Sapienza University Of Rome

Facial Biometric System for Verification and Identification in Library User Authentication Protocols

Biometric Systems Project

Guillermo Bajo Laborda - 2182363
Antonina Wolszczak - 2184342

Prof. Maria De Marsico

January 2025

1 Introduction

Biometric systems are increasingly integral to modern security, offering reliable user identification and authentication. Facial recognition, favored for its ease, speed, and non-invasiveness, is widely used in applications from smartphone unlocking to secure access control. This project designs a facial recognition system using deep neural networks (DNNs) for identification and verification, emphasizing its use in a library setting to streamline access, enhance security, and improve user experience.

1.1 The Idea

The initial idea for our project came up in the library - most of the libraries at Sapienza University, where we had the opportunity to study, did not have any identification system for the people entering. This is problematic because anyone can enter, even people not affiliated with the university. Of course, there are gates at the entrances, but they never work and are always open.

Additionally, we noticed a problem with identifying students entering exam rooms. Often, professors check each student's ID one by one. It's easy to imagine how long this process can take with a hundred people.

For these reasons, we decided to create a simplified facial recognition model for entrances to libraries or classrooms to speed up this process. Of course, this is just an outline of a potential system that could function in reality.

1.2 Project Objectives

The primary goal of this project is to design and implement a facial recognition system that can accurately identify and verify library users, simplifying the access process to services. To achieve this we developed a facial recognition model using deep neural networks (DNNs), specifically Convolutional Neural Networks (CNNs), to perform both identification (recognizing who the person is) and verification (confirming if a person is who they claim to be) based on facial features.

This project aims to demonstrate the effectiveness of facial recognition as a tool for improving user access management and simplifying everyday tasks within a library setting. Additionally, performance evaluation will provide valuable insights into the strengths and limitations of facial recognition technology, particularly in smaller-scale applications.

1.3 Scope of the Project

The initial idea for our project came up in the library - most of the libraries at Sapienza University, where we had the opportunity to study, did not have any identification system for the people entering. This is problematic because anyone can enter, even people not affiliated with the university. Of course, there are gates at the entrances, but they never work and are always open.

Additionally, we noticed a problem with identifying students entering exam rooms. Often, professors check each student's ID one by one. It's easy to imagine how long this process can take with a hundred people.

For these reasons, we decided to create a simplified facial recognition model for entrances to libraries or classrooms to speed up this process. Of course, this is just an outline of a potential system that could function in reality. Advanced features like real-time recognition via camera feeds or the integration of large-scale databases are beyond the project's scope, but the prototype will serve as a basis for exploring potential future improvements and applications.

By adopting this approach, the project aims to offer a practical demonstration of how facial recognition can provide access management while laying the foundation for potential future implementations in various environments.

2 Data

In this chapter, we will present all the operations that were performed on the dataset and on the images before they were used as input data. We will show different approaches based on the organization of the dataset and the data augmentation techniques, implementing an analysis that presents both the advantages and disadvantages of these methods.

2.1. Dataset

The dataset for our project was initially sourced from the LFW (Labeled Faces in the Wild) dataset, which organizes images of faces by individual across separate folders. While LFW provided a useful starting point, the number of images per person was insufficient for training a reliable model, with some individuals having not more than one sample in the gallery. To address this, we performed data augmentation on the remaining images, which averaged around 50 images per person. We then applied data augmentation techniques - such as random rotations, flipping, and shifting

to artificially increase the dataset's variability, ensuring the model encountered diverse facial variations, including different poses and lighting conditions. We also balanced the number of images per person to prevent bias and ensure that no person dominated the dataset. Despite these efforts, further evaluation revealed that LFW lacked the diversity and image quality necessary for our project.

As a result, we switched to the CelebA dataset, which offers a significantly larger number of images per person, providing better coverage and variety for facial recognition tasks. CelebA consists of 202,599 labeled images of 10,177 individuals, offering a more robust and diverse dataset for training. While other datasets like Yale and ATT were considered, they were either in black and white or contained limited data, making them unsuitable for our needs.

As part of the dataset preparation, we also evaluated the quality of the images by using a face detection model based on OpenCV's Haar Cascade. This model allowed us to discard images where the face was poorly detected or obscured (e.g., due to heavy occlusions, poor lighting, or non-frontal poses). These images, which were 5,609 in total, were eliminated from the dataset, as they were considered unreliable for training a robust facial recognition model. This decision was made to ensure the model's performance would not be negatively affected by low-quality or irrelevant data, which is particularly important in applications like a facial recognition system for library identification, where accuracy is critical.

After evaluating CelebA, we found it provided a far more balanced and high-quality dataset, offering more training samples for each individual. The dataset's larger, more consistent image set, combined with its improved image quality, made it a much better fit for our model's needs. Consequently, we transitioned to CelebA, as its natural diversity and consistency eliminated the need for additional augmentation or balancing.

Feature	**CelebA Dataset**	**LFW Dataset**
Nº images	202,599	13,233
Nº people	10,177	5,749
Average Images PP	19.91	2.30
Data Augmentation	A priori not necessary due to larger, diverse	Advisable due to small dataset size
Labels/Attributes	40 attribute labels (e.g., glasses, gender, smile)	Only person identity

Figure 1: Comparison of CelebA and LFW datasets.

Once we switched to CelebA, we proceeded with the following preprocessing steps to prepare the dataset for training:

- **Resizing**: The original CelebA images varied significantly in resolution. To standardize the input for facial recognition, all images were resized to 160x160 pixels, a size that provides a balance between preserving key facial features and reducing computational costs.

- **Normalization**: To enhance model performance and accelerate convergence, pixel values were normalized to the range [0, 1] by dividing by 255. This step ensures more stable training and optimizes the model's ability to learn.

- **Data Augmentation**: Although data augmentation techniques such as random rotations, flips, and shifts were initially considered to increase dataset variability, it was determined that this step was unnecessary. The CelebA dataset, with its extensive sample size, naturally provided ample variation in pose, lighting, and expressions, making additional augmentation redundant.

The final dataset consisted of 203,213 images, which were split according to CelebA's guidelines: 162,770 images for training, 19,867 for validation, and 19,962 for testing. The dataset preparation process, including resizing and normalization, ensured that the images were optimized for training the facial recognition model. The plot below illustrates how the data looks like, showing a part of the dataset.

Figure 2: Part of the CalebA Dataset

2.2. Face Detection: Method Choice and Justification

In the context of this project, the initial step towards building an effective facial recognition system is face detection. We explored several options for detecting faces in our dataset, eventually selecting Dlib's frontal face detector due to its robust performance and balance between accuracy and processing speed.

We first experimented with OpenCV's Haar Cascade classifier, a classical method widely used in face detection tasks. While this method worked adequately on smaller datasets, we quickly noticed its limitations in terms of both speed and accuracy when applied to a large dataset like ours. The Haar Cascade classifier took approximately 2 hours to process the 202,599 images in our dataset, which was significantly slower than expected, making it unsuitable for real-time applications or projects requiring faster results.

Upon further investigation, we decided to test Dlib's frontal face detector, which uses Histogram of Oriented Gradients (HOG) features combined with a linear classifier and a sliding window detection approach. Dlib's face detector achieved to detect faces with reasonable precision, even under varied conditions like different lighting or partial occlusion, while processing the 202,599 images in approximately 2 hours. This time reduction compared to both Haar Cascade and MTCNN made Dlib the most viable option for our task.

Dlib's face detection algorithm also includes the detection of facial landmarks, which are key for understanding the position and orientation of the face. Once a face is detected in an image using the Histogram of Oriented Gradients (HOG) and Support Vector Machine (SVM) classifier, Dlib further refines the process by identifying 68 specific points on the face, known as landmarks. These landmarks correspond to important facial features such as the eyes, nose, mouth, and jawline.

The algorithm works by mapping these key points onto the detected face, which helps in several applications like facial alignment, expression recognition, or even 3D pose estimation. The landmarks are essential for aligning the face in a standardized way, ensuring that the model processes facial features accurately, regardless of the pose or orientation of the face in the image.

By using these facial landmarks, Dlib provides a more robust and detailed detection of facial structures, allowing for better recognition, even in cases of slight variations in angle, scale, or lighting. These landmark points are

also helpful for tasks like facial expression analysis or to prepare the image for further facial recognition tasks.

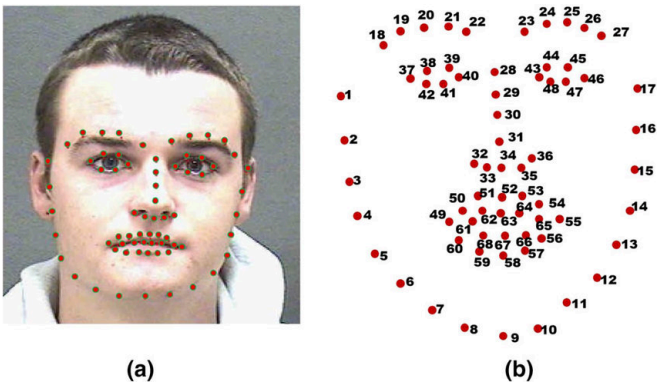

Figure 4: (A) Original face detection, (B) Facial landmarks identified by dLib

To improve reliability, we apply a dual filtering process. First, we use dlib for initial face detection, and then pass the results through MTCNN (Multi-task Cascaded Convolutional Networks). MTCNN further refines face detection by not only detecting faces but also localizing key facial landmarks, enhancing accuracy. This two-step approach helps eliminate false positives and ensures higher precision in detecting and recognizing faces.

In conclusion, after testing and evaluating multiple algorithms, we decided to proceed with Dlib's and MTCNN face detectors due to its efficient performance and robustness.

3 Methodology

Facial recognition has seen rapid advancements in recent years, with deep learning, especially Convolutional Neural Networks (CNNs), becoming the backbone of most modern systems. This chapter explores key techniques and architectures in facial recognition, such as CNNs, Fine Tunning, and models like Inception ResNet V1, providing the necessary context for the methods chosen in this project. The goal is to understand how these techniques can be applied to create an efficient and accurate facial recognition system for user identification and access control in a library environment.

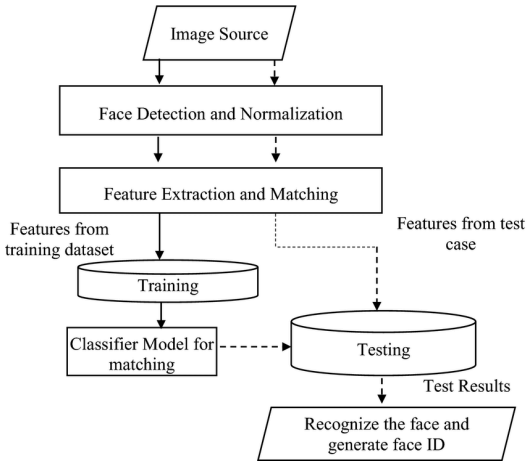

Figure 3: Flowchart of the facial recognition process: From image capture to user verification.

3.1 Convolutional Neural Networks (CNNs)

CNNs are a crucial technology for facial recognition, enabling systems to automatically learn complex facial features directly from raw image data. This ability to learn hierarchical representations of faces—starting from simple patterns like edges and gradually capturing higher-level features such as facial structures—is central to building robust face recognition systems.

In the context of this project, CNNs are particularly well-suited for recognizing and verifying users in a library setting, where the system must accurately identify individuals from a controlled, smaller dataset. ResNet and EfficientNet, two of the most advanced CNN architectures, have revolutionized facial recognition by providing models that are both highly accurate and computationally efficient. ResNet's use of residual connections allows deeper networks, which is beneficial for learning complex facial features. Meanwhile, EfficientNet optimizes network architecture to balance

depth, width, and resolution, making it a strong candidate for real-time applications with limited computational resources.

These advancements enable the facial recognition system in this project to handle variations in lighting, angle, and facial expressions, which are common in real-world applications like library access. By using CNNs, the model can learn detailed facial embeddings that allow the system to distinguish between registered users with high accuracy, even in non-ideal conditions.

3.2 Explanation of the Chosen Convolutional Neural Network (CNN) Architecture

We leveraged the Inception ResNet V1 model for facial recognition, available through the facenet_pytorch library. This architecture combines the benefits of Inception and ResNet, making it well-suited for tasks that require learning rich, discriminative features from complex and variable data, such as faces in varying poses, expressions, and lighting conditions.

The core goal is to generate facial embeddings, which are compact representations of faces in a high-dimensional space. The embeddings are then used to compare faces using distance metrics, such as Euclidean distance or cosine similarity, which tell us how similar or different two faces are. The model is trained using triplet loss, a crucial component for learning these embeddings in a way that optimally separates individuals' facial features, pushing embeddings of the same person closer and embeddings of different people further apart.

The Inception ResNet V1 model is built with a combination of innovative layers that allow it to effectively capture and distinguish facial features while overcoming issues typically encountered in deeper networks, like vanishing gradients.

- **Initial Convolutional Layers**: These layers begin by extracting basic features from the input image, such as edges and textures, crucial for distinguishing different facial shapes.

- **Inception Blocks**: These blocks use multiple parallel convolutional operations with different filter sizes allowing the model to capture both fine details (like wrinkles and skin texture) and coarser features (like face shape) simultaneously. This multi-pathway design ensures that the model is robust to variations in face appearance.

- **Residual Connections**: To handle deeper network architectures without losing accuracy, ResNet style residual connections are used. These connections help prevent the issue of vanishing gradients by allowing the network to bypass certain layers, making it feasible to train deeper, more powerful models. These connections also allow the model to focus on the most important features and ensure that learning remains stable throughout training.

- **Global Average Pooling**: This operation aggregates the feature maps generated by the previous layers into a single, fixed-size output. This step is crucial because it ensures the model is robust to input image size variations, enabling it to handle images of different dimensions without compromising performance.

- **Fully Connected Layers**: Finally, the model outputs a 512-dimensional embedding that uniquely represents the face in a compressed, high-dimensional vector. These embeddings capture essential features, like the relative positions of facial landmarks, which are vital for distinguishing between different individuals.

A key aspect that makes the Inception ResNet V1 model so effective for face recognition is the integration of triplet loss with L2 normalization in the training process. This ensures that all embeddings are of unit magnitude (i.e., they have a magnitude of 1). By normalizing the embeddings, we remove the influence of varying scales across different embeddings.

Obviously, as the input images must undergo a series of preprocessing steps mentioned in previous sections to ensure that the network receives data in a format that facilitates learning. These steps are also critical for ensuring that the embeddings generated are accurate and reliable (resizing, normalization, face detection…)

Once the images are preprocessed, they are passed through the Inception ResNet V1 model, which generates the 512-dimensional facial embeddings. These embeddings, representing the unique identity of each face, are then compared during the recognition process using cosine similarity or Euclidean distance.

Another challenge addressed by the architecture is handling multiple images of the same person, as faces may appear differently due to changes in pose, lighting, or expression. To solve this, the embeddings for each image are generated separately. However, rather than storing a unique embedding for each image, we aggregate the embeddings and save them in a single .npy file

for each individual. This strategy ensures efficient storage and retrieval of embeddings while still enabling effective face comparison.

When performing recognition, the model compares the embeddings from any image of a given person with those from other individuals in the dataset. This flexibility allows the model to work effectively in real-world scenarios, where a person may be captured in multiple conditions.

The combination of Inception ResNet V1, L2 normalization, dual filtering of face detectors and carefully preprocessed input data ensures that our face recognition system is both accurate and efficient. The triplet loss encourages the network to learn embeddings that are not just informative but also well-separated, while L2 normalization ensures that comparisons between faces are consistent and reliable. Together, these components work correctly to deliver a powerful and efficient model capable of recognizing faces with high accuracy.

3.3 Fine-Tuning and Decision to Use a Pretrained Model

During the initial phase of the project, we experimented with fine-tuning a facial recognition model using our own dataset. We started by setting up a fine-tuning pipeline for our custom dataset, leveraging the power of convolutional neural networks (CNNs). Despite these efforts, several challenges arose that made us reconsider the approach:

- **Computational Resources**: Fine-tuning a model from scratch requires substantial computational resources. Since the time required to train the model from our dataset was over 18h for our limited computational resources, we started considering other options such as using a pretrained model.

- **Dataset Limitations**: While our custom dataset had a reasonable amount of data, it was still smaller compared to well-established datasets like VGFace2.

After evaluating these challenges, we made the decision to switch to a pretrained model. We chose a pretrained model that was already trained on a large and diverse dataset, such as VGGFace2, which had over 3 million of labeled images covering various lighting conditions, angles, and ethnic groups. This model was better equipped to handle the variability we expected in a real-world environment, such as our student library use case.

However, it’s important to note that with access to better computational resources and a larger, more specific dataset tailored to the target domain,

training a model from scratch could become a more viable and effective option. In cases where a highly specialized biometric system is needed building a custom model could offer more accurate and tailored results. With sufficient computational power, this approach would allow for faster, more efficient training and could better capture the specific properties of the dataset, surpassing the capabilities of pretrained models in certain contexts.

3.4 InceptionResnetV1 and Other Techniques in Facial Recognition

From the very beginning of this project, we seriously considered using FaceNet due to its prominent reputation in facial recognition, particularly because of its ability to generate facial embeddings through convolutional neural networks (CNN) and train using a triplet loss function.

However, as the project progressed, several technical challenges arose that led us to reconsider the use of FaceNet. One of the main issues was computational resource limitations. Since FaceNet is a model that requires intensive training due to the dataset size and model complexity, running it on machines with really limited resources resulted in excessively long training times and, in some cases, even failures due to memory constraints. At this point, we decided to move the work to Google Colab, where we could access more powerful hardware, such as GPUs, which would allow us to process data more efficiently.

However, moving to Google Colab also came with its own set of problems, such as dependency conflicts and issues with environment configuration. Despite having better access to the necessary computational resources, we encountered several library version mismatches and incompatibilities with the Python version or the Colab runtime. These issues were critical for loading the FaceNet model. For example, after installing TensorFlow version 1.15.0, which was the required version for the specific implementation, the model was able to load without any further issues. However, during the installation, some compatibility problems arose due to the version of protobuf installed on the system. TensorFlow 1.15.0 required a specific version of protobuf, which caused errors when using the default version of it installed on the system. This was resolved by downgrading protobuf to version 3.20.x, which made it compatible with TensorFlow 1.15.0, eliminating the error but causing a new dependency conflict repeatedly. These conflicts led to significant time loss in setting up the environment and debugging errors.

As we evaluated alternative models, we decided to shift to the Inception ResNet V1 model, which is a popular architecture for facial recognition

tasks. This model is pre-trained using large-scale datasets like VGGFace2, which is well-suited for handling variations in lighting, pose angles, and ethnic diversity, which was important for our project since the image capture conditions could vary significantly.

What was most important was that the Inception ResNet V1 model demonstrated more stable performance within the Colab environment. Its pre-trained weights were easier to integrate into our project, requiring fewer configuration steps. Despite the initial challenges with setting up the environment, the experience was much smoother in Google Colab compared to running it locally, which made it a more viable option moving forward with the project.

Model	Strengths	Weaknesses	Suitable Use Case
Inception ResNet V1	Robust to variations in lighting, pose, and ethnicity	Computationally demanding	Facial recognition tasks with diverse datasets (e.g., VGGFace2)
EfficientNet	Optimized for computational efficiency	Can be hard to tune	Real-time applications, mobile
FaceNet	High accuracy with triplet loss	Limited to smaller datasets	Face verification, small-scale systems
ArcFace	Enhanced separation between faces	May require larger datasets	Large-scale systems, more diverse datasets

Figure 4: Comparison of CNN Models for Facial Recognition.

3.5 Comparison of CNNs to traditional machine learning methods

CNNs are the standard for face identification, surpassing traditional methods like PCA and LDA in accuracy, scalability, and robustness. PCA reduces dimensionality by projecting data onto directions of maximum variance, but it doesn't use class labels, limiting its ability to differentiate between individuals. LDA maximizes class separability by leveraging class labels, making it more suitable for classification, but it assumes normally distributed data and struggles with complex, non-linear patterns.

Both PCA and LDA rely on linear transformations, making them inadequate for handling large-scale datasets and diverse variations in faces, such as lighting, pose, and occlusion. CNNs overcome these limitations by learning

hierarchical, non-linear features directly from raw data, capturing patterns at multiple levels for superior accuracy.

CNNs also excel in scalability through techniques like data augmentation and transfer learning, enabling effective generalization on large, varied datasets. Their robustness stems from convolutional and pooling operations, which provide invariance to transformations like rotation and scaling, and reduce sensitivity to noise, ensuring reliable performance in real-world conditions.

3.6 Triplet Loss

We considered using the triplet loss function for training our face recognition model. Triplet loss works by comparing the embeddings of three face images: an anchor, a positive (same person as the anchor), and a negative (different person). The objective is to minimize the distance between the anchor and the positive, while maximizing the distance between the anchor and the negative. It computes the loss by measuring the Euclidean distances between these embeddings and ensuring the anchor-negative distance is greater than the anchor-positive distance, with a margin (alpha) to enforce this difference. If the distance difference is smaller than the margin, the loss is zero. This setup encourages the model to bring embeddings of the same person closer while pushing embeddings of different people apart, improving its ability to recognize faces.

Although triplet loss is a powerful approach, we ultimately decided not to implement it in our project. This was mainly because we opted to use a pretrained model for our face recognition task, which already provided robust and generalized embeddings trained on a large and diverse dataset. Given our specific context, where leveraging a pretrained model was more efficient, we didn't pursue training from scratch with triplet loss. However, in other contexts where training from the ground up with a custom dataset is more appropriate, triplet loss could be a valuable option to consider. Its ability to fine-tune a model for discriminative face recognition could lead to improved performance in specialized applications.

3.7 Normalization

We have chosen L2 normalization for our face recognition project because it ensures that the embeddings produced by the FaceNet model have a consistent scale with a unit norm (magnitude of 1). This is crucial for tasks like face recognition, where distance-based metrics, such as Euclidean or

cosine distance, are used to compare embeddings. L2 normalization removes the influence of varying magnitudes across different embeddings, making comparisons focus on the direction (or similarity) of the embeddings, rather than their size. By normalizing the embeddings, we ensure that the distances between similar faces are smaller and the distances between dissimilar faces are larger, which allows for more accurate and stable comparisons.

Additionally, L2 normalization is important for maintaining consistency and reliability in the embedding space, especially when using the triplet loss function. Triplet loss relies on accurate distance measurements between anchor, positive, and negative samples, and L2 normalization guarantees that these distances are comparable, improving the model's ability to differentiate between different individuals. Ultimately, L2 normalization enhances the model's ability to generalize, making it more effective for real-world face recognition tasks.

3.8 Choice of Embedding Dimensionality

The choice of embedding dimensionality directly impacts a system's performance, speed, and resource efficiency. In this project, we are using the InceptionResnetV1 model pre-trained on VGGFace2, which generates 512-dimensional embeddings. This embedding size is suitable for capturing detailed facial features, which improves accuracy in face identification and verification, particularly in complex tasks that require distinguishing between individuals with subtle facial characteristics.

The 512-dimensional embeddings strike a balance between the ability to represent detailed facial information and computational efficiency. While embeddings with higher dimensionality could capture more features, they would also increase computational and storage costs. On the other hand, embeddings with fewer dimensions, such as 128-dimensional embeddings, might lose critical information, which would affect the system's accuracy.

Therefore, the 512-dimensional embedding provides a good balance between accuracy and efficiency for this face recognition system, enabling robust identification while maintaining computational feasibility.

4 Classification

For classification purposes, we utilized Support Vector Machine (SVM), a well-known machine learning algorithm that is particularly effective for classification tasks, including face recognition. SVM works by finding the optimal hyperplane that maximizes the margin between different classes in a high-dimensional feature space. This makes SVM a powerful tool for problems where the data is not necessarily linearly separable but can be mapped to a higher-dimensional space, which is the case for face embeddings.

To organize and manage the data, we arranged the embeddings of each person into separate folders. The folder names themselves served as labels that identified each individual. Specifically, we created distinct folders for the training, test, and validation embeddings, ensuring that the embeddings for each person were stored separately. The training data, which is used to train the SVM model, was loaded from these directories, and the corresponding labels (person IDs) were assigned based on the folder names.

Once the SVM model was trained on the training dataset, it was evaluated using the validation data. The validation dataset is crucial in machine learning as it is typically used to fine-tune hyperparameters and prevent overfitting to the training data. During training, the model's hyperparameters, such as the regularization parameter (C) and the choice of kernel, can be adjusted based on the model's performance on the validation set. This allows for better generalization when the model encounters unseen data. The trained model then made predictions on the validation set, and its performance was assessed by comparing the predicted labels with the true labels using the accuracy metric.

The final performance metric reported was accuracy, which measures the percentage of correct predictions made by the classifier on the validation set. Initial tests on a larger dataset yielded a relatively low accuracy of around 68%, suggesting challenges in handling high-class variability. However, further investigation revealed that the SVM performed significantly better on smaller datasets, achieving accuracies of up to 82% when the number of classes was reduced. This observation highlights the SVM's limitations in scaling to large datasets but also its potential effectiveness in scenarios with fewer identities, where it can deliver reliable results with simpler computational requirements.

5 Results

The verification process of embeddings showed that all 1,020 embeddings adhered to the expected dimensionality of 512, with no invalid entries detected. This indicates that the preprocessing pipeline was consistent and robust.

An analysis of mean cosine similarity provided insights into the embeddings' performance in distinguishing between individuals. For each person, the mean similarity of their embeddings with others in the same class (self-similarity) and those from different classes (other similarity) was computed. Self-similarity values ranged from 0.1299 to 0.8429, suggesting some variability in intra-class similarity, possibly due to differences in pose, lighting, or occlusion. Meanwhile, other similarity scores were consistently higher, ranging from 0.9355 to 1.0342, confirming that the embeddings effectively separated individuals into distinct classes.

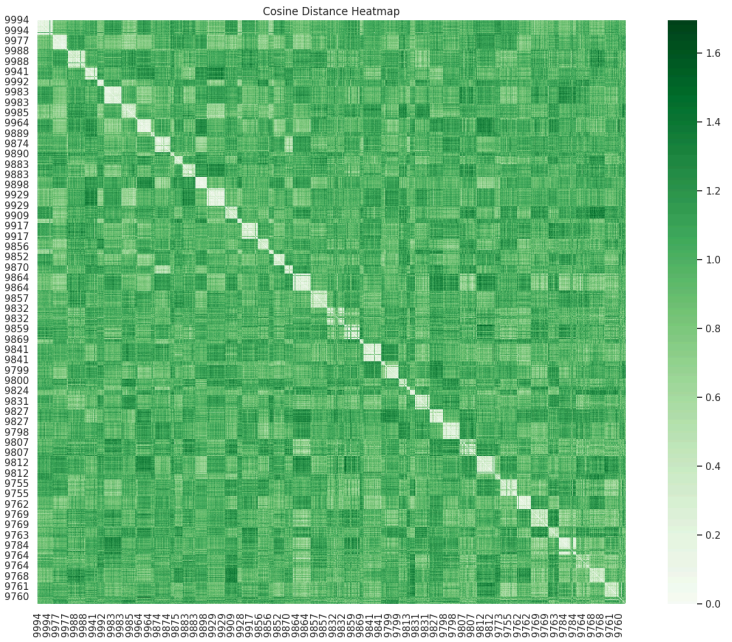

Figure 4: Cosine Distance Heatmap

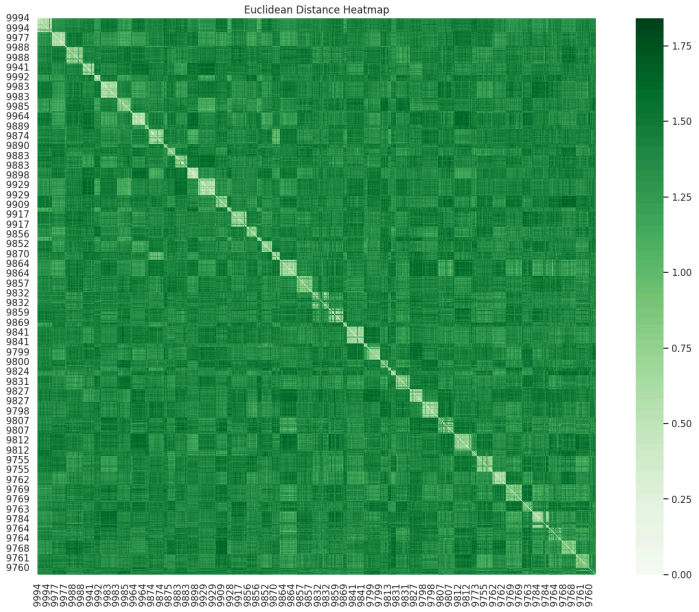

Figure 5: Euclidean Distance Heatmap

We determined that the optimal threshold for our face recognition system is 0.77. This threshold was selected as it offers an excellent trade-off between different performance metrics, ensuring that the system operates with both high accuracy and reliability in recognizing faces. A deeper look into the key performance metrics at this threshold provides insight into how well the system performs in real-world scenarios.

The **Genuine Acceptance Rate** (GAR) represents the percentage of true matches that are correctly accepted by the system. With a GAR value of 0.9304, the system successfully accepts 93.04% of genuine pairs. This metric is crucial for evaluating the system's ability to correctly identify individuals from a database of embeddings. A high GAR value indicates that the system is highly effective at recognizing faces that actually belong to the same person. This is important in real-world applications such as access control or identity verification, where genuine users must be consistently recognized.

The **Genuine Rejection Rate** (GRR) indicates the rate at which the system correctly rejects non-matching pairs. A GRR of 0.9166 means that the system correctly identifies and rejects 91.66% of pairs where the faces belong to different individuals. This is an essential metric for ensuring that the system accurately discriminates between non-matching faces, minimizing the risk of false acceptances and enhancing security.

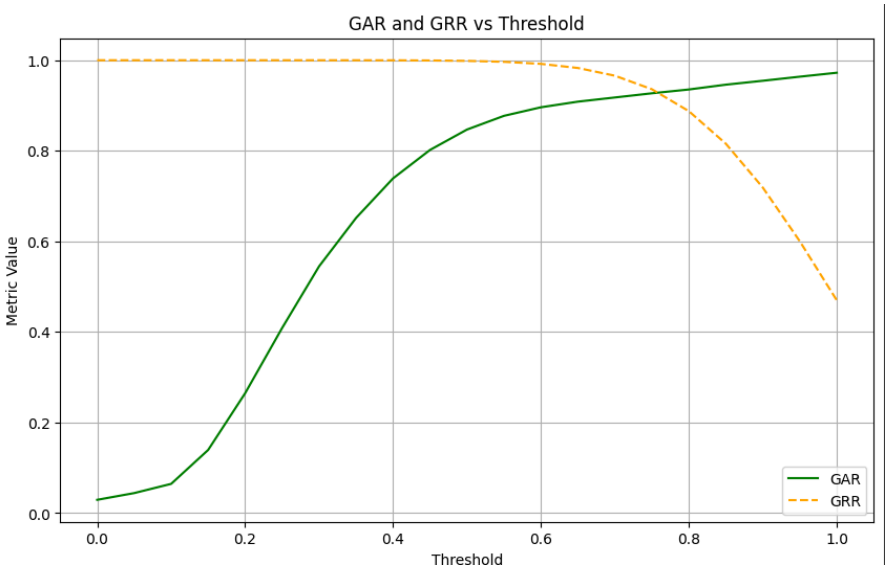

Figure 6: Comparison of GAR and GRR vs Threshold

The **False Acceptance Rate** (FAR) refers to the proportion of non-genuine pairs (i.e., faces of different individuals) that the system incorrectly classifies as a match. In this case, the FAR value of 0.0834 means that 8.34% of non-matching pairs are mistakenly accepted as genuine matches by the system. While this rate is not negligible, it is a reasonable trade-off considering the overall effectiveness of the system. A lower FAR would generally come at the expense of a higher False Rejection Rate (FRR), so this value shows a balanced approach, minimizing the risk of incorrect identification without overcompensating.

The **False Rejection Rate** (FRR) is a measure of the rate at which genuine pairs (i.e., faces from the same individual) are incorrectly rejected by the system. At a value of 0.0696, the system rejects 6.96% of true matches. While a lower FRR is always desirable to avoid unnecessary rejections, this value indicates that the system is correctly rejecting the vast majority of non-matching pairs, while keeping the number of false rejections relatively low. It shows that the system is designed to prioritize accuracy while maintaining an acceptable level of false rejections.

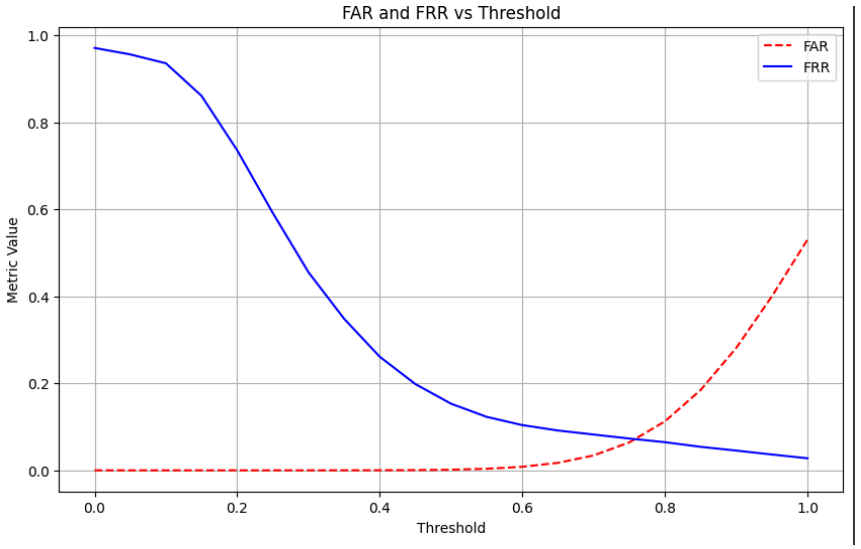

Figure 7: Comparison between FAR & FRR and Threshold Graphic

The **ROC curve**, which measures the system's ability to distinguish between positive and negative samples, achieved an impressive score of 0.97. This value indicates a very high true positive rate relative to the false positive rate, showing that the system is highly effective in correctly identifying authorized individuals while minimizing false positives. A ROC score of 0.97 suggests that the facial recognition model has excellent discriminative power, making it well-suited for practical deployment in environments like university entrances.

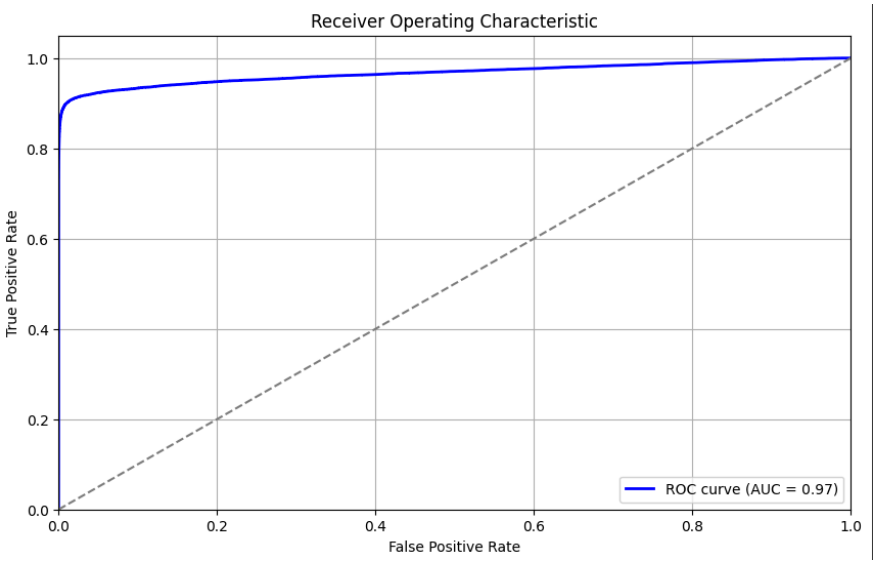

Figure 8: Receiver Operating Characteristic Graphic

Finally, **Accuracy** is the overall performance measure of the system, combining both true acceptances and true rejections. At a value of 0.9166, the system has an accuracy of 91.66%, meaning that it correctly classifies both positive and negative pairs with this level of effectiveness. This high accuracy signifies that the system is highly reliable and performs well overall. The accuracy value takes into account the balance between FAR, FRR, GAR, and GRR, demonstrating the system's ability to consistently perform correctly across various scenarios.

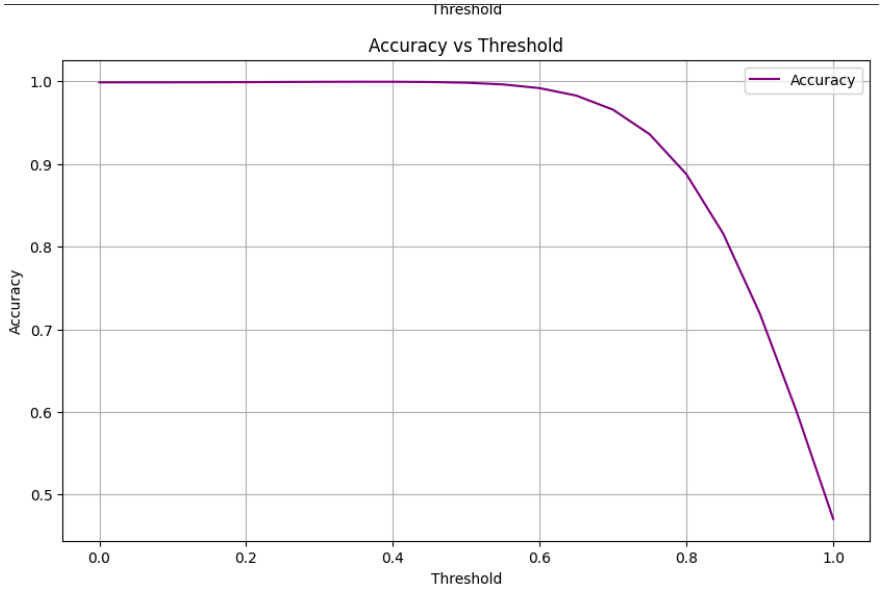

Figure 6: Receiver Operating Characteristic Graphic

The optimal **threshold of 0.77** provides a well-balanced performance, ensuring that the system operates with minimal error while maximizing reliability. The FAR of 0.0834 and FRR of 0.0696 indicate that the system effectively minimizes false positives and false negatives. Meanwhile, the GAR of 0.9304 and GRR of 0.9166 suggest that the system is highly effective in correctly identifying true matches and rejecting non-matching pairs. Overall, the system achieves an accuracy of 91.66%, which is a strong indication of its reliability and robustness.

In conclusion, the development of our facial recognition system marks a significant advancement toward enhancing security and efficiency at Sapienza University. Throughout the project, we have successfully considered and implemented key components, which have ultimately led us to achieving great results that could definitely help students and professors all over the university. With a solid balance of performance metrics and an accuracy of 91.66%, the system demonstrates its potential to streamline access control and improve security across the university.

6 References

1. LearnOpenCV. (n.d.). Face Detection using OpenCV, Dlib, and Deep Learning (C/Python): https://learnopencv.com/face-detection-opencv-dlib-and-deep-learning-c-python/

2. In-depth explanation of face detection techniques using OpenCV, Dlib, and deep learning: https://learnopencv.com/face-detection-opencv-dlib-and-deep-learning-c-python/

3. The CelebA Dataset. (2015). CelebA: Large-scale CelebFaces Attributes Dataset: https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

4. OpenCV. (n.d.). OpenCV: Open Source Computer Vision Library: https://opencv.org/

7 Appendix

7.1 Script for Data Preprocessing and Face Detection using Dlib

This script processes and organizes images based on whether they contain a detected face, using Dlib's pre-trained face detector. The images are then sorted into appropriate folders for training, validation, and testing.

home > guillermobajo > Descargas > distributeSamples.py > ...

```python
import os
import shutil
import pandas as pd
import dlib
from tqdm import tqdm

# Define folder paths
base_dir = "celeba_dataset/"  # Folder where the images will be organized
images_dir = "img_align_celeba/"  # Path to the downloaded images
partition_file = "list_eval_partition.csv"  # Partition file
identity_file = "identity_CelebA.txt"  # File with image-to-ID mapping

# Create directories for train, val, and test
train_dir = os.path.join(base_dir, "train/")
val_dir = os.path.join(base_dir, "val/")
test_dir = os.path.join(base_dir, "test/")

# Create the destination folders if they don't exist
os.makedirs(train_dir, exist_ok=True)
os.makedirs(val_dir, exist_ok=True)
os.makedirs(test_dir, exist_ok=True)

# Load the pre-trained face detector from Dlib
detector = dlib.get_frontal_face_detector()

# Read the partition file (list_eval_partition.csv)
print("Reading partition file...")
partitions = pd.read_csv(partition_file, header=None, names=["image", "partition"], skiprows=1)

# Read the identity file (identity_CelebA.txt) which is a text file
print("Reading identity file...")
identities = {}
with open(identity_file, "r") as f:
    for line in f.readlines():
        parts = line.split()
        img_name = parts[0]  # Image name (e.g., 000001.jpg)
        person_id = parts[1]  # Person ID (e.g., 1)
        identities[img_name] = person_id

# Merge partition and identity data
print("Merging partition and identity data...")
data = pd.merge(partitions, pd.DataFrame(list(identities.items()), columns=["image", "id"]), on="image")

# Function to check if a face is detected in an image using Dlib
def is_face_detected(image_path):
    # Load image using Dlib
    img = dlib.load_rgb_image(image_path)  # Dlib uses RGB images

    # Detect faces in the image
    faces = detector(img, 1)  # Detect faces with a max of 1 pyramid level

    # If faces are detected, return True, otherwise False
    return len(faces) > 0

# Initialize counters
total_images_processed = 0
images_discarded = 0
images_accepted = 0
```

```
train_images = 0
val_images = 0
test_images = 0

# Process images and organize them
print("Organizing images into folders...")
for _, row in tqdm(data.iterrows(), total=data.shape[0]):
    img_name = row["image"]
    partition = row["partition"]
    person_id = str(row["id"])  # Get the ID of the person

    # Ensure the image name has 6 digits, and add .jpg if it's not present
    img_name_with_extension = f"{img_name.zfill(6)}.jpg" if not img_name.endswith('.jpg') else img_name

    # Source image path
    src_path = os.path.join(images_dir, img_name_with_extension)

    # Increment total images processed
    total_images_processed += 1

    # Check if the face is detected in the image
    if not is_face_detected(src_path):
        images_discarded += 1
        continue  # Skip the image if no face is detected

    # Increment accepted images
    images_accepted += 1

    # Determine the destination folder based on partition
    if partition == 0:
        dest_folder = train_dir
        train_images += 1
    elif partition == 1:
        dest_folder = val_dir
        val_images += 1
    elif partition == 2:
        dest_folder = test_dir
        test_images += 1
    else:
        continue  # Skip if the partition is not recognized

    # Create a subfolder for the person (ID)
    person_folder = os.path.join(dest_folder, f"{person_id}")
    os.makedirs(person_folder, exist_ok=True)

    # Destination image path
    dest_path = os.path.join(person_folder, img_name_with_extension)

    # Copy the image to the destination folder
    if os.path.exists(src_path):
        shutil.copy(src_path, dest_path)

# Print summary of the processing
print("\nProcessing complete!")
print(f"Total images processed: {total_images_processed}")
print(f"Total images discarded (no face detected): {images_discarded}")
print(f"Total images accepted: {images_accepted}")
print(f"Total images in Train folder: {train_images}")
print(f"Total images in Validation folder: {val_images}")
print(f"Total images in Test folder: {test_images}")
```

7.2 Script for Embedding generation

This script generates the embeddings for all the elements corresponding to the test set. It uses Inception's ResNet V1 model pretrained with the VGGFace2 dataset. An almost identical code was used to generate the embeddings from the validation set. Different scripts for checking the quality of the corresponding embeddings.

home > guillermobajo > Descargas > distributeSamples.py > ...

```python
# Code to generate embeddings
import os
from facenet_pytorch import InceptionResnetV1, MTCNN
from PIL import Image
import numpy as np
from tqdm import tqdm

# Initialize MTCNN and the InceptionResnetV1 model
mtcnn = MTCNN(keep_all=True)
model = InceptionResnetV1(pretrained='vggface2').eval()

# Path to the test dataset
test_dataset_path = '/content/celeba_dataset/test'

# Path to store the embeddings
embedding_dir = '/content/embeddings'
os.makedirs(embedding_dir, exist_ok=True)

# Function to save the embeddings in a .npy file
def save_embeddings(person_folder, embeddings):
    person_embedding_dir = os.path.join(embedding_dir, person_folder)
    os.makedirs(person_embedding_dir, exist_ok=True)
    embeddings_path = os.path.join(person_embedding_dir, 'embeddings.npy')
    np.save(embeddings_path, embeddings)

# Generate embeddings for all people in the test folder
person_folders = sorted(os.listdir(test_dataset_path))

for person_folder in tqdm(person_folders, desc="Processing persons", unit="person"):
    person_path = os.path.join(test_dataset_path, person_folder)
    if os.path.isdir(person_path):
        embeddings = []

        for image_file in os.listdir(person_path):
            image_path = os.path.join(person_path, image_file)
            try:
                image = Image.open(image_path).convert('RGB')

                # Detect faces in the image
                faces, _ = mtcnn(image, return_prob=True)
                if faces is not None:
                    for face in faces:
                        face = face.unsqueeze(0)
                        embedding = model(face).detach().numpy().flatten()
                        embeddings.append(embedding)
            except Exception as e:
                print(f"Error processing the image {image_path}: {e}")

        if embeddings:
            save_embeddings(person_folder, embeddings)

print("Embedding generation process completed.")
```

7.3 Script for Fine Tunning

Although the fine-tuning approach was ultimately not used due to the reasons outlined in the report, the following code represents the script that was developed during the experimentation phase.

```
from PIL import Image
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, Dataset
from tqdm import tqdm, trange
import time
from torchvision import transforms

# Define paths to datasets
data_dir = "celeba_dataset"
train_dir = os.path.join(data_dir, "train")
val_dir = os.path.join(data_dir, "val")
test_dir = os.path.join(data_dir, "test")

# Initialize MTCNN for extracting faces from images
mtcnn = MTCNN(keep_all=True)

# Define a transformation pipeline to convert images to tensors
transform = transforms.Compose([
    transforms.Resize((160, 160)),  # Resize images to 160x160
    transforms.ToTensor(),  # Convert PIL image to a tensor
])

class FaceDataset(Dataset):
    def __init__(self, base_dir):
        self.samples = []  # List to store image file paths
        self.labels = []  # List to store labels for each image
        self.label_map = {}  # Mapping from person ID to label index

        # Iterate over each person in the dataset
        for label_idx, person_id in enumerate(sorted(os.listdir(base_dir))):
            person_path = os.path.join(base_dir, person_id)
            if os.path.isdir(person_path):  # Only consider directories (each person)
                self.label_map[person_id] = label_idx  # Assign a label to the person
                # Add images of the person to the dataset
                for img_name in os.listdir(person_path):
                    img_path = os.path.join(person_path, img_name)
                    self.samples.append(img_path)
                    self.labels.append(label_idx)

    def __len__(self):
        return len(self.samples)  # Return the number of samples in the dataset

    def __getitem__(self, idx):
        img_path = self.samples[idx]
        label = self.labels[idx]
        img = Image.open(img_path).convert('RGB')  # Load and convert image to RGB

        # Use MTCNN to detect faces in the image
        faces = mtcnn(img)
        if faces is None or len(faces) == 0:
            # If no faces are detected, resize the image to 160x160 (input size for InceptionResnetV1)
            img = img.resize((160, 160))
        else:
            img = faces[0]  # Use the first face detected

        # Ensure that the image is of type PIL before applying the transformations
        if isinstance(img, torch.Tensor):
            img = transforms.ToPILImage()(img)  # Convert tensor back to PIL Image if necessary

        # Apply transformations (resize and convert to tensor)
        img = transform(img)

        return img, label  # Return the face image (as a tensor) and its corresponding label

# Load training and validation datasets
train_dataset = FaceDataset(train_dir)
val_dataset = FaceDataset(val_dir)

# Create DataLoader objects for batching and shuffling data
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16)

# Initialize the InceptionResnetV1 model pre-trained on the VGGFace2 dataset
model = InceptionResnetV1(pretrained='vggface2', classify=True, num_classes=len(train_dataset.label_map)).train()

# Freeze all layers of the model except the final classification layer (logits)
for param in model.parameters():
    param.requires_grad = False  # Freeze all parameters
for param in model.logits.parameters():
    param.requires_grad = True  # Unfreeze the parameters of the classification layer

# Define the loss function (cross-entropy loss) and the optimizer (Adam)
```

```python

# Define the loss function (cross-entropy loss) and the optimizer (Adam)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=1e-4)

def train_epoch(model, dataloader, optimizer, criterion, progress_bar):
    model.train()  # Set the model to training mode
    total_loss = 0
    correct = 0
    total = 0
    # Iterate over the training data in batches
    for imgs, labels in dataloader:
        imgs = imgs.cuda()  # Move images to the GPU
        labels = torch.tensor(labels).cuda()  # Move labels to the GPU
        optimizer.zero_grad()  # Reset gradients from the previous batch
        outputs = model(imgs)  # Forward pass through the model
        loss = criterion(outputs, labels)  # Calculate loss
        loss.backward()  # Backpropagate the loss
        optimizer.step()  # Update model weights

        total_loss += loss.item()  # Accumulate the loss
        preds = outputs.argmax(dim=1)  # Get the predicted labels
        correct += (preds == labels).sum().item()  # Count correct predictions
        total += labels.size(0)  # Track the total number of labels

        # Update progress bar with current loss and accuracy
        progress_bar.set_postfix({"Loss": total_loss / (total or 1), "Accuracy": correct / (total or 1)})
        progress_bar.update(1)  # Move the progress bar one step forward

    return total_loss / len(dataloader), correct / total  # Return average loss and accuracy

def validate_epoch(model, dataloader, criterion, progress_bar):
    model.eval()  # Set the model to evaluation mode
    total_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():  # Disable gradient computation during validation
        # Iterate over the validation data in batches
        for imgs, labels in dataloader:
            imgs = imgs.cuda()  # Move images to the GPU
            labels = torch.tensor(labels).cuda()  # Move labels to the GPU
            outputs = model(imgs)  # Forward pass through the model
            loss = criterion(outputs, labels)  # Calculate loss

            total_loss += loss.item()  # Accumulate the loss
            preds = outputs.argmax(dim=1)  # Get the predicted labels
            correct += (preds == labels).sum().item()  # Count correct predictions
            total += labels.size(0)  # Track the total number of labels

            # Update progress bar with current loss and accuracy
            progress_bar.set_postfix({"Loss": total_loss / (total or 1), "Accuracy": correct / (total or 1)})
            progress_bar.update(1)  # Move the progress bar one step forward

    return total_loss / len(dataloader), correct / total  # Return average loss and accuracy

# Fine-tuning loop
num_epochs = 5  # Number of training epochs
model = model.cuda()  # Move the model to the GPU

# Create a single progress bar for all epochs
with tqdm(total=len(train_loader) * num_epochs, desc="Training & Validating", ncols=100,
          bar_format="{l_bar}{bar}| {n_fmt}/{total_fmt} [Time: {elapsed} | Remaining: {remaining}] {percentage:3.0f}%") as progress_bar:

    for epoch in trange(num_epochs, desc="Epochs"):
        start_time = time.time()  # Record the start time of the epoch
        # Train for one epoch
        train_loss, train_acc = train_epoch(model, train_loader, optimizer, criterion, progress_bar)
        # Validate after each epoch
        val_loss, val_acc = validate_epoch(model, val_loader, criterion, progress_bar)
        end_time = time.time()  # Record the end time of the epoch

        # Print the training and validation results for the epoch
        print(f"\nEpoch {epoch + 1}/{num_epochs}")
        print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}")
        print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
        print(f"Epoch Time: {end_time - start_time:.2f} seconds")

# Save the fine-tuned model's weights to a file
torch.save(model.state_dict(), os.path.join(data_dir, "fine_tuned_model.pth"))
print("Fine-tuning complete!")

```

7.4 Training and Evaluation of SVM Classifier for Face Recognition using Embeddings

This script loads face embeddings from training and validation datasets, assigns labels based on folder names, and trains an SVM classifier using the embeddings. The SVM model is trained with a linear kernel, with progress tracked through a simulated progress bar. After training, the model is tested on the validation data, and its accuracy is evaluated by comparing the predicted labels with the true labels. The final accuracy score is printed for evaluation.

```
import os
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import time

# Directories
embedding_dir = '/content/embeddings'
embedding_val_dir = '/content/embeddingsVal'

# Initialize lists
embeddings_val_list = []
labels_val_list = []
embeddings_list = []
labels_list = []

# Load the embeddings for each person in the training set
person_folders = sorted(os.listdir(embedding_dir))

for person_folder in person_folders:
    person_embedding_dir = os.path.join(embedding_dir, person_folder)
    embeddings_path = os.path.join(person_embedding_dir, 'embeddings.npy')

    if os.path.exists(embeddings_path):  # If the embeddings file exists
        # Load the embeddings
        embeddings = np.load(embeddings_path)

        # Add all the embeddings and the corresponding label (person's name)
        for embedding in embeddings:
            embeddings_list.append(embedding)
            labels_list.append(person_folder)  # The folder name is the label

# Convert the lists to numpy arrays
X_train = np.array(embeddings_list)  # The training embeddings
y_train = np.array(labels_list)  # The training labels

# Verify that the data has been loaded correctly
print(f"Training data shape: {X_train.shape}")

# Function to train the model and show the progress bar
def train_svm_with_progress(X_train, y_train):
    # Initialize the SVM classifier
    svm_model = SVC(kernel='linear')

    # Create an empty list to simulate progress, since SVM does not have a direct progress bar
    # This is done by splitting the dataset into smaller blocks for tqdm to iterate over
    n_samples = X_train.shape[0]
```

```
    chunk_size = max(1, n_samples // 100)  # Split into 100 chunks to show progress

    # Use tqdm to simulate progress during training
    with tqdm(total=n_samples, desc="Training SVM", unit="sample") as pbar:
        # Train in small blocks to show progress
        for i in range(0, n_samples, chunk_size):
            # The actual training with SVM is done once
            svm_model.fit(X_train, y_train)  # Train the model

            # Simulate progress with the progress bar
            pbar.update(chunk_size)
            time.sleep(0.1)  # Simulate a small delay

    # Return the trained model
    return svm_model

# Call the function to train the SVM model and show the progress bar
svm_model = train_svm_with_progress(X_train, y_train)

# Verify that the model has been trained correctly
print("SVM model trained successfully.")

# Load the embeddings for each person in the validation set
person_folders_val = sorted(os.listdir(embedding_val_dir))

for person_folder in person_folders_val:
    person_embedding_dir = os.path.join(embedding_val_dir, person_folder)
    embeddings_path = os.path.join(person_embedding_dir, 'embeddings.npy')

    if os.path.exists(embeddings_path):  # If the embeddings file exists
        # Load the embeddings
        embeddings = np.load(embeddings_path)

        # Add all embeddings and the corresponding label (person's name)
        for embedding in embeddings:
            embeddings_val_list.append(embedding)
            labels_val_list.append(person_folder)  # The folder name is the label

# Convert the lists to numpy arrays
X_val = np.array(embeddings_val_list)  # Validation embeddings
y_val = np.array(labels_val_list)  # Validation labels

# Make predictions with the trained model
y_pred = svm_model.predict(X_val)

# Evaluate the model with accuracy
accuracy = accuracy_score(y_val, y_pred)
print(f"Accuracy on validation set: {accuracy:.2f}")
```

7.5 Final testing and embeddings generation

This script was used to compute the final metrics obtained from the system's performance, used to later generate visual graphics as the ones seen in the results section.

```python
from sklearn.metrics.pairwise import cosine_distances
import time

# Calculate metrics by comparing finaltest embeddings with original embeddings
print("Loading embeddings for metric calculation...")
start_time = time.time()

original_embeddings = []  # Original embeddings
original_labels = []  # Original labels

for person_folder in tqdm(os.listdir(embedding_dir), desc="Loading originals", unit="folder"):
    person_path = os.path.join(embedding_dir, person_folder)
    if os.path.isdir(person_path):
        embeddings_file = os.path.join(person_path, 'embeddings.npy')
        if os.path.exists(embeddings_file):
            embeddings = np.load(embeddings_file)
            original_embeddings.extend(embeddings)
            original_labels.extend([person_folder] * len(embeddings))

# Convert to numpy arrays
original_embeddings = np.array(original_embeddings)
original_labels = np.array(original_labels)

# Create distance matrix
print("Calculating distance matrix...")
DM = cosine_distances(probes, original_embeddings)

# Calculate metrics
thresholds = np.arange(0, 1.05, 0.05)  # Thresholds
FAR = []
FRR = []
GAR = []
GRR = []
accuracy = []

total_positive_pairs = np.sum([probe_labels[i] == original_labels[j] for i in range(len(probes)) for j in range(len(original_embeddings))])
total_negative_pairs = len(probes) * len(original_embeddings) - total_positive_pairs

print("Calculating metrics...")
for th in tqdm(thresholds, desc="Evaluating thresholds", unit="threshold"):
    false_accepts = 0
    false_rejects = 0
    true_accepts = 0
    true_rejects = 0

    for i in range(len(probes)):
        for j in range(len(original_embeddings)):
            if DM[i, j] <= th:  # Accepted pair
                if probe_labels[i] != original_labels[j]:
                    false_accepts += 1  # FAR
                else:
                    true_accepts += 1  # GAR
            else:  # Rejected pair
                if probe_labels[i] == original_labels[j]:
                    false_rejects += 1  # FRR
                else:
                    true_rejects += 1  # GRR

    # Calculate metrics
    FAR.append(false_accepts / total_negative_pairs if total_negative_pairs > 0 else 0)
    FRR.append(false_rejects / total_positive_pairs if total_positive_pairs > 0 else 0)
    GAR.append(true_accepts / total_positive_pairs if total_positive_pairs > 0 else 0)
    GRR.append(true_rejects / total_negative_pairs if total_negative_pairs > 0 else 0)
    accuracy.append((true_accepts + true_rejects) / (len(probes) * len(original_embeddings)))

elapsed_time = time.time() - start_time
print(f"Metric calculation completed in {elapsed_time:.2f} seconds.")

```