THOMAS MORE

A.I. FRAMEWORKS

Who Is In The Picture

2025 - 2026

Table of Contents

Introduction

In this project, a face recognition model is trained to predict the identities of individuals within a classroom, consisting of twelve distinct people. The available dataset contains images in which these individuals appear, but the labels are unreliable. Manually correcting each face annotation would be both time-consuming and error-prone, particularly due to variations in image quality and the uncertainty regarding which faces can be successfully detected in advance.

To address this challenge, an automated processing pipeline is proposed that minimizes manual effort. Face detection is performed using the RetinaFace model, which identifies and localizes faces in the images. Only faces that are successfully detected are retained and cropped, ensuring that the training data consists exclusively of faces that can be reliably processed by the detection model. This approach avoids overlabeling low-quality samples and improves the overall consistency of the dataset.

Once detected and aligned, faces are transformed into numerical representations using a pre-trained ArcFace model. These embeddings capture identity-related facial features in a compact 512-dimensional space. Since the dataset is unlabelled, unsupervised clustering is applied to group embeddings that are likely to belong to the same individual. This step enables the automatic discovery of identity groups without prior annotations.

Based on the resulting clusters, supervised classification models are trained to enable identity prediction for new, unseen face images.

By combining face detection, embedding extraction, clustering and classification, the proposed system forms a complete end-to-end face recognition pipeline.

1. Detect and Align Faces

1.1. Face Detection Model Selection

After evaluating the research into various face detection approaches, RetinaFace was selected as the face detector used in this project. RetinaFace offers a strong balance between detection accuracy and robustness in real-world conditions, making it suitable for uncontrolled environments.

Unlike other models, which only output bounding boxes, RetinaFace also predicts five facial landmarks (eyes, nose and mouth corners) along with a confidence score for each detection. These landmarks are essential for reliable face alignment, which is a key requirement for generating embeddings.

The detection performance of RetinaFace is illustrated in *Figure 1* using a confusion matrix, yielding an overall accuracy of 89.0% across 1064 evaluated samples. Most detections fall within the true positive category, indicating that the model reliably detects faces in the dataset. Although some false positives and false negatives remain, their proportion is relatively small, confirming that the detector is suitable for this application *(Amos Stailey-Young, 2024, fig. 1)*.

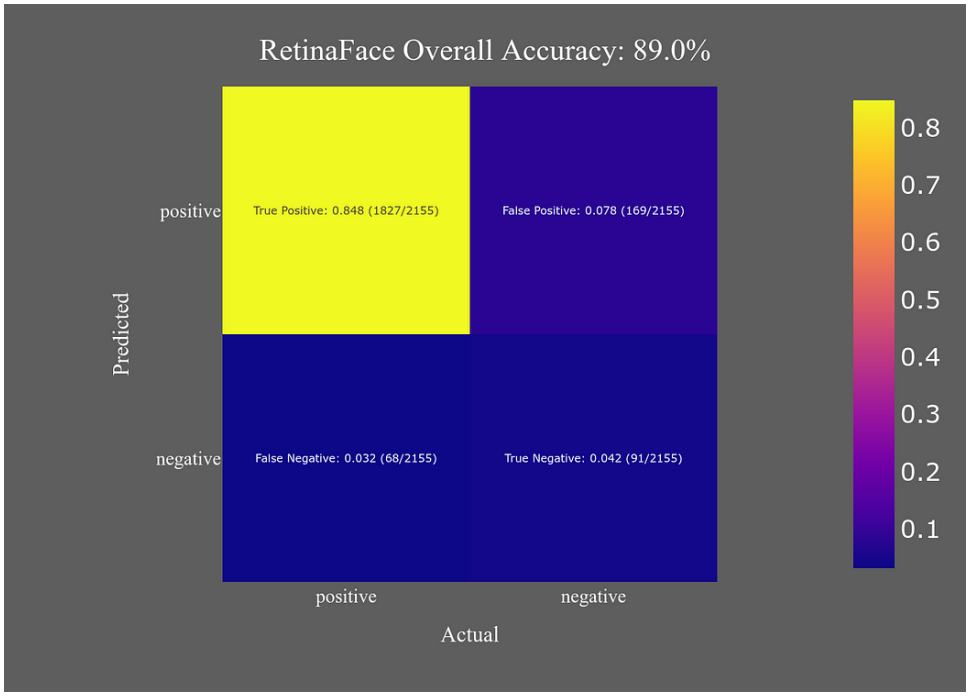


Figure 1: RetinaFace Confusion Matrix on Evaluation Dataset

1.1.1. RetinaFace

RetinaFace is a deep-learning-based face detector, provided through the Uniface library. It is designed to operate under challenging real-world conditions, including cluttered backgrounds and large variations in face appearance.

Figure 2 demonstrates that RetinaFace successfully detects both large, clearly visible faces and smaller faces in the background, even in a complex scene. This confirms the model's sensitivity to facial features under difficult conditions.

However, increased sensitivity can also result in false positive detections, where non-face regions resemble facial patterns, as shown in the image. These false detections can negatively affect the processing pipeline and should therefore be filtered out by applying a higher confidence threshold or by ignoring very small detections *(Amos Stailey-Young, 2024, fig. 2)*.

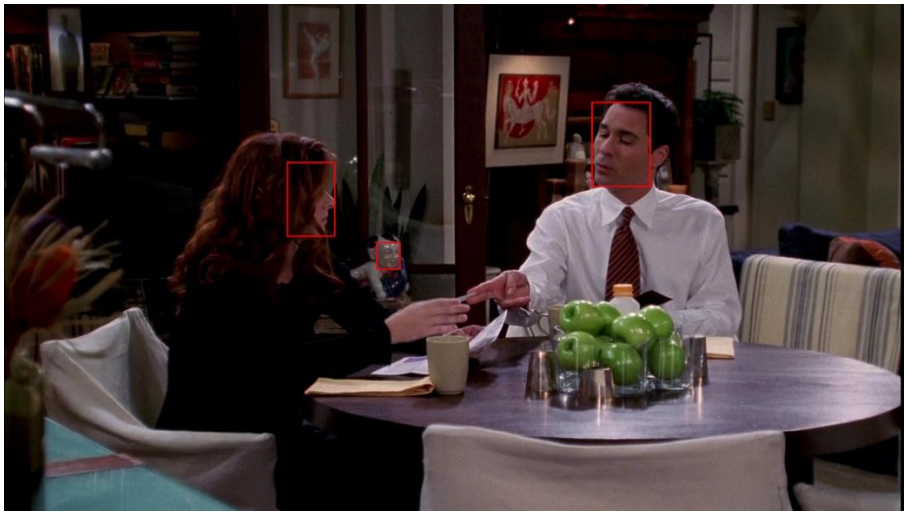


Figure 2: RetinaFace Detections

1.1.2. ResNet-34 Backbone

```
detector = RetinaFace(model_name=RetinaFaceWeights.RESNET34)
```

The ResNet-34 backbone is selected to balance detection accuracy and computational efficiency. It provides sufficient capacity to detect faces reliably while remaining fast enough to process large datasets.

Live detection results are shown in *Figure 3*, demonstrating stable detection across multiple individuals in an outdoor environment.

Figure 3: Live Detection

1.2. Goal and Context

Rather than risking the manual labelling of unusable face samples, the pipeline first detects faces, filters unreliable detections, aligns faces into a standardized format and stores both the processed images and detailed metadata.

This approach is chosen because face recognition models are sensitive to variations in pose, scale and alignment. By normalizing these factors during preprocessing, the learning task for the used models becomes significantly easier and more stable.

1.3. Detection Workflow

1.3.1. Metadata Logging and Traceability

To ensure full traceability of the preprocessing pipeline, metadata for each aligned face is stored in a CSV file:

```
writer.writerow([
    "aligned_file",
    "original_file",
    "face_index",
    "x1", "y1", "x2", "y2",
    "confidence"
])
```

This metadata includes the aligned face filename, the source image filename, the index of the detected face, bounding box coordinates and the detection confidence score. Storing this information enables quality control and reproducibility.

1.3.2. Face Detection

Each image is forwarded individually through the RetinaFace convolutional neural network, which performs dense, multi-scale predictions across the image to identify potential face regions. If an image cannot be loaded, it is skipped, making the script robust against corrupted or missing files.
If no faces are detected in an image, the image is displayed without annotations and processing continues with the next image.

```
faces = detector.detect(image)
```

For visualization and debugging purposes, a copy of the original image is created. This ensures that drawing operations do not modify the original data used for cropping and alignment.

1.3.3. Processing Detected Faces

RetinaFace returns a dictionary object for each detected face, including:

- A confidence score
- Facial landmarks (eyes, nose, mouth corners)
- Bounding box coordinates

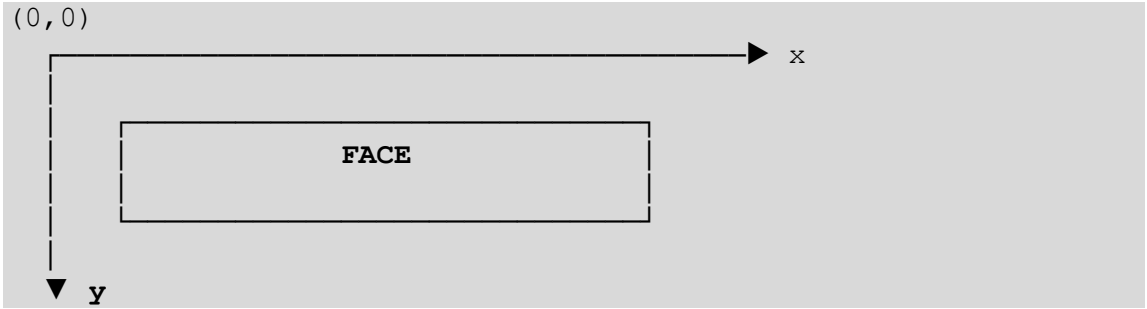
```
{
 'bbox': [x1, y1, x2, y2],
 'confidence': 0.9966,
 'landmarks': [
     [left_eye_x, left_eye_y],
     [right_eye_x, right_eye_y],
     [nose_x, nose_y],
     [mouth_left_x, mouth_left_y],
     [mouth_right_x, mouth_right_y]
 ]
}
```

The bounding box coordinates are then converted to integers:

```
x1, y1, x2, y2 = map(int, face_data["bbox"])
w, h = (x2 - x1), (y2 - y1)
```

Width and height are used to calculate the size of the bounding box, which will be used later to filter out possible noise.

1.3.4. Bounding Box Visualization



Bounding boxes are drawn for all detections, including those that may later be rejected. This allows visual inspection of both accepted and rejected detections.

```
cv2.rectangle(display_image, (x1, y1), (x2, y2), (0, 255, 0), 3)
```

The bounding boxes are used purely for visualization and debugging, allowing us to visually confirm where the face detector believes a face is located.
The second parameter, **(x1, y1)**, represents the top-left corner of the bounding box in pixel coordinates. The third parameter, **(x2, y2)**, represents the bottom-right corner of the bounding box. Together, this fully defines the rectangular region containing the face.

1.3.5. Landmark Visualization

Facial landmarks are visualized as filled circles:

```
for lm in face_data["landmarks"]:
    cv2.circle(
        display_image,
        (int(lm[0]), int(lm[1])),
        3,
        (255, 0, 0),
        -1
    )
```

As previously described, RetinaFace detects five facial landmarks: left eye, right eye, nose, left mouth corner and right mouth corner. Each **lm** represents a single landmark of a detected face and consists of a two-dimensional coordinate pair, where **lm[0]** and **lm[1]** correspond to the x and y pixel coordinates of that landmark. These values are converted to integers because OpenCV drawing functions require integer pixel coordinates.

1.3.6. Detection Filtering

1.3.6.1. Confidence-Based Filtering

To filter unreliable detections, a confidence threshold is applied:

```
CONF_THRESHOLD = 0.7
```

As previously discussed, false positive detections may occur in background regions. By enforcing a minimum confidence score of **0.7**, the pipeline prioritizes precision over recall. This reduces noise in the dataset and improves the quality of the aligned faces.

```
if score < CONF_THRESHOLD:
    continue
```

1.3.6.2. Bounding Box Size Filtering

Faces with bounding boxes smaller than 40 pixels in width or height are excluded, as previously motivated, since such detections typically lack sufficient facial detail for reliable recognition.

```
if w < 40 or h < 40:
    continue
```

1.3.7. Face Alignment

Facial alignment is performed using the detected facial landmarks to geometrically normalize the face before further processing. The landmark coordinates are first converted to floating-point NumPy arrays to enable accurate geometric transformations:

```
landmarks = np.array(face_data["landmarks"], dtype=np.float32)
```

The face is then aligned and resized to a fixed resolution of 112x112 pixels:

```
aligned_face, _ = face_alignment(image, landmarks, 112)
```

The alignment estimates a similarity transformation that maps the detected facial landmarks to a predefined landmark configuration. This transformation typically consists of:

- Translation to center the face
- Rotation to horizontally align the eyes
- Scaling to normalize face size

By aligning to fixed reference positions, variations caused by head tilt, in-plane rotation and scale differences are reduced.

Figure 4: Face Before and After Alignment

2. Embedding Creation

2.1. Goal and Context

The `CreateEmbeddings.py` script transforms previously aligned face images into numerical embeddings. Each embedding represents one face in a compact vector representation of facial identity and forms the basis for clustering and classification.

2.2. Face Embeddings

A face embedding is a numerical vector that encodes high-level facial characteristics, learned by a deep neural network. In this project, each face is represented as a 512-dimensional vector.

Although individual dimensions have no direct interpretation, the embedding as a whole acts as a numerical signature of facial identity. Embeddings belonging to the same individual are expected to lie close together in feature space, whereas embeddings of different individuals are separated by larger distances.

2.2.1. Cosine Similarity and Euclidean Distance

Face embeddings reside in a high-dimensional feature space, where distances between vectors reflect facial similarity. Two distance measures are considered in this project: cosine similarity and Euclidean distance.

Although both metrics compare embedding vectors, they capture fundamentally different geometric properties. Cosine similarity measures the angular difference between vectors, focusing on their relative orientation. Euclidean distance, in contrast, measures the straight-line distance between points in feature space and is influenced by both direction and magnitude.

Understanding how these metrics relate to each other is essential for correctly interpreting distances between face embeddings and for selecting suitable algorithms in later stages of the pipeline.

2.2.1.1. Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors:

$$\cos\vartheta = \frac{x \cdot y}{||x|| \cdot ||y||}$$

This metric evaluates how similar the directions of two vectors are, independent of their magnitude. Its theoretical range is [-1, 1], where values closer to 1 indicate a smaller angular difference.

In the context of face embeddings, this property is particularly important because identity-related information is primarily encoded in the direction of the embedding vector rather than its length. ArcFace is trained using angular margin–based loss functions that explicitly enforce small angular distances between embeddings of the same identity, while embeddings of different identities are separated by larger angular margins. As a result, cosine similarity aligns directly with the training objective of the embedding model and is well suited for measuring identity similarity.

2.2.1.2. Euclidean Distance

Euclidean distance measures the straight-line distance between two vectors:

$$||x - y||^2 = (x - y) \cdot (x - y) = ||x||^2 + ||y||^2 - 2 \cdot (x \cdot y)$$

Unlike cosine similarity, Euclidean distance is sensitive to both the direction and the magnitude of the vectors. While this property makes it suitable for many distance-based algorithms, it poses a limitation when applied directly to raw face embeddings.

Since the magnitude of face embeddings does not reliably encode identity-related information, two embeddings belonging to the same individual may exhibit a large Euclidean distance if their vector lengths differ significantly. As a result, Euclidean distance applied to unnormalized embeddings can lead to inconsistent similarity measurements that do not accurately reflect identity similarity *(Harshit Garg, 2025, fig. 5)*.

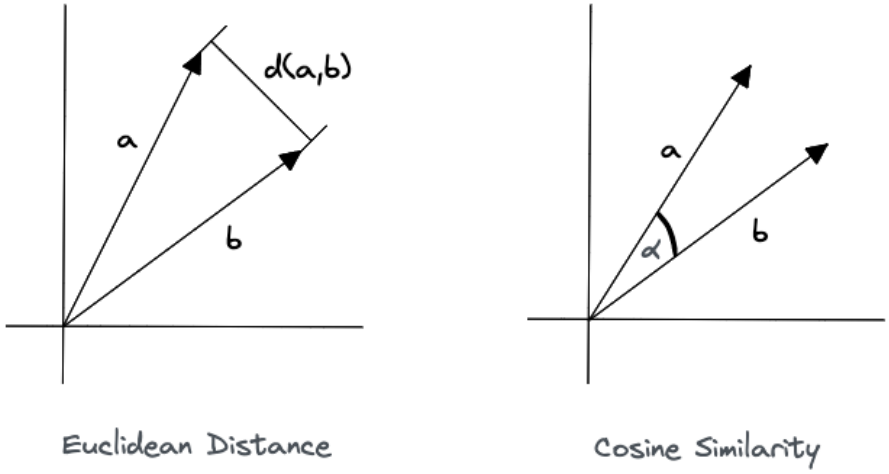


Figure 5: Euclidean Distance VS Cosine Similarity

2.2.2. Embedding Normalisation

To make Euclidean distance suitable for comparing face embeddings, all embeddings are L2-normalized prior to distance computation:

$$\text{L2-norm: } ||x||_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$
$$\text{Normalized: } x_i^{norm} = \frac{x_i}{||x||_2}$$

This normalization enforces unit length for all embeddings, constraining them to lie on the surface of a unit hypersphere. As a result, differences in vector magnitude are eliminated and comparisons between embeddings become independent of scale.

By applying this normalization step, Euclidean distance becomes dependent on the angular relationship between embeddings, making it suitable for identity-based similarity comparison *(Bi, 2022, fig. 6)*.

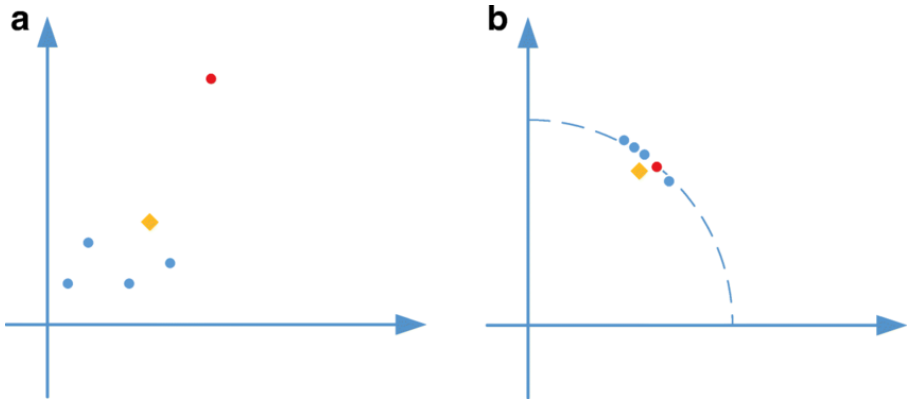


Figure 6: Before and After L2-Normalisation

2.2.3. Euclidean Distance in Function of Cosine Similarity

Euclidean Distance:

$$x, y \in \mathbb{R}$$
$$||x - y||^2$$
$$\Leftrightarrow (x - y) \cdot (x - y)$$
$$\Leftrightarrow ||x||^2 + ||y||^2 - 2(x \cdot y)$$
$$(L2\ normalisation: ||x|| = ||y|| = 1)$$
$$\Leftrightarrow 2 - 2(x \cdot y)$$
$$\Leftrightarrow 2(1 - (x \cdot y))$$
$$\Rightarrow ||x - y||^2 = 2(1 - \cos(\theta))$$

Cosine Similarity:

$$\cos(\theta) = \frac{x \cdot y}{||x||||y||}$$
$$(L2\ normalisation: ||x|| = ||y|| = 1)$$
$$\Leftrightarrow cos(\theta) = x \cdot y$$

This result shows that, for L2-normalized embeddings, Euclidean distance is a monotonic function of cosine similarity. An increase in cosine similarity therefore implies a decrease in Euclidean distance, as vectors pointing in similar directions move closer together on the unit hypersphere. Consequently, distance-based algorithms operating in Euclidean space, such as KMeans clustering, can be applied without modifying the underlying notion of identity similarity encoded by the embeddings.

Although cosine similarity is well suited for measuring identity similarity, this equivalence justifies the use of Euclidean distance–based algorithms, such as KMeans clustering, without altering the underlying notion of identity similarity.

2.3. Embedding Model

```
embedder = ArcFace(model_name=ArcFaceWeights.RESNET)
```

ArcFace is a deep learning model specifically designed for face recognition. Rather than directly predicting an identity label, the model maps each face to a high-dimensional embedding.

file - str	dim_0 - f32	dim_1 - f32	dim_2 - f32	dim_3 - f32	dim_4 - f32	dim_5 - f32	dim_6 - f32	dim_7 - f32	dim_8 - f32	dim_9 - f32
aligned_173_2.png	-0.389685	0.056165	0.059096	0.448501	0.884615	0.843353	-0.153908	0.059364	-2.123758	1.169462
aligned_174_0.png	-0.118827	-0.635897	-0.174014	-0.919134	1.628118	2.369373	-0.956576	-0.562717	-1.010049	0.433288
aligned_174_1.png	-0.594739	-0.74059	0.469664	-0.050605	-0.47456	-0.338091	0.14361	-0.316783	0.231605	-0.078518
aligned_174_2.png	-0.231136	0.33681	0.245477	-0.109478	-0.651773	-0.11963	0.107572	0.967912	-0.095592	0.003295
aligned_176_0.png	-0.443859	-0.852765	0.981592	-1.748522	-0.974531	0.450586	-0.9853	-1.220072	0.538449	-0.659764
aligned_176_1.png	-0.094066	-0.909573	-0.00995	0.054121	1.373206	1.972945	-1.177344	-0.204156	-0.644039	0.319068
aligned_176_2.png	0.600803	-0.83837	-0.556346	-0.14592	0.569438	1.16088	-1.710269	0.669511	-2.260967	-0.433735
aligned_176_3.png	-0.837301	-0.914874	0.800649	1.024178	0.286846	-0.869441	0.884502	0.240719	0.310785	-1.169966
aligned_176_4.png	-0.170135	0.277394	-0.438745	-0.574953	-0.040937	1.085695	0.294123	0.539109	0.212079	0.228011
aligned_177_0.png	-0.020619	-1.869232	-0.032261	-0.033065	0.654347	1.081709	-1.319039	-0.086141	1.755422	-1.945658
aligned_177_1.png	-0.67725	-0.525605	0.20921	0.019573	-0.17852	-0.354811	-0.103648	-0.284347	-0.295475	-0.028364
aligned_178_0.png	0.103387	0.652821	1.190625	-1.552893	0.001761	0.066409	-0.45109	0.31036	0.40814	0.515665
aligned_178_1.png	-0.472277	-0.282584	1.030866	1.488105	0.085273	0.865899	-1.164108	-1.206711	-1.431118	0.88107
aligned_178_2.png	-0.31541	0.438166	-0.125828	-0.675908	0.558578	0.709581	0.108235	-0.499688	-1.172749	-0.36309
aligned_179_0.png	0.311427	-0.178876	0.420477	-0.082194	0.312102	0.116759	-0.092539	-0.106365	-1.605049	-0.001189
aligned_179_1.png	0.885466	2.145618	0.178391	1.259797	-0.286273	-1.086988	-1.048159	0.278163	-0.096976	0.822569
aligned_179_2.png	-0.901575	-0.971618	0.615389	-0.204196	-0.071888	-1.140344	0.1817	-0.37461	-0.609439	0.108829
aligned_17_0.png	-2.007159	-0.599896	-1.916903	0.718742	0.119534	-0.547493	-0.297516	1.366971	0.464764	1.472492
aligned_180_0.png	0.526655	0.148281	1.977968	-1.227337	1.819048	1.297503	0.283347	0.788869	1.129194	-0.901122

Figure 7: Face Embeddings

2.4. Embedding Generation Workflow

The embedding generation process iterates over all aligned face images produced in the previous preprocessing stage.

2.4.1. Data Structures

Two parallel lists are used to store the results:

```
embeddings = []
filenames = []
```

The **embeddings list** stores the 512-dimensional vectors generated by the ArcFace model, while the **filenames list** stores the corresponding image filenames. This ensures that every embedding can later be traced back to its original face image.

2.4.2. Image Validation

Before generating an embedding, the input images are validated to check if the images have the expected ArcFace input requirements:

```
if img.shape[0] != 112 or img.shape[1] != 112:
    continue
```

This is confirmed by the debug output printed during execution:

```
File: aligned_9_0.png
shape: (112, 112, 3)
dtype: uint8
```

2.4.3. Embedding Vector Generation

The embedding vector is generated by forwarding the image through the ArcFace network:

```
emb = embedder.get_embedding(img_rgb).squeeze()
```

ArcFace uses a convolutional backbone to extract identity-related facial features and projects them into a 512-dimensional embedding space. The output of the `get_embedding` function is a tensor of shape (1, 512), where the singleton batch dimension is removed using `squeeze()`, resulting in a one-dimensional vector of shape (512,).

This is verified through debug output:

```
Embedding shape: (512,)
Embedding sample: [-0.6200673  -0.40661862 -1.3557601  ...
-0.09619124 -0.9543795]
```

The embedding consists of 512 floating-point values. These values can be positive or negative and are typically centered around zero. While individual values are not interpretable on their own, the full vector captures the identity-related structure of the face.

As a result, different images of the same person are expected to produce similar embedding vectors, while images of different people should produce clearly distinct vectors.

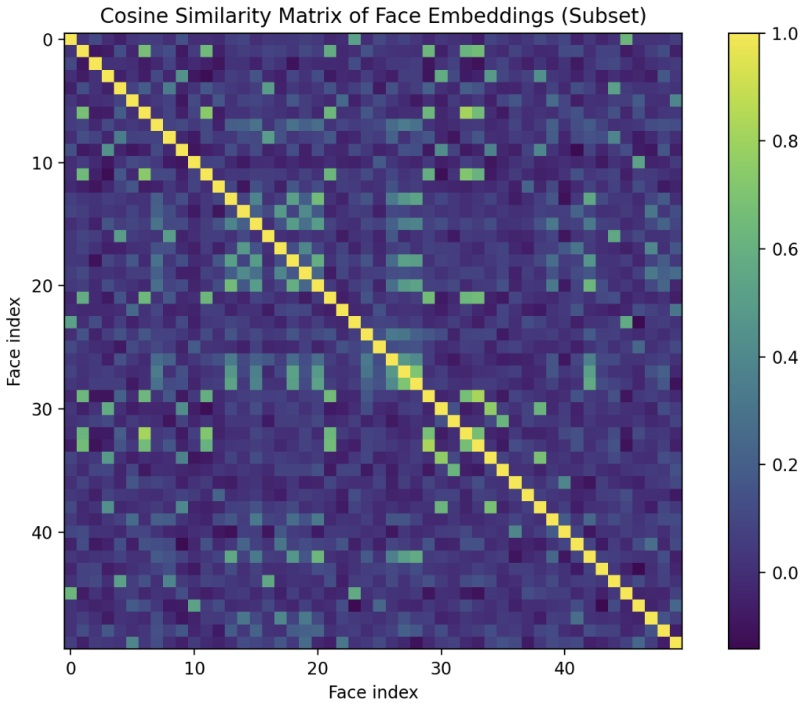


Figure 8: Cosine-Similarity Matrix of Face Embeddings

To analyse whether the generated embeddings capture identity-related information, cosine similarity was computed between pairs of face embeddings. *Figure 8* shows the resulting cosine similarity matrix for a subset of the dataset.

The diagonal line corresponds to self-similarity, where each embedding is compared with itself and therefore has a similarity of 1.0. Off-diagonal elements represent pairwise similarities between different face images. Higher similarity values indicate faces that are more alike, while lower values correspond to faces that are more dissimilar.

The observed structure in the similarity matrix indicates that the embedding space is not random but encodes meaningful relationships between faces. In particular, localized regions of higher similarity suggest that images with similar facial characteristics, potentially belonging to the same individual, are embedded closer together than unrelated faces. This confirms that the embedding space captures meaningful identity structure and motivates the use of clustering methods in the next chapter.

3. Selection of K for Face Clustering

3.1. Goal and Context

In the previous parts, faces were first detected and aligned, after which each face was transformed into a numerical embedding. These embeddings reside in a high-dimensional feature space in which faces belonging to the same individual are expected to be located close to one another, while faces of different individuals are separated by larger distances.

The purpose of `GlobalK.py` is to determine the appropriate number of clusters (k) for grouping face embeddings using K-means. Multiple values of k are evaluated using both elbow plots, based on numerical quality measures (distortion and inertia) as well as PCA scatter plots that allow for visual inspection of the resulting cluster structures.

Figure 9 shows the PCA projection of all face embeddings without clustering. Several dense regions are visible, indicating that the embedding space already contains structure prior to clustering. This observation supports the assumption that identity-related information is present in the embeddings, generated by ArcFace.

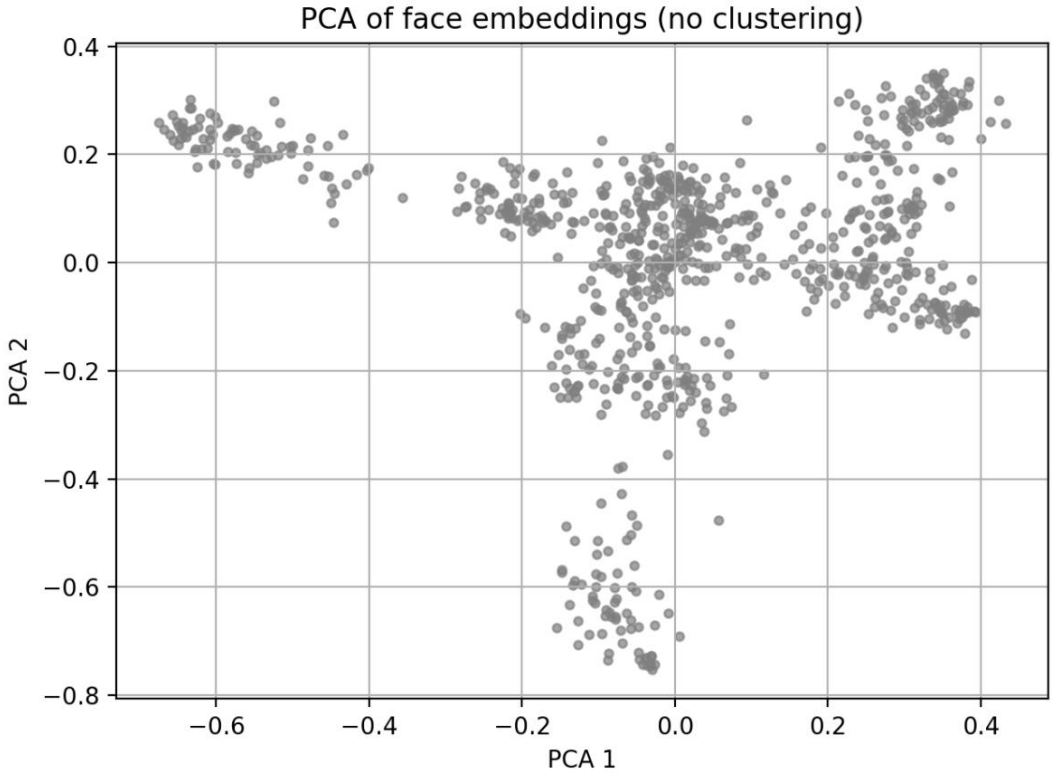


Figure 9: Face Embeddings PCA

3.2. Selection Workflow

3.2.1. K Range Selection

The script evaluates values of k in the following range:

```
K_range = range(5, 16)
```

The images used in this project originate from a classroom environment with twelve known individuals. In theory, this implies that the face embeddings should form approximately twelve distinct clusters, each representing a single person. In practice, however, not all detected faces necessarily belong to these known individuals. Some images may include people outside the class, partial faces, or incorrectly detected faces. As a result, an additional cluster is expected to capture embeddings that do not clearly correspond to any of the known individuals, such as faces of external persons, partial detections or ambiguous samples.

For this reason, the expected number of clusters is thirteen: twelve clusters corresponding to the known individuals and one additional cluster representing unknown faces. The selected range of k values from five to fifteen includes this expected value, while still allowing the behaviour of the clustering to be observed for smaller and larger values of k.

3.2.2. KMeans Clustering

For each value of k, a KMeans model is trained on the set of normalized face embeddings:

```
kmeans = KMeans(n_clusters=k, random_state=42, n_init="auto")
kmeans.fit(embeddings)
```

KMeans is an unsupervised clustering algorithm that divides the embedding space into k clusters by minimizing the sum of squared Euclidean distances. Each cluster is represented by a centroid, defined as the mean of all embeddings assigned to that cluster. During training, KMeans iteratively alternates between assigning embeddings to the nearest centroid, based on Euclidean distance, and updating centroid positions until convergence is achieved.

Because the embeddings are L2-normalized, Euclidean distance between embeddings and centroids is directly related to angular similarity. As a result, clustering is effectively performed based on the directional similarity of embeddings, which aligns with the angular structure learned by the ArcFace model.

A fixed `random_state` is used to ensure reproducibility of the clustering results. The parameter `n_init="auto"` instructs KMeans to perform multiple initializations with different centroid seeds and to select the solution with the lowest final inertia. Without this, the clustering results varied when running the script multiple times.

3.2.3. Selection of the Number of Clusters (k)

3.2.3.1. Distortion Analysis

Distortion is defined as the mean squared Euclidean distance between each embedding and its closest cluster centroid:

```
distortion = (
    np.min(cdist(embeddings, kmeans.cluster_centers_,
"euclidean"), axis=1) ** 2
).mean()
```

First, the Euclidean distances between all embeddings and the cluster centroids are computed. For each embedding, only the smallest distance is kept, corresponding to the cluster it belongs to. This distance is squared to give more weight to embeddings that lie further away from their respective cluster centroids.

The resulting distortion value represents the average compactness of the cluster. Lower distortion values indicate more compact clusters, where embeddings are located closer to their assigned centroids.

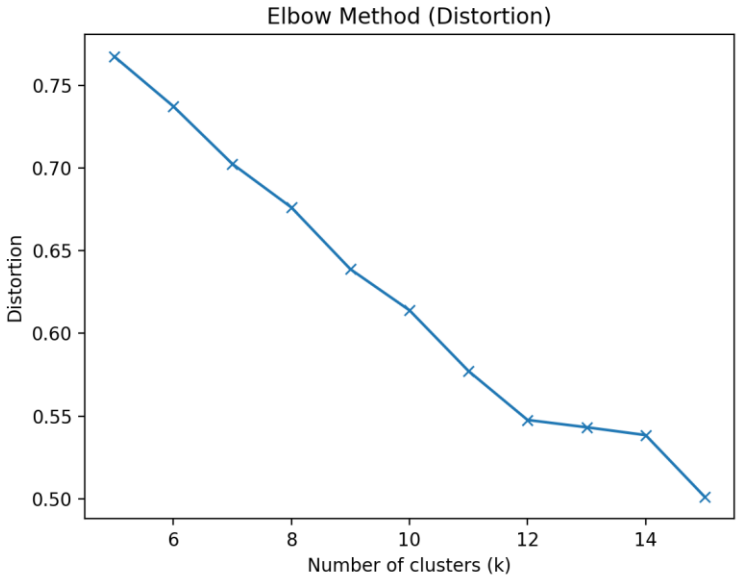


Figure 10: Elbow (distortion)

Figure 10 illustrates the distortion values for different numbers of clusters. There is a decrease in distortion as the number of clusters increases from *k* = 5 to approximately k = 12, indicating substantial improvements in cluster compactness within this range. Beyond this point, the rate of decrease slows, with only smaller reductions between k = 12 and k = 14.

At k = 13, the distortion curve has stabilized, suggesting that the main structure of the embedding space is captured. Increasing the number of clusters beyond this value would introduce additional model complexity without yielding meaningful improvements in clustering quality.

3.2.3.2. Inertia Analysis

In addition to distortion, the inertia value from KMeans is also stored:

```
inertias.append(kmeans.inertia_)
```

Inertia represents the total sum of squared Euclidean distances between all embeddings and their assigned cluster centroids. Although inertia and distortion are closely related, storing both metrics makes it easier to confirm that the observed trends are consistent and not caused by a single metric.

A similar plot like the distortion plot is created for inertia:

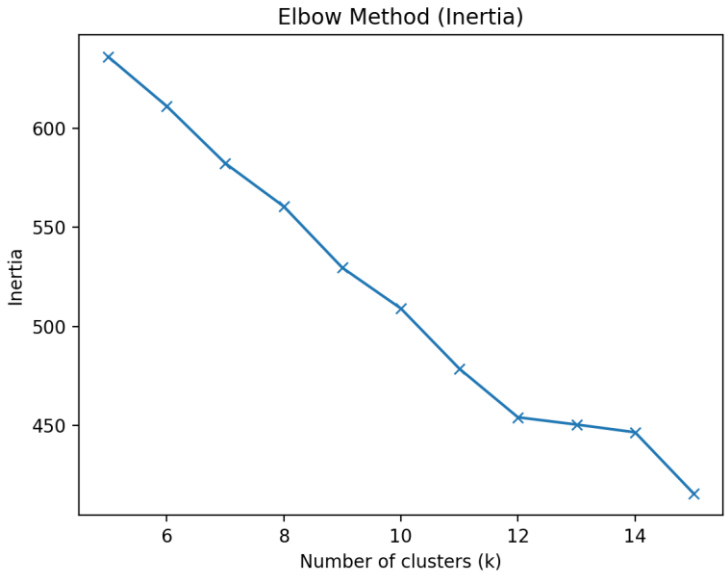


Figure 11: Elbow (inertia)

Figure 11 shows the inertia values for different numbers of clusters. Similar to the distortion analysis, a strong decrease in inertia is observed as the number of clusters increases from k = 5 to k = 12. This indicates substantial improvements in within-cluster cohesion within this range, as embeddings are increasingly grouped closer to their respective centroids.

This behaviour confirms the conclusions drawn from the distortion analysis.

3.2.4. PCA Projection of the Embedding Space

To better understand how the clusters are distributed, the normalized embeddings are projected onto two dimensions using Principal Component Analysis (PCA):

```
pca = PCA(n_components=2, random_state=42)
emb2d = pca.fit_transform(embeddings)
```

PCA reduces the high-dimensional embedding space to two dimensions so the data can be visualized. This transformation is used only for visualization.

For each value of k, the clustering result is visualized in PCA space. Each point represents a face and the colour indicates the cluster assignment.

3.2.4.1. Low Number of Clusters (k = 5)

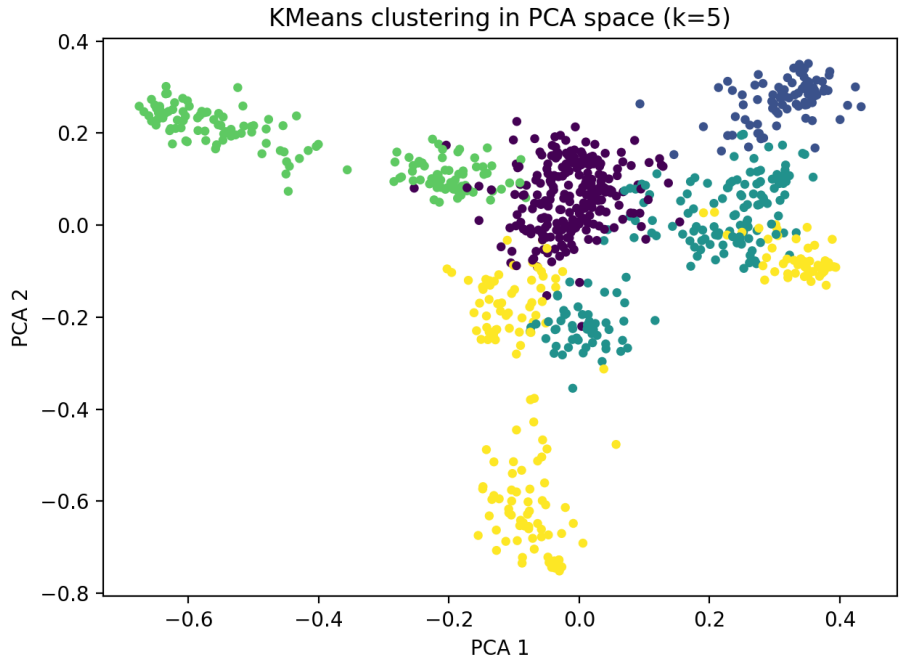


Figure 12: Clustering (k = 5)

At this low number of clusters, the resulting clusters are relatively broad and span multiple visually distinct regions in the embedding space.

Several clusters cover areas that appear to contain multiple dense substructures, suggesting that embeddings with different facial characteristics are grouped together within the same cluster.

As a result, the clustering fails to capture the structure of the embedding space, leading to clusters that likely represent multiple individuals rather than a single coherent identity. This observation highlights the limitation of using a low k value and motivates the exploration of higher numbers of clusters.

3.2.4.2. Intermediate Number of Clusters (k = 9)

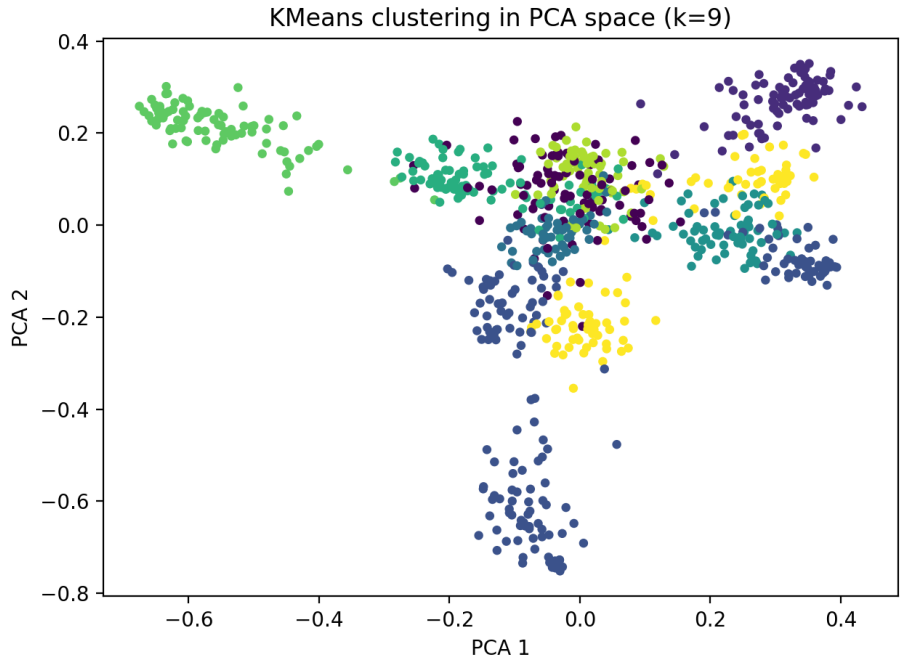


Figure 13: Clustering (k = 9)

Compared to the case of k = 5, the clusters become more clearly separated and several regions, that were previously grouped together, are now split into distinct clusters.

The clustering at k = 9 reveals a more refined structure in the embedding space, with clusters that are more compact and better aligned with visually separable regions in the PCA projection. This indicates a reduction in underclustering, as embeddings with different facial characteristics are less frequently grouped into the same cluster.

However, some overlap between clusters remains visible, suggesting that certain clusters may still contain multiple identities or that the separation between similar identities is not yet fully resolved.

3.2.4.3. Higher Number of clusters (k = 13 and k = 15)

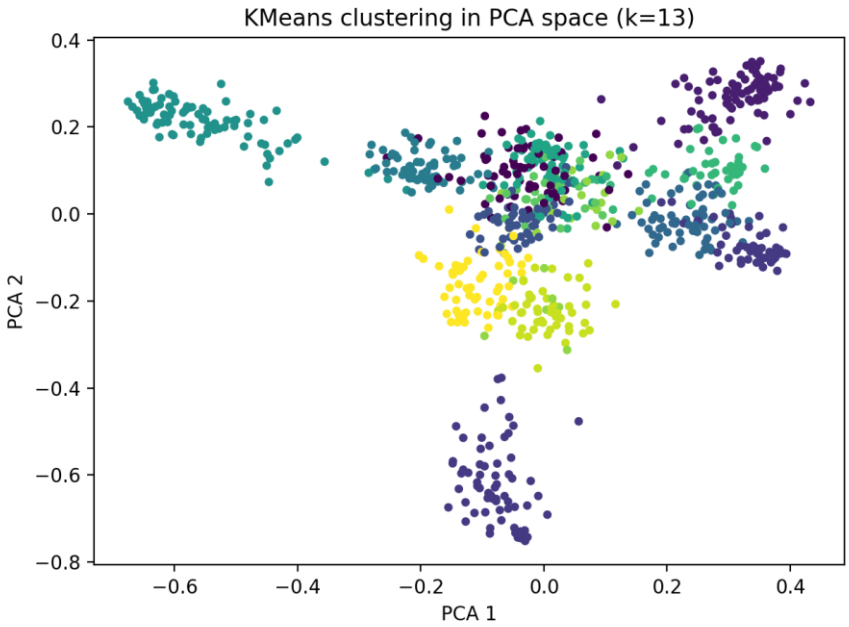


Figure 14: Clustering (k = 13)

Figure 14 illustrates the KMeans clustering result in the two-dimensional PCA projection for k = 13. At this value, clusters appear more compact and more clearly separated compared to lower values of k. The majority of dense regions visible in the PCA space are represented by distinct clusters, indicating that the clustering captures a substantial portion of the structure present in the embedding space.

The clustering at k = 13 aligns well with the trends observed in both the distortion and inertia elbow analyses, where diminishing returns were observed beyond approximately k = 12. This suggests that k = 13 represents a suitable balance between cluster compactness and model complexity, making it a reasonable choice for the final clustering configuration.

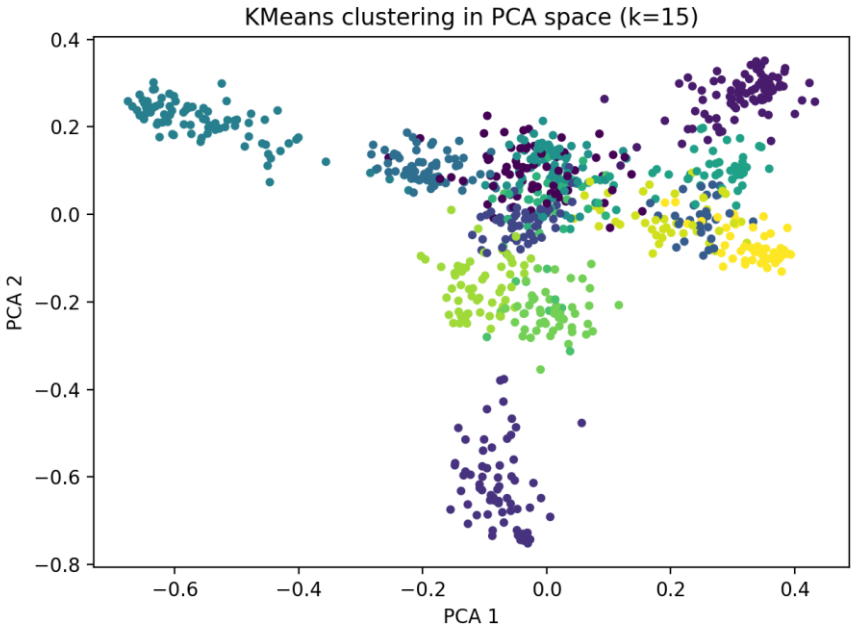


Figure 15: Clustering (k = 15)

Figure 15 shows the clustering result for k = 15. While some clusters are further subdivided, this additional splitting does not introduce clearly new, well-separated regions in the PCA projection. Instead, several previously coherent clusters are partitioned into smaller subclusters.

Although this finer granularity may capture variations related to factors such as pose, illumination or facial expression, it increases the risk of overclustering, where embeddings belonging to the same individual are divided across multiple clusters. Consequently, increasing the number of clusters beyond k = 13 appears to provide limited benefits in terms of identity separation while increasing the likelihood of fragmenting existing clusters.

4. Face Clustering

4.1. Goal and Context

Instead of manually labelling all detected face images, unsupervised clustering is used to group face embeddings into identity-based folders. In the previous chapter, an appropriate range for the number of clusters was identified using elbow analysis and PCA-based visualization. Based on this analysis, a value of k = 13 was selected. `Clustering.py` takes the 512-dimensional face embeddings generated earlier (stored in `faceEmbeddings.parquet`) and applies KMeans clustering to assign each embedding to a cluster.

Each cluster is treated as a “person group”, meaning that faces belonging to the same individual are expected to be grouped together.

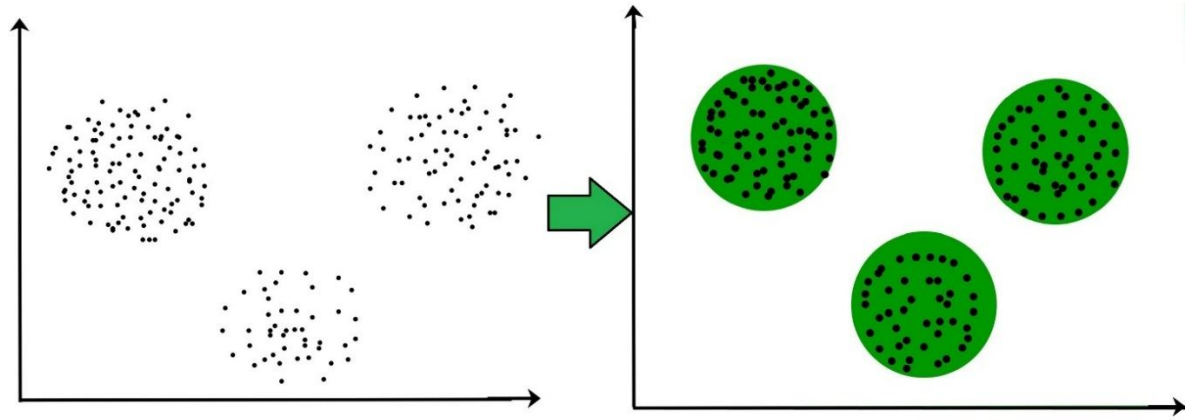


Figure 16: Clustering Example

4.2. KMeans Clustering Workflow

KMeans clustering is applied using k = 13:

```
kmeans = KMeans(n_clusters=K, n_init=20, random_state=42)
labels = kmeans.fit_predict(embeddings)
```

Figure 17 visualizes the result of KMeans clustering in a two-dimensional PCA projection of the face embeddings. Each point represents a single face embedding projected onto the first two principal components, while the coloured groups indicate the cluster assignments produced by KMeans. The cross-shaped markers denote the cluster centroids, which correspond to the mean position of all embeddings assigned to each cluster.

Although the clustering itself is performed in the original 512-dimensional embedding space, the PCA projection provides an interpretable low-dimensional visualization of the cluster structure. In this projection, embeddings belonging to the same cluster form compact groups around their corresponding centroid, indicating low within-cluster variance.

KMeans assigns each embedding to the nearest centroid based on Euclidean distance. The result **labels** is a 1D array where each element is the cluster index (0 to K−1) for a face, in the same order as the **files** list.

Figure 17: KMeans Clustering with Centroids

4.3. Cluster Folders

After clustering, each face image is assigned to a directory corresponding to its predicted cluster label. This is achieved by iterating simultaneously over the list of face image filenames and the array of cluster labels produced by the KMeans algorithm:

```
for file, label in tqdm(zip(files, labels), total=len(files),
desc="Assigning clusters"):
    dst_dir = os.path.join(CLUSTER_DIR, f"person_{label}")
    os.makedirs(dst_dir, exist_ok=True)

    src = os.path.join(CROPPED_DIR, file)
    dst = os.path.join(dst_dir, file)

    if os.path.exists(src):
        shutil.copy(src, dst)
```

During this process, each filename is paired with its corresponding cluster label, ensuring a one-to-one mapping between a face image and its assigned cluster. For each unique cluster label, a dedicated directory is created following the naming convention **./clusters/person_<label>**. This directory represents a single cluster identified by the KMeans model.

The **source** path refers to the aligned face image stored in the **cropped_faces** directory, while the **destination** path points to the appropriate cluster directory. Each image is then copied into its corresponding folder, effectively grouping all face images assigned to the same cluster into a single location.

5. Silhouette Analysis of Face Clusters

5.1. Goal and Context

After clustering the face embeddings and organizing the results into identity-based folders, it is necessary to evaluate the quality of the obtained clusters. While visual inspection of cluster contents provides intuitive insight, a quantitative evaluation is required to evaluate how well individual face embeddings fit within their assigned clusters.

In `silhouetteScore.py`, clustering quality is evaluated using silhouette analysis. The silhouette score measures how similar a face embedding is to other embeddings within the same cluster compared to embeddings in neighbouring clusters. This allows both per-cluster and per-sample evaluation of clustering performance.

5.2. Silhouette Score Workflow

5.2.1. Silhouette Score Computation

Before computing silhouette scores, a validation step is performed to ensure that the clustering result contains at least two distinct clusters:

```
if len(unique_labels) < 2:
    raise ValueError("Not enough clusters for silhouette
scoring.")
```

Silhouette scoring requires a minimum of two clusters, as the metric is defined by comparing the distance of a sample to its own cluster with the distance to the nearest neighbouring cluster. When only a single cluster is present, no meaningful comparison can be made and the silhouette score is undefined.

Silhouette scores are computed for every individual face embedding:

```
scores = silhouette_samples(embeddings, labels)
```

The resulting **scores** array is one-dimensional, where each element corresponds to a single face embedding in the dataset. Each score quantifies how well that embedding fits within its assigned cluster relative to other clusters in the embedding space.

5.2.2. Interpretation of Silhouette Values

The silhouette score for an individual sample lies within the range [-1, +1] and is interpreted as follows:

- **Values close to +1** indicate that the face embedding is well matched to its own cluster and clearly separated from neighbouring clusters.
- **Values around 0** suggest that the embedding lies near a cluster boundary, indicating an uncertain assignment.
- **Negative values** indicate that the embedding is closer to another cluster than to its assigned cluster.

5.3. Silhouette Scores

5.3.1. Silhouette Score per Cluster

Figure 18 shows the average silhouette score for each cluster along with an additional unknown cluster. The silhouette score measures how well the faces within a cluster fit together relative to faces in other clusters.

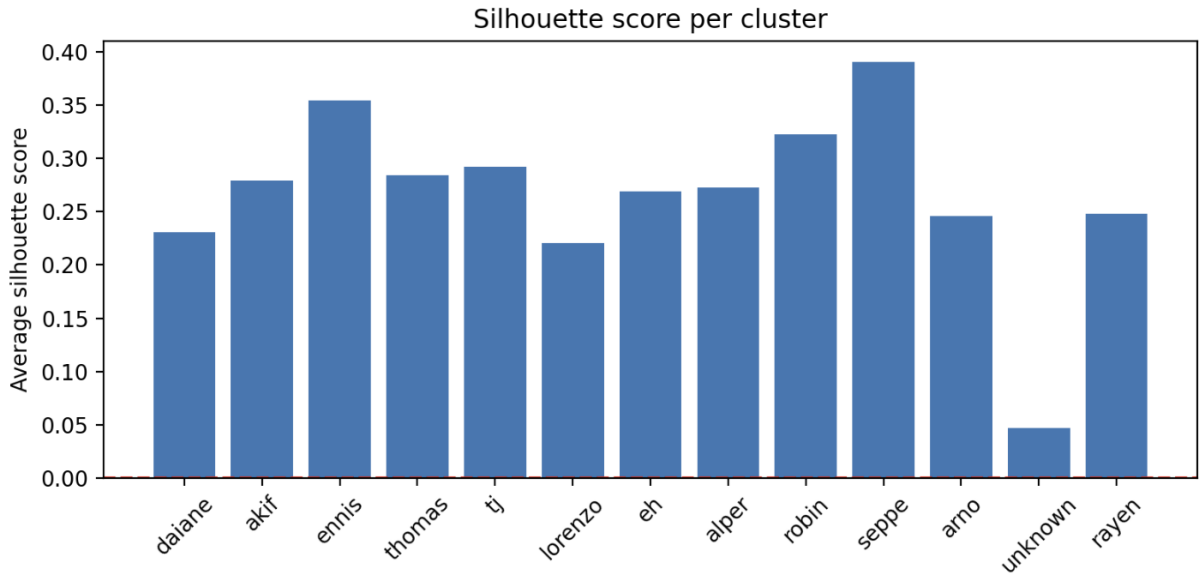


Figure 18: Silhouette Score per Cluster

Most clusters have an average silhouette score between **0.20 and 0.40**. This indicates that, for these identities, the majority of faces are closer to their own cluster than to neighbouring clusters. Clusters with higher values, such as **Seppe**, **Ennis** and **Robin**, are more compact and better separated. This suggests that these identities are represented by consistently similar embeddings.

Some clusters have noticeably lower average scores. In particular, the cluster interpreted as the “unknown” cluster has a silhouette score close to zero. This cluster is not explicitly enforced during clustering, but emerges naturally as a group of embeddings that do not fit well into the main identity clusters.

Clusters with moderately lower scores, such as **Lorenzo** or **Daiane**, may contain more variation in pose, lighting or expression, which slightly reduces cluster cohesion but does not indicate severe misclustering.

Overall, the silhouette score per cluster provides evidence that most identity clusters are reasonably compact and well separated, while also highlighting clusters that are more ambiguous or internally diverse and may warrant closer inspection.

5.3.2. Silhouette Diagrams for Different Values of *k*

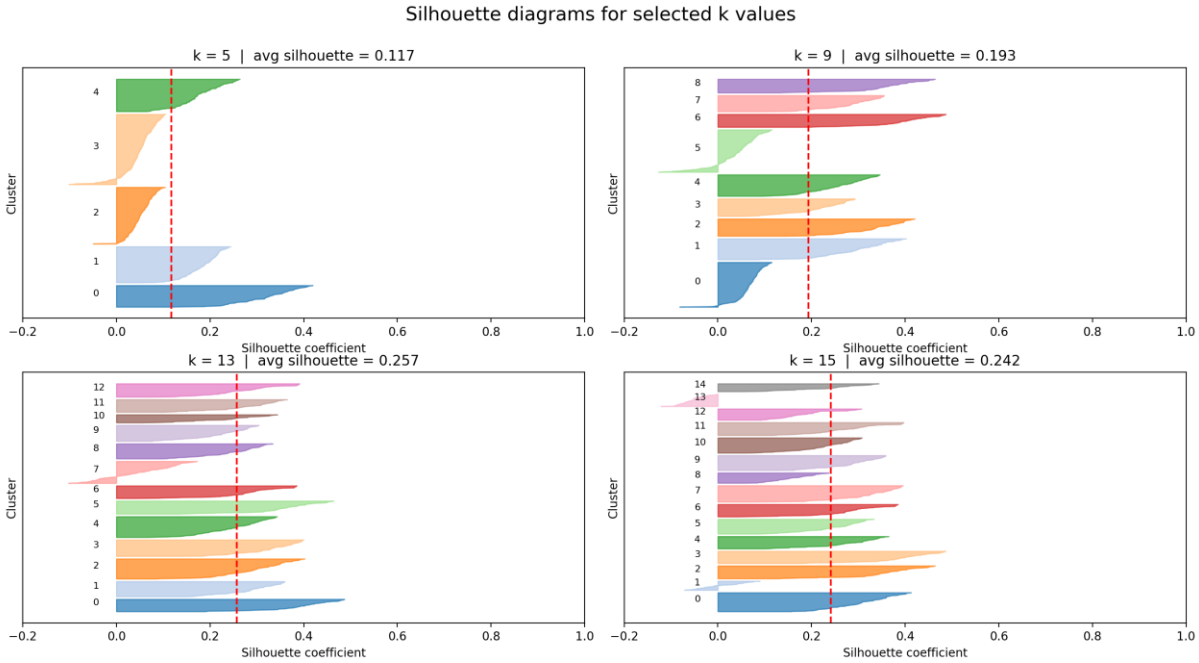


Figure 19: Silhouette Diagrams

Figure 19 presents silhouette diagrams for selected values of k (k = 5, 9, 13 and 15). Each diagram visualizes the distribution of silhouette scores within each cluster, while the dashed vertical line indicates the average silhouette score across all samples for the corresponding k value.

For k = 5, the average silhouette score is relatively low (0.117). Several clusters show wide internal variation and contain a significant number of samples with silhouette scores close to zero or slightly negative. This indicates there might be an overlap between clusters and supports the observation of underclustering, where multiple identities are grouped together within the same cluster.

The silhouette diagram for k = 13 shows the highest average silhouette score of the evaluated values (0.257). Most clusters show consistently positive silhouette scores with relatively compact internal distributions. Only samples in cluster 7 have silhouette scores close to zero, suggesting that this cluster may correspond to the unknown cluster, where faces are less clearly separated from each other. On the other hand, faces in the other clusters are clearly closer to their assigned cluster than to neighbouring clusters. Overall, this configuration reflects a good balance between cluster separation and compactness, which is consistent with the conclusions drawn from the analyses based on distortion, inertia and PCA.

For k = 15, the average silhouette score decreases slightly to 0.242. Some clusters remain well separated, while others are split into smaller subclusters. This behaviour indicates the start of overclustering, where additional clusters mainly subdivide existing identity clusters rather than revealing new, well-separated clusters.

Overall, the silhouette diagrams provide a detailed, cluster-level perspective on clustering quality and reinforce the selection of k = 13 as the most appropriate number of clusters. This value achieves the highest overall silhouette score while maintaining stable and well-separated clusters, supporting its use in the final clustering configuration.

6. Model Training

6.1. Goal and Context

The goal of the `TrainSaveModel.py` script is to convert the previously obtained face clusters into supervised classification models that can be used for identity recognition during inference. At this stage of the processing pipeline, all face images have already undergone extensive preprocessing. Each face has been detected and geometrically aligned, transformed into a 512-dimensional embedding using the ArcFace model and assigned to a cluster corresponding to a single identity.

This script trains multiple classification models on top of the embeddings, allowing identity prediction for new, unseen faces (of the known identities).

6.2. Training Workflow

6.2.1. Data Loading and Preparation

The training process starts by loading the clustered face images and preparing the feature and label arrays:

```
X = []            # embeddings
y = []            # numeric labels
label_names = [] # index -> person name
label_map = {}     # person name -> numeric label
label_counter = 0
```

Here, **X** will store the input feature vectors (ArcFace embeddings), while **y** will contain the corresponding numeric class labels. The list **label_names** preserves the mapping between numeric labels and human-readable identity names.

The clustered dataset is organized in a directory structure where each subfolder represents one identity. The script iterates over these folders, assigns a unique numeric label to each identity and stores this mapping:

```
for person in sorted(os.listdir(CLUSTER_DIR)):
    if person in JUNK_LABELS:
        continue

    label_map[person] = label_counter
    label_names.append(person)
    label_counter += 1
```

Example:

```
label_map = {
    "person_0": 0,
    "person_1": 1,
    "person_2": 2,
   ...
}
```

The numeric label is stored in the target array **y**, which is used to train the classifiers.

```
y.append(label_map[person])
```

After training, the model will output numeric predictions. These numbers can be translated back into human-readable person identifiers using the **label_names** list, where the index corresponds to the class label.

```
predicted_person = label_names[predicted_label]
```

6.3. Centroid-Based Classifier

The first classification model is a centroid-based approach. This approach represents each identity by a single vector, referred to as the class centroid, which summarizes the embeddings belonging to that identity.

For each identity, all embeddings belonging to that class are selected:

```
class_embeddings = X[y == class_id]
```

The centroid of a class is computed as the mean embedding across all embeddings belonging to that class:

```
center = np.mean(class_embeddings, axis=0)
```

This centroid represents the average facial representation of the person in embedding space.

During inference, a new face embedding is compared to the centroid of each class using cosine similarity. The predicted identity corresponds to the class whose centroid is most similar to the given embedding (Adcock, fig. 18).

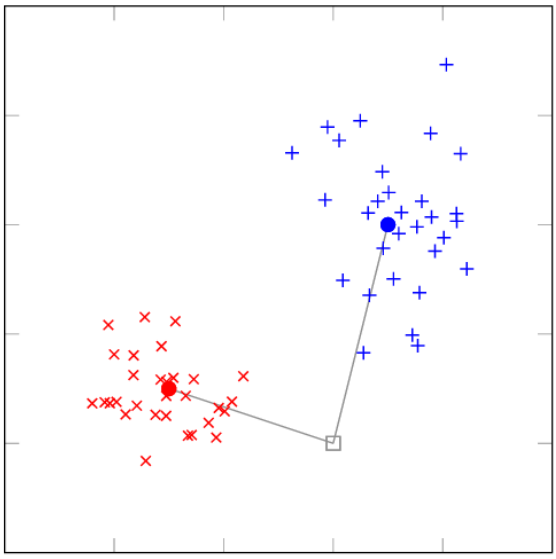


Figure 20: Centroid-Based Classifier

6.4. k-Nearest Neighbours Classifier

The second classification model used in this project is the k-Nearest Neighbours (k-NN) classifier, which operates directly on the set of face embeddings without explicitly learning a parametric model.

```
knn = KNeighborsClassifier(
    n_neighbors=3,
    metric="cosine"
)
knn.fit(X, y)
```

Given a new embedding, the classifier computes the cosine distance to every training embedding and selects the k = 3 nearest neighbours, defined as the embeddings with the smallest cosine distance. These nearest neighbours represent the most similar facial representations in the embedding space.

The predicted identity of the new face is then determined using majority voting over the labels of the selected neighbours. Specifically, the identity that occurs most frequently among the k nearest neighbours is assigned to the new embedding (Nami, z.d., fig. 21).

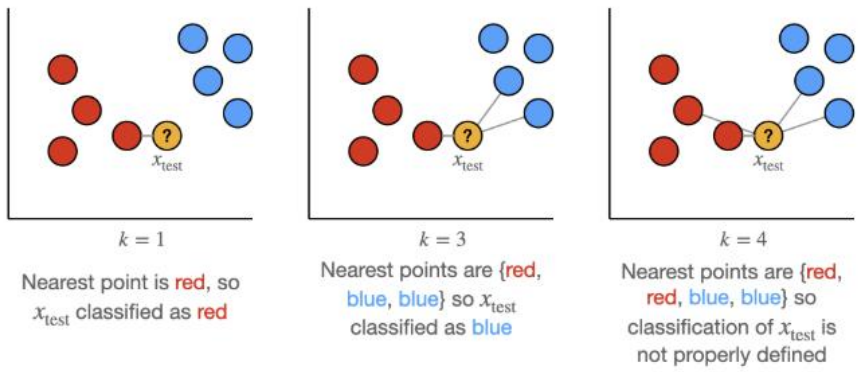


Figure 21: KNN

6.5. Support Vector Machine (SVM)

The third classification model used in this project is a Support Vector Machine (SVM) with a linear kernel. The SVM is a discriminative classifier that learns decision boundaries between classes in the embedding space.

The SVM aims to separate classes by learning a hyperplane that maximizes the margin between samples of different identities.

```
svm = SVC(
    kernel="linear",
    probability=True,
    random_state=42
)
```

For a given embedding, the classifier evaluates its position relative to these learned hyperplanes. Each hyperplane corresponds to a specific class and is defined such that embeddings belonging to that class lie on one side of the boundary, while embeddings of other classes lie on the opposite side. The decision is based on the signed distance of the embedding to each hyperplane, which reflects the confidence of the classification (Ashish Mehta, 2023, fig. 22).

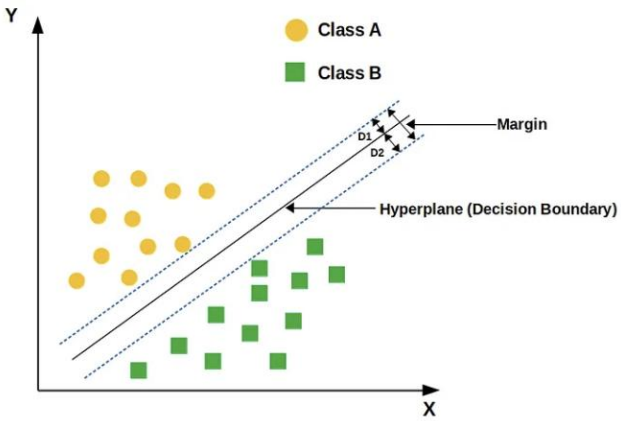


Figure 22: SVM

6.6. Logistic Regression

The final classification model evaluated in this project is Logistic Regression, which is trained as a probabilistic classifier operating directly on the face embedding space.

```
logreg = LogisticRegression(
    max_iter=2000,
    random_state=42
)
```

Logistic Regression models the conditional probability of an embedding belonging to each identity class by learning a linear decision function in the embedding space. For a given input embedding, the model computes a weighted sum of the embedding features, followed by a logistic function to produce class probabilities.

During inference, a query face embedding is passed through the learned linear model to compute a probability distribution over all identity classes. The predicted identity corresponds to the class with the highest estimated probability.

6.7. Result

Submission and Description	Private Score	Public Score	Selected
submission_logistic.csv Complete · 19d ago	0.26666	0.15625	☐
submission_svm.csv Complete · 19d ago	0.26666	0.15625	☐
submission_knn.csv Complete · 19d ago	0.26666	0.15625	☐
submission_centroid.csv Complete · 19d ago	0.26666	0.15625	☐

Figure 23: Model Score

The evaluation results show that all models achieved identical scores on both the public and private test sets. This suggests that the performance is mostly determined by the quality of the face embeddings rather than by the choice of classification algorithm or by the limited size of the training set.

Bibliography

i. Adcock, Jeremy & Allen, Euan & Day, Matthew & Frick, Stefan & Hinchliff, Janna & Johnson, Mack & Morley-Short, Sam & Pallister, Sam & Price, Alasdair & Stanisic, Stasja. (2015). Advances in quantum machine learning.

ii. Allohvk. (2025, 27 mei). *Cosine Distance vs Dot Product vs Euclidean in vector similarity search*. Medium.com. https://medium.com/data-science-collective/cosine-distance-vs-dot-product-vs-euclidean-in-vector-similarity-search-227a6db32edb

iii. Amos Stailey-Young. (2024, 9 juni). *What's the Best Face Detector?* Medium.cm. https://medium.com/pythons-gurus/what-is-the-best-face-detector-ab650d8c1225

iv. Ashish Mehta. (2023, 11 maart). *Support Vector Machine (SVM) Algorithm For Machine Learning*. Medium.com. https://ashish-mehta.medium.com/support-vector-machine-svm-algorithm-for-machine-learning-350fe9139a52

v. Aurélien Géron. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* (3rd edition). O'Reilly Media, In!c.

vi. Bi. (2022). *Critical direction projection networks for few-shot learning. Applied Intelligence*. researchgate.net.

vii. Harshit Garg. (2025, 9 juli). *The Hidden Retrieval Mistake in GenAI: Cosine Similarity vs Euclidean Distance*. medium.com. https://medium.com/@hgarg97/the-hidden-retrieval-mistake-in-genai-cosine-similarity-vs-euclidean-distance-57313f46f393

viii. Nami, Y. (z.d.). *Why does increasing K decrease variance in KNN?* Towards Data Science. https://towardsdatascience.com/why-does-increasing-k-decrease-variance-in-knn-9ed6de2f5061/