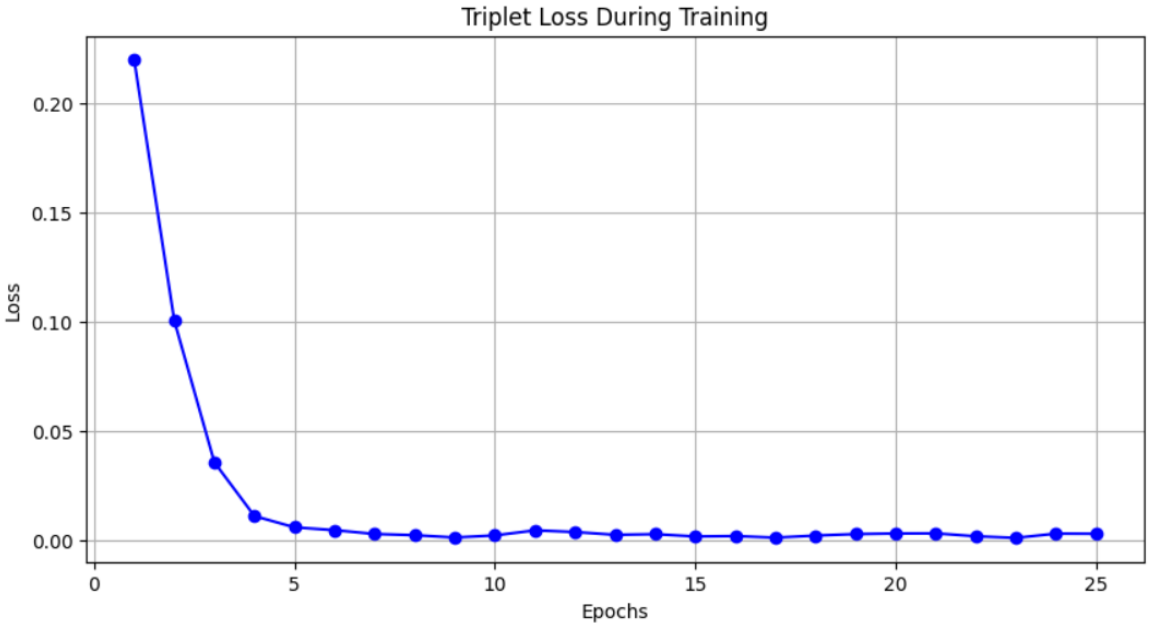

Figure 11 Triplet Loss during training (with hyperparameter tuning)

Overall, there is not much difference between the two versions, the triplet loss starts to go down to 0 for both versions at 5th epoch. To determine which model is better, please go to Testing Result section.

4. Testing Result

For further information, these terms will be used for this section:

Metric	Description	Value	What it Shows
Cosine Similarity	Measures how similar two vectors are by their direction	**Higher better** (closer to 1)	Higher value means vectors point in the same direction means more similar
Euclidean Distance	Measures the straight-line distance between two points	**Lower better**	Lower value means points are closer means more similar
AUC	Summary score of ROC curve performance	**Higher better** (max 1.0)	Higher value means better overall classification ability

4.1. Comparison between two versions of Classification-Based model

After tuning the hyperparameters, the model shows a clear improvement. The test loss decreased by about **0.36**, indicating better generalization, and the test accuracy increased by nearly **8%**, showing more reliable performance on unseen data.

Figure 12 Test Accuracy for classification model (without hyperparameter tuning)

Figure 13 Test Accuracy for classification model (with hyperparameter tuning)

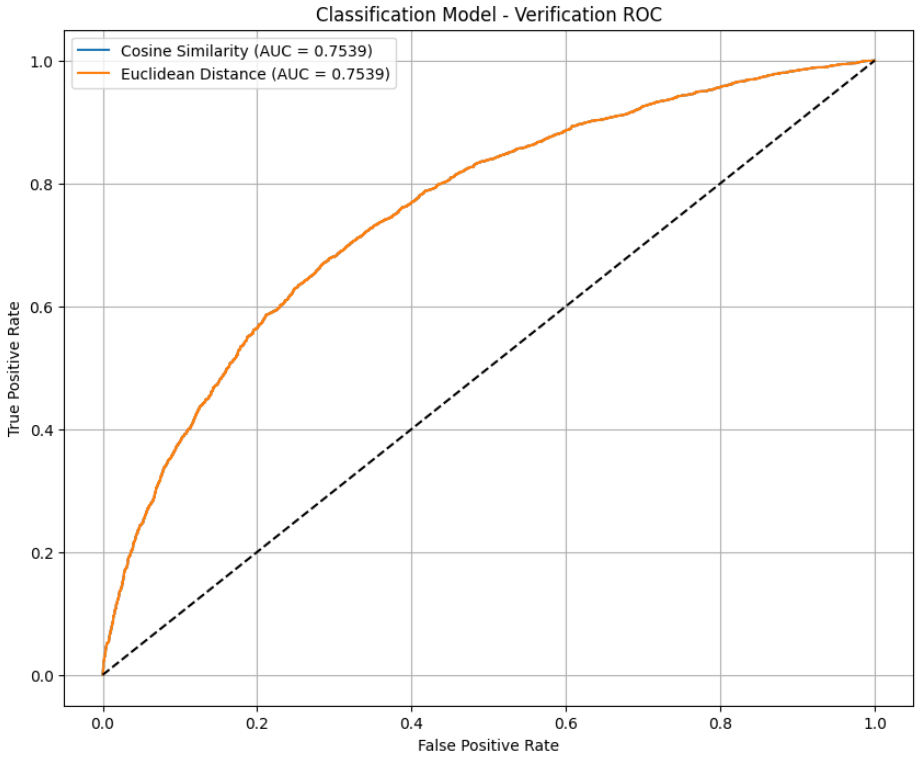

Figure 14 Curve without hyperparameter tuning

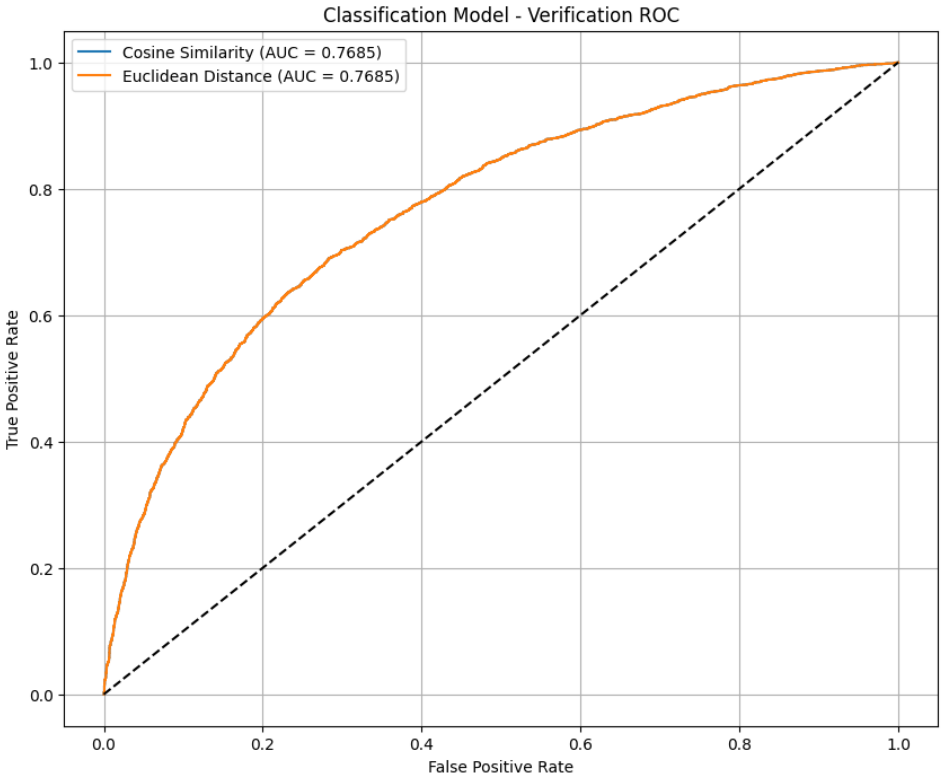

Figure 15 Curve with hyperparameter tuning

The first ROC curve, which shows the model performance without hyperparameter tuning, has an AUC value of 0.7539 for both cosine similarity and Euclidean distance. After tuning, the second ROC curve shows a slightly better AUC of 0.7685 for both metrics. This means the tuned model is better at distinguishing between matching and non-matching face pairs.

Both ROC curves are well above the diagonal line that represents random guessing, which indicates that both models perform much better than chance. The curve from the tuned model is closer to the top-left corner, which reflects its improved ability to correctly verify faces.

When comparing the two metrics, cosine similarity and Euclidean distance, they produce nearly identical ROC curves and AUC values in both cases. This suggests that either metric works equally well for face verification with these models.

Overall, the tuned model performs better and is more reliable for face verification. Although the improvement is not large, it clearly shows that adjusting model training and hyperparameters can lead to better verification results.

4.1. Comparison between two versions of Classification-Based model

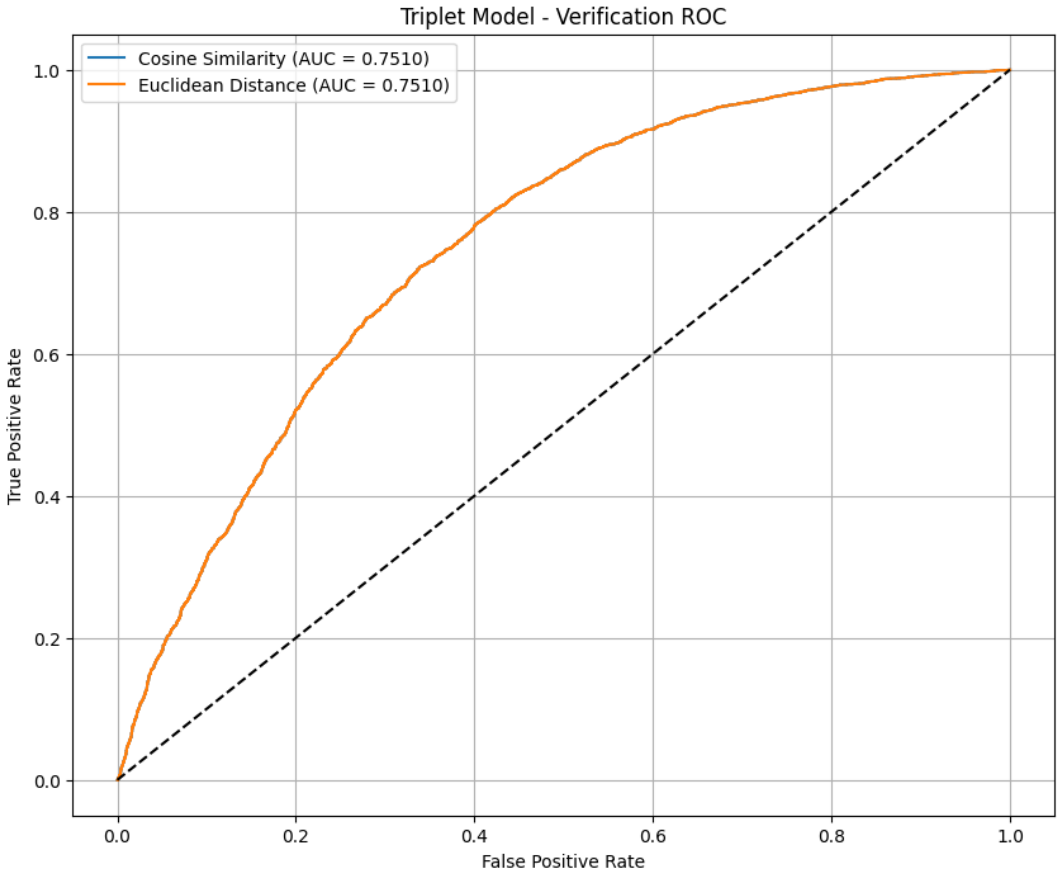

Figure 16 Curve with hyperparameter tuning (Triplet)

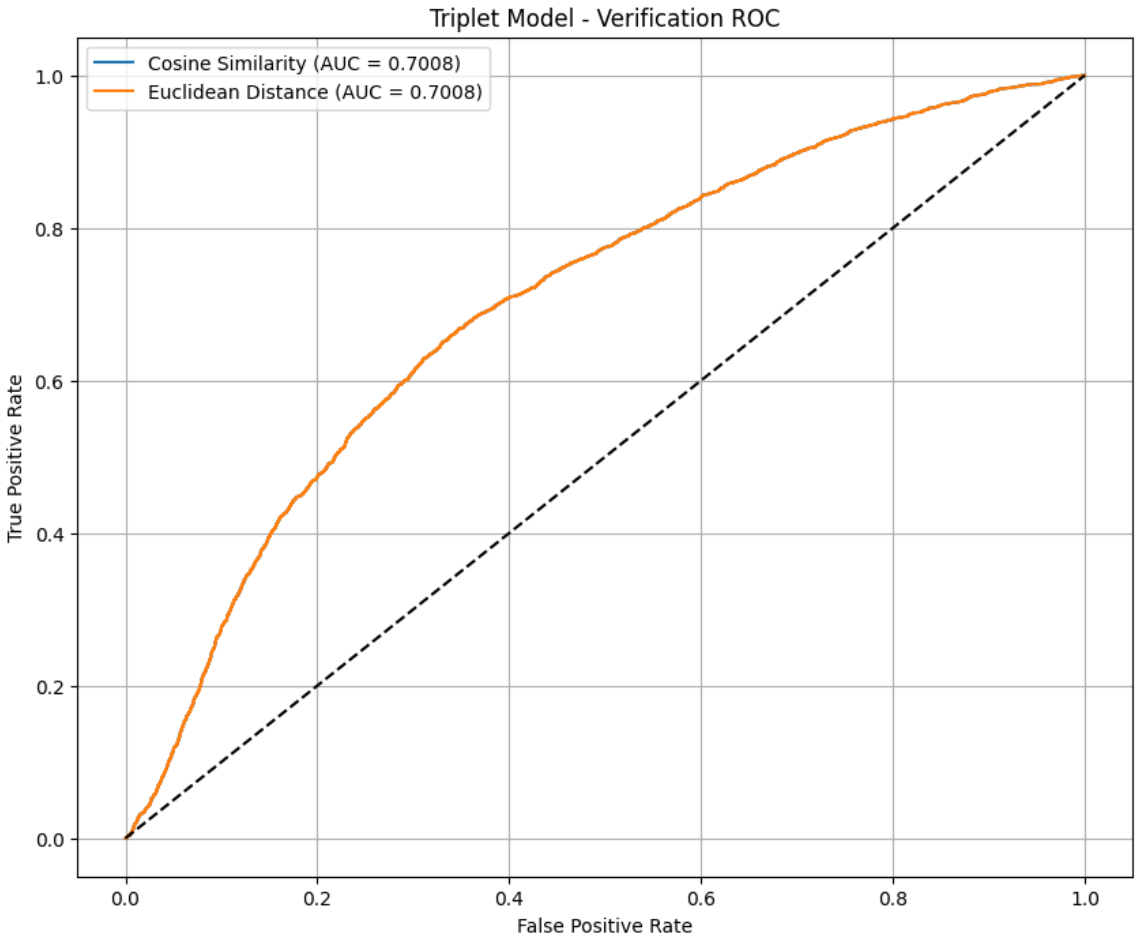

Figure 17 Curve with hyperparameter tuning (Triplet)

The triplet model without hyperparameter tuning performed better, achieving an AUC of 0.7510. This means it was better at telling apart images of the same person from different people. Its ROC curve was closer to the top-left corner, showing it had a good balance of true positives and false positives. Surprisingly, the model with hyperparameter tuning scored a bit lower, with an AUC of 0.7008. While it still did better than randomly guessing, its performance was more moderate, and the ROC curve wasn't as strong. This suggests that the hyperparameter choices made during tuning might not have been optimal and could have even hurt the model's ability to verify faces accurately.

Overall, tuning doesn't always guarantee better results and sometimes needs careful adjustment.

4.3 Summary of the testing phase

```
Triplet Model Results:
Cosine Similarity AUC: 0.7510
Euclidean Distance AUC: 0.7510
Best Accuracy: 0.6912 (threshold: 0.1587)

=== Verification Performance Comparison ===
Classification Model - Cosine AUC: 0.7539, Euclidean AUC: 0.7539, Best Accuracy: 0.6927
Triplet Model - Cosine AUC: 0.7510, Euclidean AUC: 0.7510, Best Accuracy: 0.6912

=== Final Conclusion ===
The Classification model performed better for verification using cosine similarity.
The Classification model performed better for verification using Euclidean distance.

=== Generalization Analysis ===
Train-Val Accuracy Gap: 0.4560
Val-Test Accuracy Gap: -0.0130
The classification model shows signs of overfitting between training and validation.
```

Figure 18 Recap for the first notebook (without hyperparameter tuning)

```
Triplet Model Results:
Cosine Similarity AUC: 0.7008
Euclidean Distance AUC: 0.7008
Best Accuracy: 0.6602 (threshold: 0.0938)

=== Verification Performance Comparison ===
Classification Model - Cosine AUC: 0.7685, Euclidean AUC: 0.7685, Best Accuracy: 0.7035
Triplet Model - Cosine AUC: 0.7008, Euclidean AUC: 0.7008, Best Accuracy: 0.6602

=== Final Conclusion ===
The Classification model performed better for verification using cosine similarity.
The Classification model performed better for verification using Euclidean distance.

=== Generalization Analysis ===
Train-Val Accuracy Gap: 0.3760
Val-Test Accuracy Gap: -0.0120
The classification model shows signs of overfitting between training and validation.
```

Figure 19 Recap for the first notebook (with hyperparameter tuning)

In conclusion, the two models will be used for actual implementation for the attendance system will be:

- Classification-based embedding model **(with hyperparameter tuning)**
- Metric learning embedding model **(without hyperparameter tuning)**

5. Face recognition attendance system

For this attendance system, the features are:

1. Anti-spoofing
2. Age and Gender Detection
3. Emotion Detection
4. Face Verification
5. Create new employee
6. Automatically mark attendance
7. Switching between two trained models

5.1. Age and Gender Detection

The AgeGenderDetector module is responsible for identifying a person's age group and gender from a face image. It uses OpenCV's DNN (Deep Neural Network) module to load two pre-trained Caffe models, one for age estimation and one for gender classification. These models are loaded from the following files:

- Age model: age_net.caffemodel and age_deploy.prototxt
- Gender model: gender_net.caffemodel and gender_deploy.prototxt

These models are downloaded from a Youtuber's video tutorial, he saved the model in Google Drive for everyone to download and use them.

https://drive.google.com/drive/folders/1rM7R7nD1FdTGPIMKnbkulBPof2j7Y59A

When a face image is passed to the detect() method, the image is first preprocessed and converted into a blob (a suitable format for the models). This blob is then fed into the gender model to predict the gender ("Male" or "Female") and into the age model to predict the age group ("(25-32)"). The method returns both predictions as strings.

5.2. Emotion Detection

The EmotionDetector module is designed to recognize facial emotions. It uses the FER (Facial Expression Recognition) library, which includes built-in face detection using MTCNN. The detect() method first converts the input face image from BGR to RGB (since FER expects RGB format), then analyses it to identify emotional expressions.

The method returns the emotion with the highest confidence score ("Happy", "Sad", "Angry"). If no emotion is clearly detected, it returns a default value such as "Neutral".

5.3. Anti-Spoofing Detection

The SpoofDetector module performs anti-spoofing to ensure the detected face is real and not a printed photo or video replay. It uses a pre-trained deep learning model from the Silent Face Anti-Spoofing project.

The is_real() method takes an image and a bounding box (the region of the face), extracts the face area, resizes it appropriately, and feeds it into the anti-spoofing model. The model returns a prediction indicating whether the face is live (real) or spoofed (fake).

Overall, the code used:

- Age and Gender Detection: Done using OpenCV DNN with Caffe-based models.
- Emotion Detection: Uses the FER library to detect facial emotions from images.
- Anti-Spoofing Detection: Uses a Silent Face Anti-Spoofing model to distinguish real faces from spoofed ones.

Initially, Deepface library was also considered and implemented, but it was not chosen since the webcam after integrating Deepface is slow and laggy.

5.4 User Scenarios

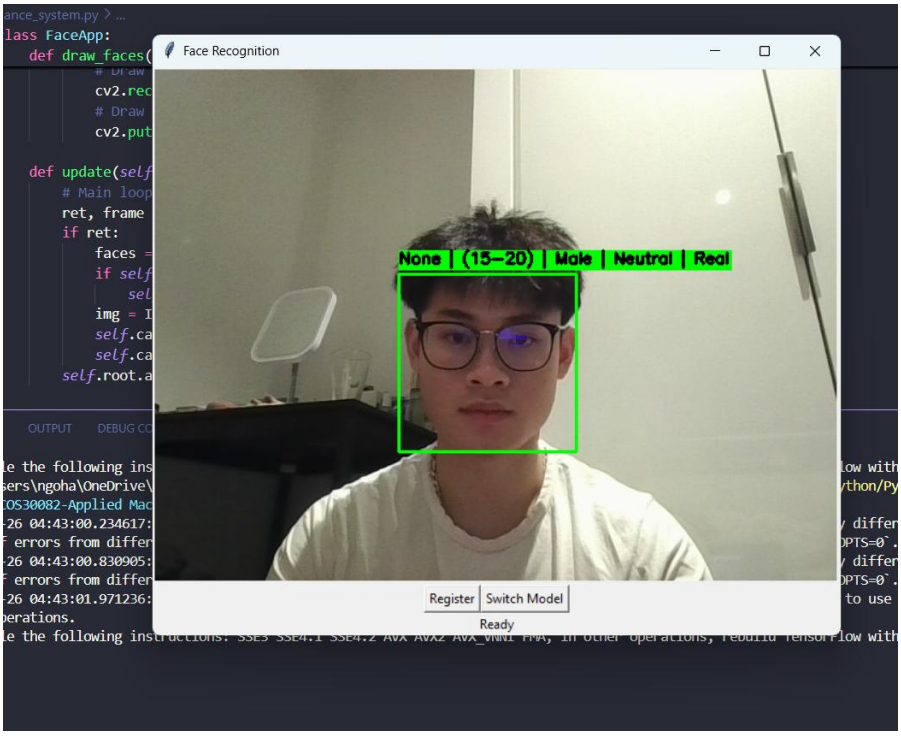

Figure 20 Case 1: New person coming to the system

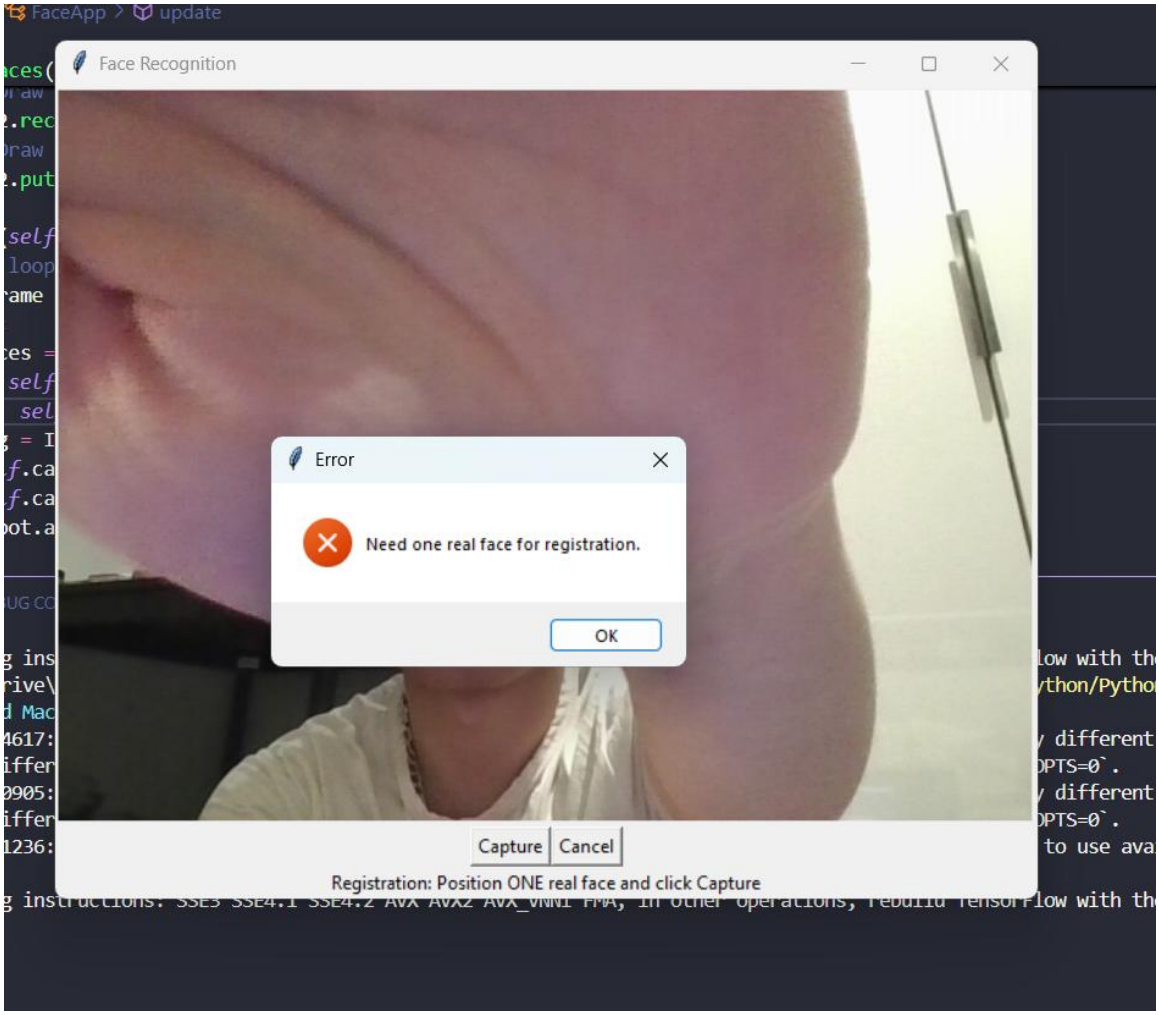

Figure 20 Case 2: Registration – when no face detected

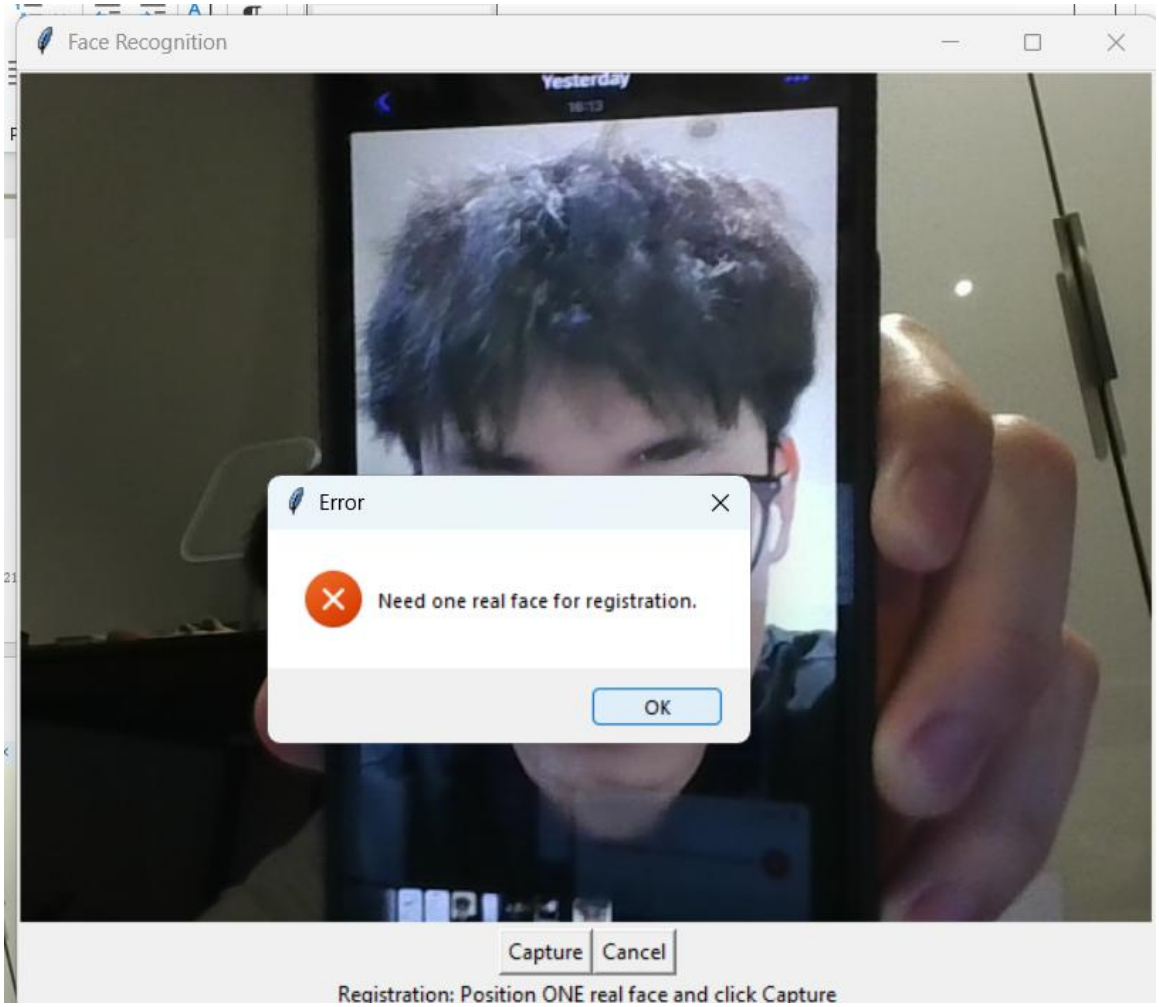

Figure 21 Case 3: Registration – when fake face detected

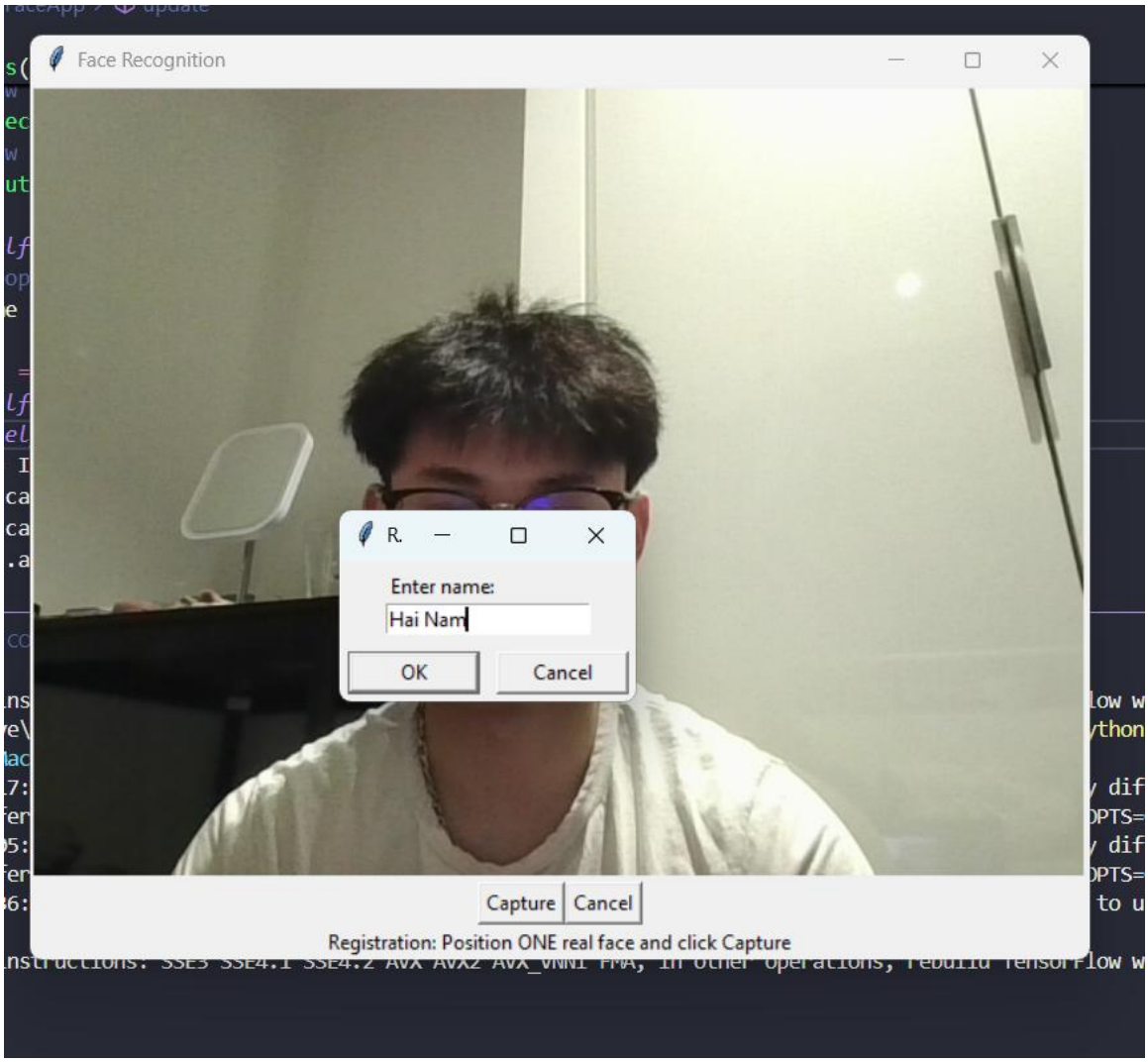

Figure 22 Case 4: Registration – when real face detected

When real face detected, the system will allow you to enter the name of the new employee.

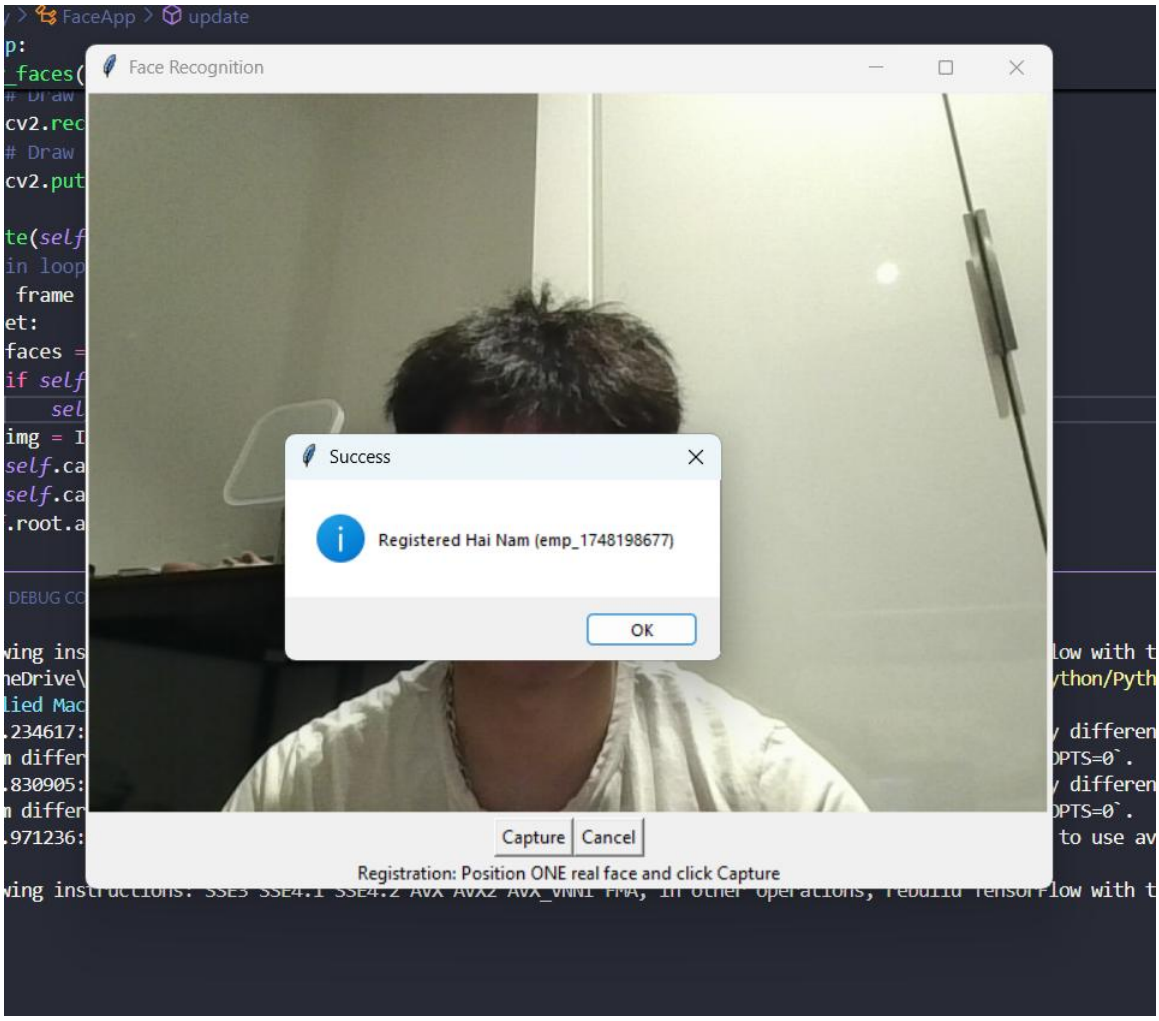

Figure 23 Case 5: Registration successfully

The system will produce a random employee ID, and notify the registration is successful.

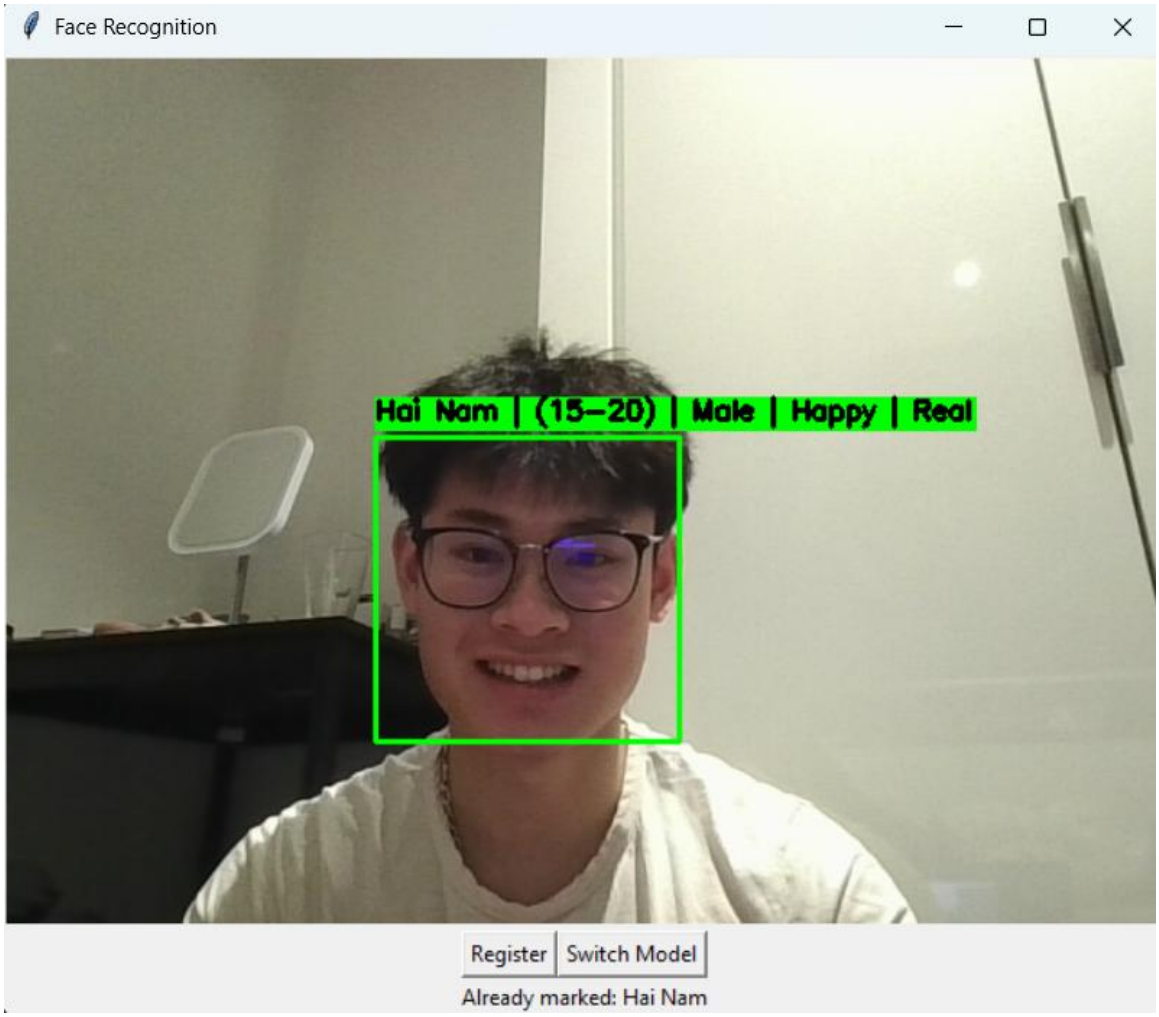

Figure 24 Case 6: Current employee coming to the system

If the database has the info of the person, it will automatically mark attendance and display the necessary info on top of the bounding box.

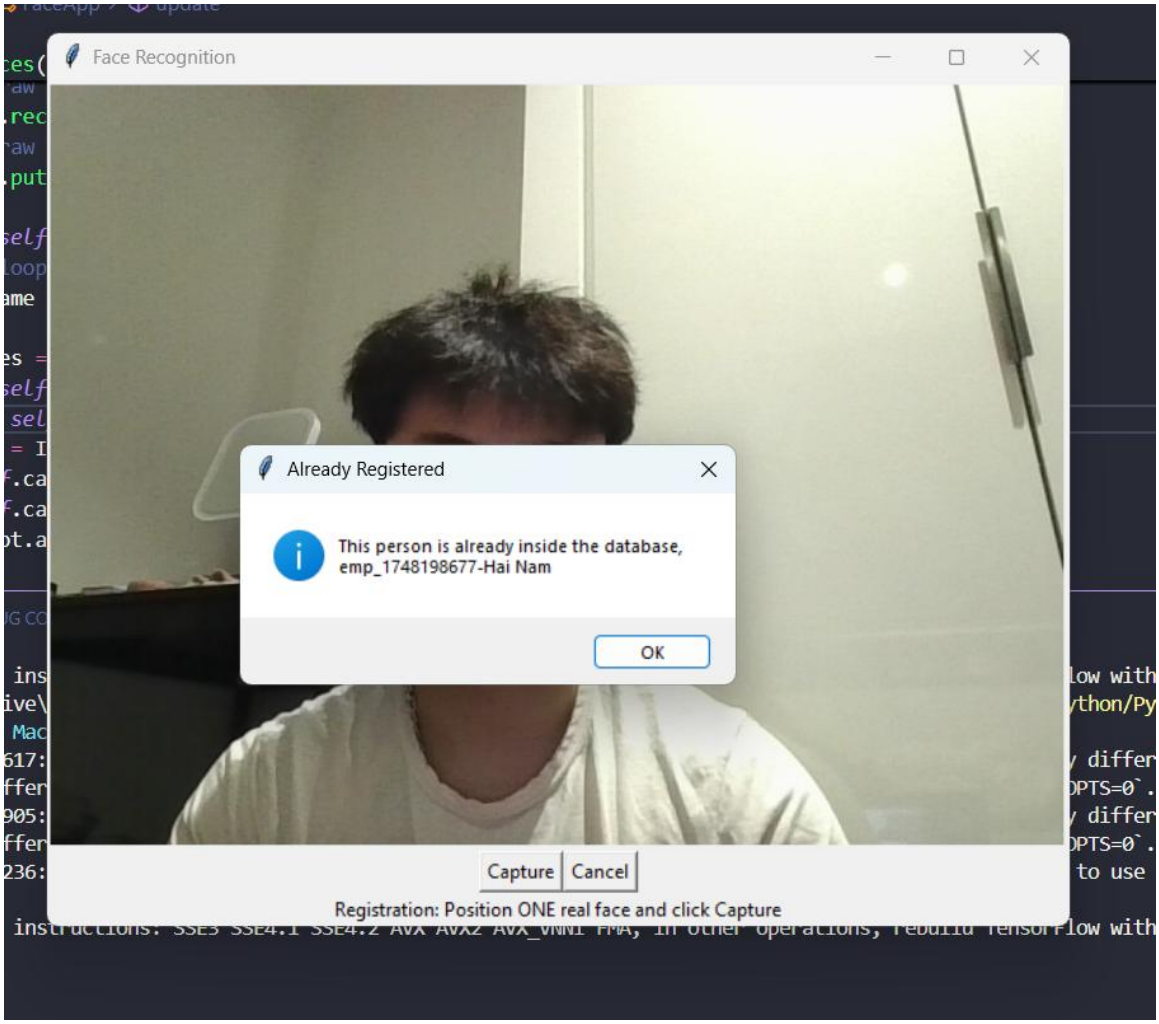

Figure 25 Case 7: Current employee makes new registration

If the current employee goes to the system and registers again, the system will not allow it.

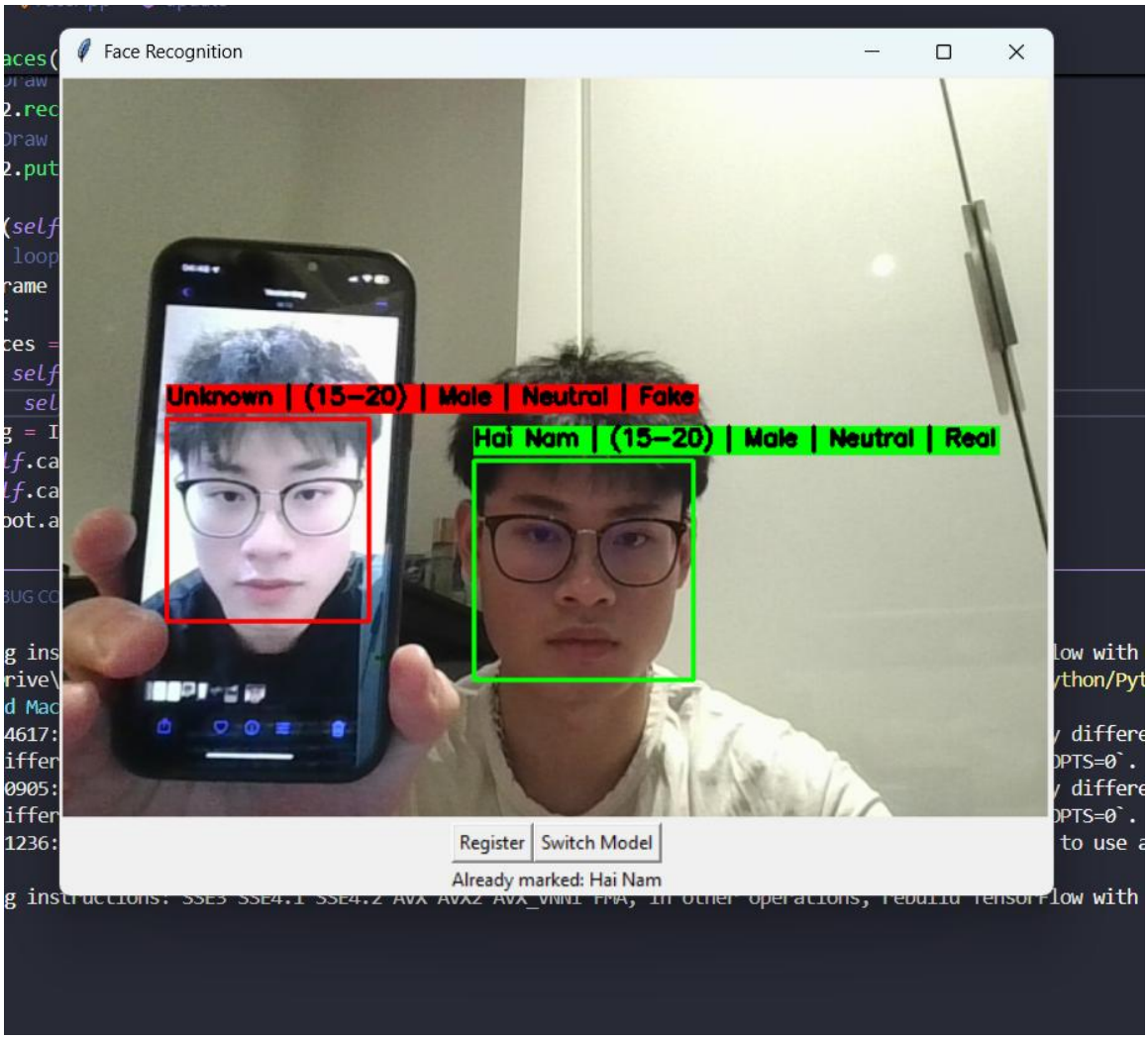

Figure 26 Case 8: Anti Spoofing

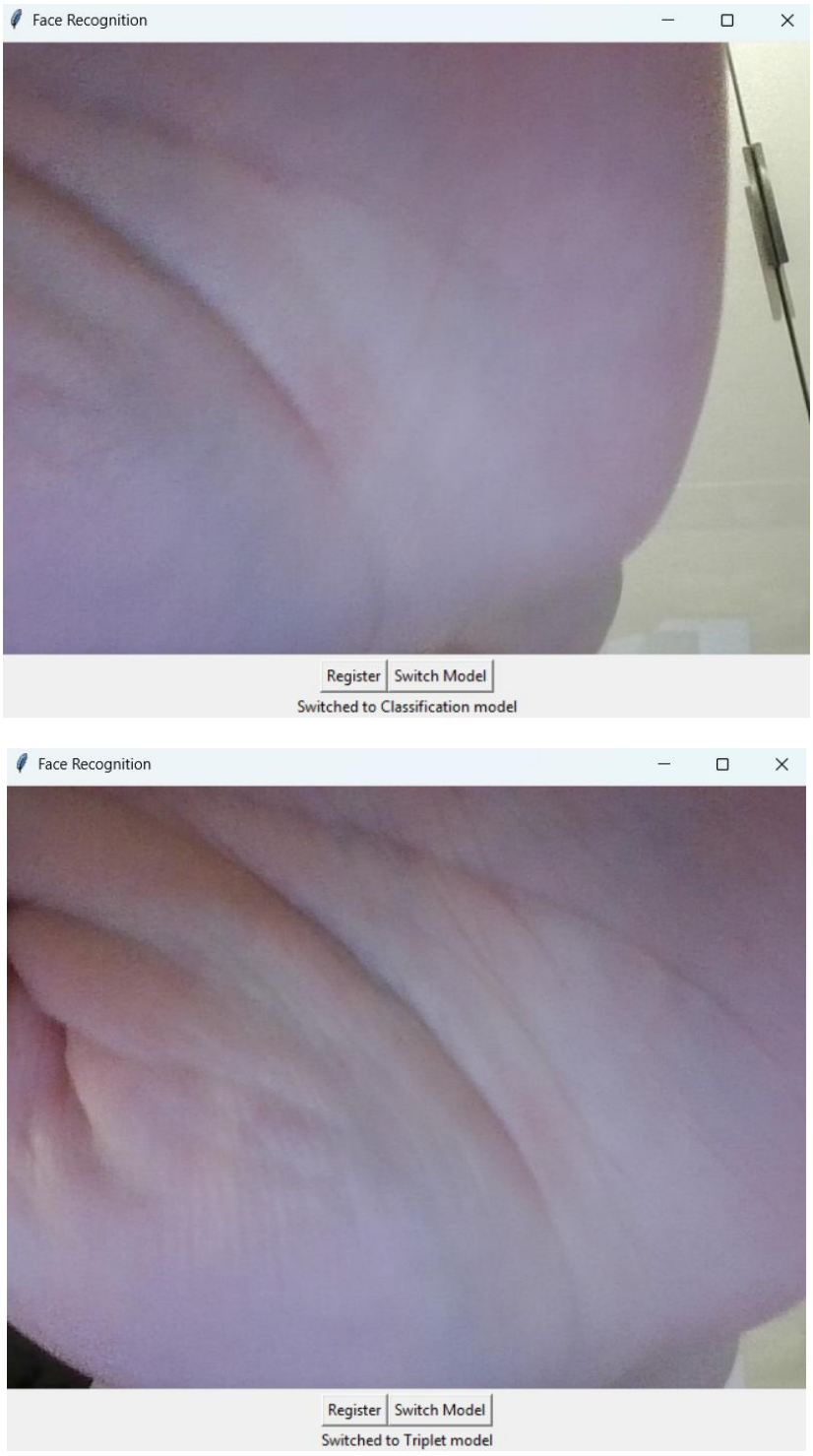

Figure 27 Case 9: Switching model feature