Facial Recognition Using PCA and Logistic Regression Classification + FaceNet

STA 141C w/ Prof. Q. Jiang

Edward Kim, Priyanshi Singh, Murabet Sarsur, Sanjay Manivasagam

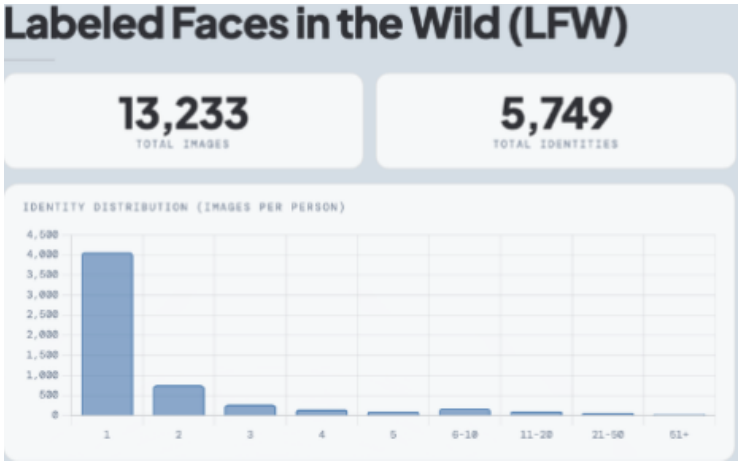
Introduction:

Facial recognition technology is being increasingly used by law enforcement and security services around the world, and is a rapidly growing sector of technology. Widespread concerns exist regarding the ethics, cost, and accuracy of it. The technology comes in many diverse forms, and we aim to see which of them produce the highest accuracy based on results from a sample dataset of facial images. Specifically, we will test facial recognition using FaceNet and logistic regression classification, and compare it to a deep learning method that uses FaceNet and cosine distance.

Through our project, we aim to answer the following questions:

- Can we get good results using these methods?
- Is one approach significantly better than the other?

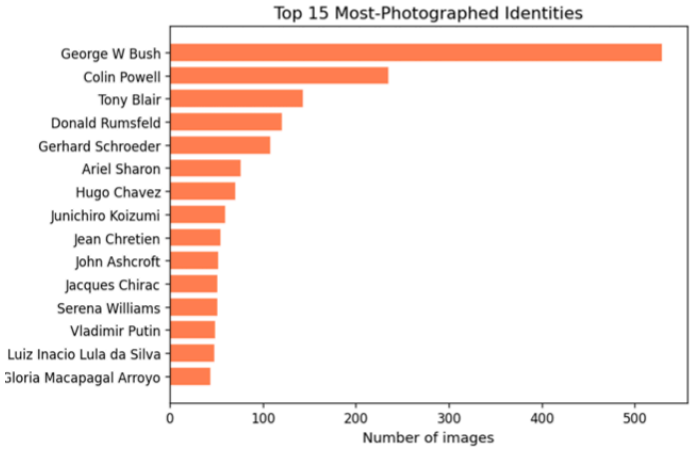
EDA:



Sample Faces from Filtered Dataset

Our dataset, Labeled Faces in the Wild, contains over 13,000 images of around 5700 different people, or identities. However, most of these identities correspond to only one image, so

to improve the performance of our models, we eliminate identities with less than 5 images. Each image is pre-aligned or funneled to ensure that each face appears in the same upright position. The challenge, however, is posed by the difference in lighting, facial expression, background, and occlusion between the images, which facial recognition technology must consider. This dataset contains images of not just randomly selected individuals among the global population, but also many images corresponding to famous people, such as politicians and celebrities, for example, shown below.



Methodology:

Face identification is the process of identifying a person's face and returning their name if they are found within our database, and if they are not in our database, the model's ability to accurately identify them as unknown is equally important. The underlying engine we used for face identification is FaceNet, a neural network which has been pretrained on VGGFace2, a dataset of 9,131 identities with nearly 3.3 million images (average of 362 images per identity) of faces with varied pose, race, age, and lighting, similar to Labelled Faces in the Wild.

FaceNet is designed to map faces into a vector space whereby the same persons are closer together and different people are farther apart. This is done using Triplet Loss, which aims to enforce that images of the same person are closer together in embedding space compared to images of different people. The goal behind this approach is to learn features which best discriminate between different people. While the default approach is to keep one embedding vector per image for each identity, we may choose to take the average so that each identity is only represented by one embedding vector. There is also another approach, called clustered templates, where you can tune the number of embeddings to balance between capturing variation without overfitting to the noise from single images.

Once we use the pretrained FaceNet model to translate the identities in our database into embeddings, these embeddings are then normalized so that every embedding lies on a unit hypersphere which means the norm or length of that embedding is equal to one. This is important

because it allows us to compare different embeddings in relation to the direction. So instead of thinking in terms of distance in space, we look at the angle between the vectors.

Once we have the normalized embeddings of every identity in our database, and we would like to use a test image to check if that image is of a known person from the database or whether this image is of an unknown person who is not in our database, we need to use a mechanism that tells us which identity is the closest in direction to the test image, and whether the distance is small enough to conclude that the person in the test image is in fact that person. We chose to compare cosine distance versus logistic regression for this task. We will discuss the performance of both in the following section.

The next question is how do we evaluate the performance of our model so that we are able to compare these different mechanisms. First, we filtered our dataset and excluded identities that have less than 5 images per that identity. This step is important as we need more images to capture more variation since our dataset has a high level of variation, where each image of the same person is significantly different from the others in relation to pose, lighting, and expression. The next step is to hold out one of these images for each identity to create a test set which we will use at the end for reporting the final accuracy metrics. With the remaining images left for each identity, we did 5-fold cross validation to find the optimal threshold for decision making. A threshold that is too low would result in a very strict decision where even the same person may be rejected as unknown due to variation in different factors such as lighting, expression, and others. Similarly, a threshold that is too high is also problematic because our decision would be too loose and will result in the model claiming we know who the person is despite them being unknown.

Metric	Measures	Ideal Direction	Most Important For
TIR	Correctly identifying known people	High (close to 100%)	Overall system effectiveness; user trust when recognized
FRR	Rejecting known people as unknown (misses)	Low (close to 0%)	User friendliness and convenience; minimizing frustration
FAR	Accepting unknown people as known (false alarms)	Low (close to 0%)	Security-critical systems; preventing unauthorized access

Our final question is what metrics should we look at to compare the different thresholds and different mechanisms and their performance on our face identification project. First, we will look at the True Identification Rate which tells us how many accurate predictions were made from among the test set images for the identities we hold in our database. The ideal value for TIR is 100%. While this is great to tell how well the model performs for people who are actually in the database, it doesn't cover the full picture. We need another metric for test images that are not in our database to tell how well the model rejects those images, which we hope it would. The metric for this is called the False Acceptance Rate. The ideal FAR is 0%. Usually, we start with a constraint on FAR based on how much risk we are willing to accept, and based on that select the

best TIR for usability. In summary, we start by setting a constraint on FAR based on how much risk we are willing to accept, and follow that by selecting the highest TIR which represents the model letting the real people in our dataset pass smoothly. The way we are able to compute FAR is by using identities that are not included in our database (unknown people) and we run them through our model to compute the model's ability to correctly identify these people as unknown. We used 600 identities from the lfw-dataset which only had one image for the purpose of computing FAR.

Main Result

Average 5-fold Cross Validation results for select thresholds:

Logistic Regression

Threshold	TIR	FAR
-3.263737	0.9527	0.9983
9.914608	0.8085	0.01
12.583015	0.5934	0.0017

Cosine Distance

Threshold	TIR	FAR
0.693520	0.9835	0.9983
0.307819	0.9338	0.02
0.268603	0.9078	0.0133
0.213931	0.8203	0.0017

Notice how as FIR decreases, the challenge is that TIR also decreases. To help understand this phenomenon, a good analogy to help understand why this happens, think of airport security, when you have a very strict guard, while they will stop more threats, they will also delay real passengers. So this is a tradeoff between usability and security. For this project, we decided to select the threshold by constraining FAR to ~ 1% such that out of 100 imposters, one will likely go through.

Test set accuracy using selected threshold from 5-fold cross validation:

	Selected Threshold	TIR	FAR

Logistic Regression	9.914608	0.7069	0.0033
Cosine Distance	0.268603	0.8936	0.0133

We can notice from the table that not only does logistic regression perform more poorly when compared to cosine distance, but it is also unstable. We know this because the FAR and TIR change significantly when we evaluate the same threshold against the test set as opposed to the 5-fold cross validation. On the other hand, notice how cosine distance yields similar TIR and FAR in both settings and outperforms logistic regression through its stability and balanced TIR at FAR.

Discussion:

This result was expected since cosine distance fits more appropriately into the pipeline given the following: Since FaceNet embeddings are L2-Normalized (length of embedding x is equal to the length of embedding y which is equal to one). We also know that for any two vectors x and y: $x \cdot y = |x|\,|y|\,cos(\theta)$ and since length of x and y are equal to one this reduces to $x \cdot y = cos(\theta)$ and therefore the angle $\theta = arccos(x \cdot y)$ which tells us how many degrees apart are the two embeddings. A small angle means the two images are more similar. Furthermore, we can compute a proxy for this distance in a more efficient manner such that instead of using $arccos$, we can use cosine distance which acts as a proxy and is more computationally efficient. Cosine distance $= 1 - (xy)$. This works because the embeddings are normalized and therefore $x \cdot y = cos(\theta)$. This acts as a quick score which tells us how large the angle is.

On the other hand, by using logistic regression instead of cosine distance, we are adding another classifier on top of an existing one. As discussed, FaceNet acts as a classifier because it learns features that aim to minimize intra-class distance and maximize inter-class distance. Therefore, when we use logistic regression with FaceNet, we are trying to learn a boundary on an already optimized space which adds noise instead of improving results.

We also attempted to use other forms of distance based measures, such as euclidean, squared euclidean, and Manhattan distance to explore their potential. We found they perform almost identically to cosine distance which shows there is no inherent advantage to using any of these other methods. One of the strengths of the cosine distance method is its simplicity and interpretability. Since FaceNet embeddings are L2-normalized, cosine distance may be applied directly to the geometry of the embedding space without requiring any extra learned parameters. This makes the approach lightweight, simple to reproduce, and less susceptible to overfitting on small data. It is also the fact that the threshold has a geometrical interpretation that is related to the angle between two vectors and this makes tuning and debugging more understandable. Also, cosine distance was highly stable both in 5-fold cross validation and the held-out test set, indicating that it generalizes well beyond the training data.

Although ultimately less useful here, logistic regression does have the theoretical benefit of being capable of learning a decision boundary that may capture patterns beyond what a simple

distance threshold can, in the event that the embedding space has been less well-structured. When the pretrained model fails to yield clean separations of identities, a learned classifier might have the chance to offset this by discovering patterns that a simple distance measure would omit. But, as we find, the FaceNet embedding space is already optimized to make such a separation, and hence this added flexibility is a liability, not an asset.

The major weakness of our method is that we did not fine-tune or retrain FaceNet on our dataset. Rather, we relied entirely on a model pretrained on VGGFace 2, and this implies that the embeddings might not entirely reflect the specific attributes of the Labeled Faces in the Wild dataset. Although similarities can be drawn between VGGFace 2 and LFW in the aspect of variation in pose, lighting, and expression, there are disparities in image quality, resolution, and demographic composition of the two datasets which may bring about a domain gap that influences performance. Perhaps by fine-tuning the model on LFW, or a subset of it, we can better optimize the TIR and FAR by generating embeddings that are more well-calibrated to the problems that exist in our data.

The other limitation is the number of images per identity is relatively small following the filtering. Although the identities have been narrowed down to those with at least 5 images, there are very few examples of some identities, which limits the model's ability to capture the full range of variation for those individuals. This applies especially to identities in which the images available vary widely in terms of lighting or expression since the test image held out might not be within the range represented by the rest of the training images. Both methods would probably perform better with a larger and more balanced dataset.

Lastly, we have not considered the possibility of biases in the dataset when it comes to our evaluation. Labeled Faces in the Wild is strongly biased toward public figures, and the demographic representation of the dataset is not representative of the world population. This implies that the accuracy measures we report might not be applicable to more diverse or representative populations, in which facial recognition systems have been shown to perform unevenly across different demographic groups

Conclusion:

Using a pretrained neural network (FaceNet) alongside cosine distance, we were able to achieve results that are satisfactory and stable, especially when you take into consideration the significant variation in relation to pose that is found within the images for each identity which makes facial recognition challenging. If we were using this model against test images that are clearly captured such as is the case during airport security or login systems used by phones, the test image would more clearly capture the face and we believe our current model would significantly boost its performance as a result. In addition, our focus on constraining ourselves to a lower False Acceptance Rate to ensure our model is robust for identifying imposters as unknowns makes the model more realistic and robust.

In addition, through this project we were able to appreciate the fact that tradeoffs always exist whenever we are dealing with building machine learning models. In this project, this was manifested through the threshold choice and balancing between usability (as reflected by the True Acceptance Rate) and security (as reflected by the False Acceptance Rate). While we were able to achieve a strong and stable TIR (~90%) and FAR (~1%) using FaceNet and cosine distance, we believe we can enhance our model further to meet the needs of higher security and usability by fine tuning our FaceNet model to our dataset, and by, as mentioned previously, using test images which more clearly capture the face, as opposed to some of the test images we had in our dataset which have extreme pose variation. In the real world, when we deploy these systems, we will most likely control the images we use by asking the person to look at the camera and so extreme variation in pose wouldn't be a problem. However, in other applications, for example where we would aim to identify people walking in the street using a security camera, the variation is significantly more relevant and in such cases we would appreciate having images in our database that reflect that setting. The ultimate lesson here is that, depending on the specific application we are building this model for, we must build a database that reflects that use-case and fine tune the neural network model to that dataset, and finally choose a threshold that reflects your specific values for your application.

Sources:
https://www.kaggle.com/datasets/atulanandjha/lfwpeople

Supplementary Material and Code:

Code found in this collab:

https://colab.research.google.com/drive/1ampRaCRcj2Rp2yAsyH3eV6LSOvYvsm-M?usp=sharing

Embeddings Extracted and saved:

```
import os
import random
import torch
import numpy as np
from PIL import Image
```

```
from facenet_pytorch import InceptionResnetV1, MTCNN

# ----------------------
# CONFIG
# ----------------------
DATASET_PATH = r"C:\Users\muras\OneDrive\Desktop\FaceNet\lfw_funneled"
PRETRAINED_MODEL_PATH =
r"C:\Users\muras\Downloads\20180402-114759-vggface2.pt"
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

MIN_IMAGES_PER_ID = 5
MAX_UNKNOWN_IDENTITIES = 600
BASE_SEED = 42

# NEW cache file name
NEW_EMBEDDINGS_CACHE_FILE = "lfw_facenet_embeddings_cache_v2.npz"

# ----------------------
# SEEDS
# ----------------------
np.random.seed(BASE_SEED)
random.seed(BASE_SEED)
torch.manual_seed(BASE_SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed(BASE_SEED)
    torch.cuda.manual_seed_all(BASE_SEED)

# ----------------------
# LOAD FACENET
# ----------------------
print("Loading FaceNet model...")
facenet = InceptionResnetV1(pretrained=None, classify=False).to(DEVICE)
state_dict = torch.load(PRETRAINED_MODEL_PATH, map_location=DEVICE)
state_dict = {k: v for k, v in state_dict.items() if not
k.startswith('logits.')}
facenet.load_state_dict(state_dict, strict=False)
facenet.eval()
print("FaceNet loaded successfully!")

# ----------------------
```

```
# INIT MTCNN
# ----------------------
mtcnn = MTCNN(image_size=160, margin=20, device=DEVICE)

# ----------------------
# HELPERS
# ----------------------
def embed_image(path):
    try:
        img = Image.open(path).convert('RGB')
    except Exception as e:
        print(f"Could not open image: {path} | Error: {e}")
        return None

    aligned = mtcnn(img)
    if aligned is None:
        return None

    aligned = aligned.unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        emb = facenet(aligned)
        emb = torch.nn.functional.normalize(emb, p=2, dim=1)
        emb = emb.cpu().numpy()[0]

    return emb.astype(np.float32)

def save_embeddings_cache(known_embs, unknown_embs, unknown_paths,
filename):
    print("\nSaving embeddings cache...")
    np.savez_compressed(
        filename,
        known_embs=np.array([known_embs], dtype=object),
        unknown_embs=np.array(unknown_embs, dtype=np.float32),
        unknown_paths=np.array(unknown_paths, dtype=object)
    )
    print(f"Embeddings saved to {filename}")

# ----------------------
# LOAD DATASET PATHS
# ----------------------
```

```
print("Loading dataset...")
known_person_to_paths = {}
unknown_candidates = []

for person in os.listdir(DATASET_PATH):
    folder = os.path.join(DATASET_PATH, person)
    if not os.path.isdir(folder):
        continue

    image_paths = [
        os.path.join(folder, f)
        for f in os.listdir(folder)
        if f.lower().endswith(('jpg', 'jpeg', 'png'))
    ]

    if len(image_paths) >= MIN_IMAGES_PER_ID:
        known_person_to_paths[person] = image_paths
    elif len(image_paths) == 1:
        unknown_candidates.append((person, image_paths[0]))

print(f"Known identities (>={MIN_IMAGES_PER_ID} images):
{len(known_person_to_paths)}")
print(f"Unknown single-image candidates: {len(unknown_candidates)}")

# ------------------------
# LIMIT UNKNOWNS
# ------------------------
if len(unknown_candidates) > 0:
    selected_indices = np.random.choice(
        len(unknown_candidates),
        min(MAX_UNKNOWN_IDENTITIES, len(unknown_candidates)),
        replace=False
    )
else:
    selected_indices = []

unknown_person_to_path = {}
for idx in selected_indices:
    person, img_path = unknown_candidates[idx]
    unknown_person_to_path[person] = img_path
```

```
print(f"Unknown identities used (max {MAX_UNKNOWN_IDENTITIES}):
{len(unknown_person_to_path)}")

# ----------------------
# EXTRACT KNOWN EMBEDDINGS
# ----------------------
print("\nExtracting embeddings for known identities...")
known_person_to_embs = {}
dropped_known = 0

for i, (person, image_paths) in enumerate(known_person_to_paths.items(),
1):
    embs = []
    valid_paths = []

    for path in image_paths:
        emb = embed_image(path)
        if emb is not None:
            embs.append(emb)
            valid_paths.append(path)

    if len(embs) >= MIN_IMAGES_PER_ID:
        known_person_to_embs[person] = {
            "embs": embs,
            "paths": valid_paths
        }
    else:
        dropped_known += 1

    if i % 100 == 0 or i == len(known_person_to_paths):
        print(f"Processed known identities:
{i}/{len(known_person_to_paths)}")

print(f"Known identities retained after embedding:
{len(known_person_to_embs)}")
print(f"Known identities dropped after failed detections:
{dropped_known}")

# ----------------------
```

```
# EXTRACT UNKNOWN EMBEDDINGS
# -----------------------
print("\nExtracting embeddings for unknown identities...")
unknown_embs = []
unknown_valid_paths = []

for i, (person, path) in enumerate(unknown_person_to_path.items(), 1):
    emb = embed_image(path)
    if emb is not None:
        unknown_embs.append(emb)
        unknown_valid_paths.append(path)

    if i % 100 == 0 or i == len(unknown_person_to_path):
        print(f"Processed unknown identities:
{i}/{len(unknown_person_to_path)}")

unknown_embs = np.array(unknown_embs, dtype=np.float32)
print(f"Unknown embeddings retained: {len(unknown_embs)}")

# -----------------------
# SAVE NEW CACHE
# -----------------------
save_embeddings_cache(
    known_person_to_embs,
    unknown_embs,
    unknown_valid_paths,
    NEW_EMBEDDINGS_CACHE_FILE
)

print("\nDone.")
```

Logistic Regression versus Distance-Based Methods Using saved embeddings already extracted and produce accuracy for a range of thresholds using 5 fold cross validation

```
import os
import json
import math
import torch
import random
```

```
import numpy as np
from PIL import Image
from itertools import combinations
from facenet_pytorch import InceptionResnetV1, MTCNN
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import pandas as pd

# ------------------------
# CONFIG
# ------------------------
DATASET_PATH = r"C:\Users\muras\OneDrive\Desktop\FaceNet\lfw_funneled"
PRETRAINED_MODEL_PATH =
r"C:\Users\muras\Downloads\20180402-114759-vggface2.pt"
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

MIN_IMAGES_PER_ID = 5
MAX_UNKNOWN_IDENTITIES = 600

# Clustered template settings
MAX_PROTOTYPES_PER_ID = 2
CLUSTER_MIN_SAMPLES = 4
KMEANS_RANDOM_STATE = 42

# Cross-validation settings
NUM_VAL_RUNS = 5
BASE_SEED = 42

# Threshold generation per method
NUM_THRESHOLDS_PER_METHOD = 25

# Pairwise classifier settings
MAX_POSITIVE_PAIRS_PER_ID = 20
NEGATIVE_TO_POSITIVE_RATIO = 2
MAX_TOTAL_POSITIVE_PAIRS = 12000

# Embedding cache
```

```
EMBEDDINGS_CACHE_FILE = "lfw_facenet_embeddings_cache.npz"

# Methods to evaluate
METHODS = [
    "cosine_distance",
    "euclidean_distance",
    "squared_euclidean_distance",
    "manhattan_distance",
    "logistic_regression",
    "linear_svm"
]

# Output files
ALL_RUNS_CSV = "open_set_validation_all_methods_5runs_all.csv"
AVG_RUNS_CSV = "open_set_validation_all_methods_5runs_avg.csv"
BEST_THRESHOLDS_CSV =
"open_set_validation_all_methods_best_thresholds.csv"
SPLIT_INFO_JSON = "open_set_fixed_test_split.json"

# -----------------------
# SEEDS
# -----------------------
np.random.seed(BASE_SEED)
random.seed(BASE_SEED)
torch.manual_seed(BASE_SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed(BASE_SEED)
    torch.cuda.manual_seed_all(BASE_SEED)

# -----------------------
# LOAD FACENET
# -----------------------
print("Loading FaceNet model...")
facenet = InceptionResnetV1(pretrained=None, classify=False).to(DEVICE)
state_dict = torch.load(PRETRAINED_MODEL_PATH, map_location=DEVICE)
state_dict = {k: v for k, v in state_dict.items() if not
k.startswith('logits.')}
facenet.load_state_dict(state_dict, strict=False)
facenet.eval()
print("FaceNet loaded successfully!")
```

```
# ----------------------
# INIT MTCNN
# ----------------------
mtcnn = MTCNN(image_size=160, margin=20, device=DEVICE)

# ----------------------
# HELPERS
# ----------------------
def l2_normalize(v: np.ndarray) -> np.ndarray:
    return v / (np.linalg.norm(v) + 1e-12)

def embed_image(path):
    try:
        img = Image.open(path).convert('RGB')
    except Exception:
        return None

    aligned = mtcnn(img)
    if aligned is None:
        return None

    aligned = aligned.unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        emb = facenet(aligned)
        emb = torch.nn.functional.normalize(emb, p=2, dim=1)
        emb = emb.cpu().numpy()[0]

    return emb.astype(np.float32)

def build_clustered_templates(
    embs: np.ndarray,
    max_prototypes: int = 3,
    cluster_min_samples: int = 6
) -> list:
    n = len(embs)
    if n == 0:
        return []

    if n < cluster_min_samples or max_prototypes <= 1:
```

```
        centroid = l2_normalize(np.mean(embs, axis=0))
        return [centroid.astype(np.float32)]

    k = min(max_prototypes, n)
    if k < 2:
        centroid = l2_normalize(np.mean(embs, axis=0))
        return [centroid.astype(np.float32)]

    km = KMeans(n_clusters=k, n_init=10, random_state=KMEANS_RANDOM_STATE)
    labels = km.fit_predict(embs)

    prototypes = []
    for c in range(k):
        cluster_embs = embs[labels == c]
        if len(cluster_embs) == 0:
            continue
        proto = l2_normalize(np.mean(cluster_embs, axis=0))
        prototypes.append(proto.astype(np.float32))

    if len(prototypes) == 0:
        centroid = l2_normalize(np.mean(embs, axis=0))
        prototypes = [centroid.astype(np.float32)]

    return prototypes

def pair_feature_absdiff(a: np.ndarray, b: np.ndarray) -> np.ndarray:
    return np.abs(a - b).astype(np.float32)

def method_is_distance(method: str) -> bool:
    return method in {
        "cosine_distance",
        "euclidean_distance",
        "squared_euclidean_distance",
        "manhattan_distance"
    }

def sample_combinations(index_pairs, max_samples, rng):
    if len(index_pairs) <= max_samples:
        return index_pairs
    chosen = rng.choice(len(index_pairs), size=max_samples, replace=False)
```

```
    return [index_pairs[i] for i in chosen]

def build_pairwise_training_data(gallery_raw_embs: dict, rng:
np.random.Generator):
    X = []
    y = []
    positive_count = 0

    for person, embs in gallery_raw_embs.items():
        if len(embs) < 2:
            continue

        all_pos_pairs = list(combinations(range(len(embs)), 2))
        sampled_pos_pairs = sample_combinations(all_pos_pairs,
MAX_POSITIVE_PAIRS_PER_ID, rng)

        for i, j in sampled_pos_pairs:
            X.append(pair_feature_absdiff(embs[i], embs[j]))
            y.append(1)
            positive_count += 1

    if positive_count == 0:
        return None, None

    if positive_count > MAX_TOTAL_POSITIVE_PAIRS:
        pos_indices = [i for i, label in enumerate(y) if label == 1]
        keep_pos_indices = rng.choice(pos_indices,
size=MAX_TOTAL_POSITIVE_PAIRS, replace=False)
        X = [X[i] for i in keep_pos_indices]
        y = [1] * len(X)
        positive_count = len(X)

    persons = [p for p, embs in gallery_raw_embs.items() if len(embs) > 0]
    target_negative_count = NEGATIVE_TO_POSITIVE_RATIO * positive_count

    negatives_added = 0
    attempts = 0
    max_attempts = max(target_negative_count * 10, 1000)
```

```
    while negatives_added < target_negative_count and attempts <
max_attempts:
        attempts += 1
        p1, p2 = rng.choice(persons, size=2, replace=False)
        emb1 = gallery_raw_embs[p1][rng.integers(0,
len(gallery_raw_embs[p1]))]
        emb2 = gallery_raw_embs[p2][rng.integers(0,
len(gallery_raw_embs[p2]))]

        X.append(pair_feature_absdiff(emb1, emb2))
        y.append(0)
        negatives_added += 1

    X = np.array(X, dtype=np.float32)
    y = np.array(y, dtype=np.int32)
    return X, y

def train_pairwise_models(gallery_raw_embs: dict, rng:
np.random.Generator):
    X, y = build_pairwise_training_data(gallery_raw_embs, rng)
    models = {
        "logistic_regression": None,
        "linear_svm": None
    }

    if X is None or y is None:
        print("  Pairwise models skipped: not enough positive pairs.")
        return models

    unique_classes = np.unique(y)
    if len(unique_classes) < 2:
        print("  Pairwise models skipped: only one class present.")
        return models

    try:
        lr_model = make_pipeline(
            StandardScaler(),
            LogisticRegression(
                solver="liblinear",
                max_iter=300,
```

```
                class_weight="balanced",
                random_state=BASE_SEED
            )
        )
        lr_model.fit(X, y)
        models["logistic_regression"] = lr_model
        print(f"  Logistic regression trained on {len(y)} pairs.")
    except Exception as e:
        print(f"  Logistic regression training failed: {e}")

    try:
        svm_model = make_pipeline(
            StandardScaler(),
            LinearSVC(
                class_weight="balanced",
                max_iter=5000,
                random_state=BASE_SEED
            )
        )
        svm_model.fit(X, y)
        models["linear_svm"] = svm_model
        print(f"  Linear SVM trained on {len(y)} pairs.")
    except Exception as e:
        print(f"  Linear SVM training failed: {e}")

    return models

# ----------------------
# FAST SCORING HELPERS
# ----------------------
def build_gallery_arrays(gallery: dict):
    """
    Converts gallery dict into flat arrays for faster scoring.
    Returns:
        proto_matrix: shape [num_prototypes, emb_dim]
        proto_persons: list of person names, one per prototype
        unique_persons: list of unique person names in stable order
    """
    proto_list = []
    proto_persons = []
```

```
    for person, protos in gallery.items():
        for p in protos:
            proto_list.append(p.astype(np.float32))
            proto_persons.append(person)

    if len(proto_list) == 0:
        return None, None, []

    proto_matrix = np.stack(proto_list).astype(np.float32)
    unique_persons = list(gallery.keys())
    return proto_matrix, proto_persons, unique_persons

def best_match_distance(query_emb: np.ndarray, proto_matrix: np.ndarray,
proto_persons: list, method: str):
    """
    Returns best_person, best_score for distance-based methods.
    Lower score is better.
    """
    if method == "cosine_distance":
        scores = 1.0 - np.dot(proto_matrix, query_emb)

    elif method == "euclidean_distance":
        diff = proto_matrix - query_emb
        scores = np.sqrt(np.sum(diff * diff, axis=1))

    elif method == "squared_euclidean_distance":
        diff = proto_matrix - query_emb
        scores = np.sum(diff * diff, axis=1)

    elif method == "manhattan_distance":
        scores = np.sum(np.abs(proto_matrix - query_emb), axis=1)

    else:
        raise ValueError(f"Unknown distance method: {method}")

    best_idx = int(np.argmin(scores))
    return proto_persons[best_idx], float(scores[best_idx])
```

```
def best_match_classifier(query_emb: np.ndarray, proto_matrix: np.ndarray,
proto_persons: list, model):
    """
    Returns best_person, best_score for classifier-based methods.
    Higher score is better.
    Uses decision_function for speed.
    """
    feats = np.abs(proto_matrix - query_emb).astype(np.float32)
    scores = model.decision_function(feats)

    best_idx = int(np.argmax(scores))
    return proto_persons[best_idx], float(scores[best_idx])

def precompute_best_matches_for_queries(
    known_val_embs: dict,
    unknown_embs: np.ndarray,
    gallery: dict,
    method: str,
    pair_models=None
):
    """
    Precompute best gallery identity and best score once per query.
    This is the main speedup.
    """
    proto_matrix, proto_persons, _ = build_gallery_arrays(gallery)

    if proto_matrix is None:
        return [], []

    known_results = []
    unknown_results = []

    if method_is_distance(method):
        for true_person, emb in known_val_embs.items():
            best_person, best_score = best_match_distance(emb,
proto_matrix, proto_persons, method)
            known_results.append({
                "true_person": true_person,
                "best_person": best_person,
                "best_score": best_score
```

```
            })

        for emb in unknown_embs:
            best_person, best_score = best_match_distance(emb,
proto_matrix, proto_persons, method)
            unknown_results.append({
                "best_person": best_person,
                "best_score": best_score
            })

    else:
        model = pair_models.get(method, None) if pair_models is not None
else None
        if model is None:
            return [], []

        for true_person, emb in known_val_embs.items():
            best_person, best_score = best_match_classifier(emb,
proto_matrix, proto_persons, model)
            known_results.append({
                "true_person": true_person,
                "best_person": best_person,
                "best_score": best_score
            })

        for emb in unknown_embs:
            best_person, best_score = best_match_classifier(emb,
proto_matrix, proto_persons, model)
            unknown_results.append({
                "best_person": best_person,
                "best_score": best_score
            })

    return known_results, unknown_results

def collect_candidate_scores_from_precomputed(known_results,
unknown_results):
    scores = []

    for r in known_results:
```

```
        if np.isfinite(r["best_score"]):
            scores.append(r["best_score"])

    for r in unknown_results:
        if np.isfinite(r["best_score"]):
            scores.append(r["best_score"])

    return scores

def build_threshold_grid(scores, num_thresholds=25):
    if len(scores) == 0:
        return []

    smin = float(np.min(scores))
    smax = float(np.max(scores))

    if math.isclose(smin, smax):
        return [smin]

    return [float(t) for t in np.linspace(smin, smax, num_thresholds)]

def evaluate_threshold_from_precomputed(method, threshold, known_results,
unknown_results):
    true_id = 0
    misidentified = 0
    false_accept = 0

    if method_is_distance(method):
        for r in known_results:
            if r["best_score"] < threshold:
                pred = r["best_person"]
            else:
                pred = "unknown"

            if pred == r["true_person"]:
                true_id += 1
            elif pred != "unknown":
                misidentified += 1

        for r in unknown_results:
```

```
            pred = r["best_person"] if r["best_score"] < threshold else
"unknown"
            if pred != "unknown":
                false_accept += 1

    else:
        for r in known_results:
            if r["best_score"] > threshold:
                pred = r["best_person"]
            else:
                pred = "unknown"

            if pred == r["true_person"]:
                true_id += 1
            elif pred != "unknown":
                misidentified += 1

        for r in unknown_results:
            pred = r["best_person"] if r["best_score"] > threshold else
"unknown"
            if pred != "unknown":
                false_accept += 1

    num_known = len(known_results)
    num_unknown = len(unknown_results)

    TIR = true_id / num_known if num_known > 0 else 0.0
    FIR = misidentified / num_known if num_known > 0 else 0.0
    FAR = false_accept / num_unknown if num_unknown > 0 else 0.0

    return TIR, FIR, FAR

def save_embeddings_cache(known_embs, unknown_embs, unknown_paths,
filename):
    print("\nSaving embeddings cache...")
    np.savez_compressed(
        filename,
        known_embs=np.array([known_embs], dtype=object),
        unknown_embs=np.array(unknown_embs, dtype=np.float32),
        unknown_paths=np.array(unknown_paths, dtype=object)
```

```
    )
    print(f"Embeddings saved to {filename}")

def load_embeddings_cache(filename):
    print("\nLoading embeddings cache...")
    data = np.load(filename, allow_pickle=True)
    known_embs = data["known_embs"].item()
    unknown_embs = data["unknown_embs"]
    unknown_paths = data["unknown_paths"].tolist()
    print("Embeddings loaded successfully!")
    return known_embs, unknown_embs, unknown_paths

# -----------------------
# LOAD DATASET PATHS
# -----------------------
print("Loading dataset...")
known_person_to_paths = {}
unknown_candidates = []

for person in os.listdir(DATASET_PATH):
    folder = os.path.join(DATASET_PATH, person)
    if not os.path.isdir(folder):
        continue

    image_paths = [
        os.path.join(folder, f)
        for f in os.listdir(folder)
        if f.lower().endswith(('jpg', 'jpeg', 'png'))
    ]

    if len(image_paths) >= MIN_IMAGES_PER_ID:
        known_person_to_paths[person] = image_paths
    elif len(image_paths) == 1:
        unknown_candidates.append((person, image_paths[0]))

print(f"Known identities (>={MIN_IMAGES_PER_ID} images):
{len(known_person_to_paths)}")
print(f"Unknown single-image candidates: {len(unknown_candidates)}")

# -----------------------
```

```
# LIMIT UNKNOWNS (FIXED ONCE)
# -----------------------
if len(unknown_candidates) > 0:
    selected_indices = np.random.choice(
        len(unknown_candidates),
        min(MAX_UNKNOWN_IDENTITIES, len(unknown_candidates)),
        replace=False
    )
else:
    selected_indices = []

unknown_person_to_path = {}
for idx in selected_indices:
    person, img_path = unknown_candidates[idx]
    unknown_person_to_path[person] = img_path

print(f"Unknown identities used (max {MAX_UNKNOWN_IDENTITIES}):
{len(unknown_person_to_path)}")

# -----------------------
# PRECOMPUTE OR LOAD EMBEDDINGS
# -----------------------
if os.path.exists(EMBEDDINGS_CACHE_FILE):
    known_person_to_embs, unknown_embs, unknown_valid_paths =
load_embeddings_cache(
        EMBEDDINGS_CACHE_FILE
    )
else:
    print("\nPrecomputing embeddings once for all known identities...")
    known_person_to_embs = {}
    dropped_known = 0

    for i, (person, image_paths) in
enumerate(known_person_to_paths.items(), 1):
        embs = []
        valid_paths = []

        for path in image_paths:
            emb = embed_image(path)
            if emb is not None:
```

```
                embs.append(emb)
                valid_paths.append(path)

        if len(embs) >= MIN_IMAGES_PER_ID:
            known_person_to_embs[person] = {
                "embs": embs,
                "paths": valid_paths
            }
        else:
            dropped_known += 1

        if i % 100 == 0 or i == len(known_person_to_paths):
            print(f"Processed known identities:
{i}/{len(known_person_to_paths)}")

    print(f"Known identities retained after embedding:
{len(known_person_to_embs)}")
    print(f"Known identities dropped after failed detections:
{dropped_known}")

    print("\nPrecomputing embeddings once for unknown identities...")
    unknown_embs = []
    unknown_valid_paths = []

    for i, (person, path) in enumerate(unknown_person_to_path.items(), 1):
        emb = embed_image(path)
        if emb is not None:
            unknown_embs.append(emb)
            unknown_valid_paths.append(path)

        if i % 100 == 0 or i == len(unknown_person_to_path):
            print(f"Processed unknown identities:
{i}/{len(unknown_person_to_path)}")

    unknown_embs = np.array(unknown_embs, dtype=np.float32)
    print(f"Unknown embeddings retained: {len(unknown_embs)}")

    save_embeddings_cache(
        known_person_to_embs,
        unknown_embs,
```

```
        unknown_valid_paths,
        EMBEDDINGS_CACHE_FILE
    )

# ----------------------
# HOLD OUT FIXED TEST IMAGE ONCE
# ----------------------
print("\nCreating fixed held-out test split...")
rng_test = np.random.default_rng(BASE_SEED)

known_fixed_test = {}
known_remaining_for_cv = {}

for person, data in known_person_to_embs.items():
    embs = data["embs"]
    paths = data["paths"]

    idxs = np.arange(len(embs))
    rng_test.shuffle(idxs)

    test_idx = idxs[0]
    rem_idxs = idxs[1:]

    if len(rem_idxs) < 2:
        continue

    known_fixed_test[person] = {
        "test_emb": embs[test_idx],
        "test_path": paths[test_idx]
    }
    known_remaining_for_cv[person] = {
        "embs": [embs[j] for j in rem_idxs],
        "paths": [paths[j] for j in rem_idxs]
    }

print(f"Known identities with fixed held-out test image:
{len(known_fixed_test)}")

# ----------------------
# FIVE RANDOM VALIDATION RUNS
```

```
# ----------------------
all_run_results = []

for run_idx in range(NUM_VAL_RUNS):
    print(f"\n================ RUN {run_idx + 1}/{NUM_VAL_RUNS}
================")
    rng = np.random.default_rng(BASE_SEED + run_idx)

    gallery = {}
    gallery_raw_embs = {}
    known_val_embs = {}

    for person, data in known_remaining_for_cv.items():
        embs = data["embs"]

        idxs = np.arange(len(embs))
        rng.shuffle(idxs)

        val_idx = idxs[0]
        gallery_idxs = idxs[1:]

        if len(gallery_idxs) == 0:
            continue

        val_emb = embs[val_idx]
        gallery_embs = np.array([embs[j] for j in gallery_idxs],
dtype=np.float32)

        protos = build_clustered_templates(
            gallery_embs,
            max_prototypes=MAX_PROTOTYPES_PER_ID,
            cluster_min_samples=CLUSTER_MIN_SAMPLES
        )

        if len(protos) > 0:
            gallery[person] = protos
            gallery_raw_embs[person] = [embs[j] for j in gallery_idxs]
            known_val_embs[person] = val_emb

    print(f"Gallery identities: {len(gallery)}")
```

```
    if len(gallery) > 0:
        avg_protos = np.mean([len(v) for v in gallery.values()])
        print(f"Avg prototypes per identity: {avg_protos:.2f}")

    print(f"Known validation embeddings: {len(known_val_embs)}")
    print(f"Unknown embeddings: {len(unknown_embs)}")

    print("Training pairwise models...")
    pair_models = train_pairwise_models(gallery_raw_embs, rng)

    for method in METHODS:
        if method in {"logistic_regression", "linear_svm"} and
pair_models.get(method) is None:
            print(f"\nSkipping {method}: model unavailable.")
            continue

        print(f"\n----- METHOD: {method} | RUN {run_idx + 1} -----")

        # MAIN SPEEDUP: compute best match and best score ONCE per query
        known_results, unknown_results =
precompute_best_matches_for_queries(
            known_val_embs=known_val_embs,
            unknown_embs=unknown_embs,
            gallery=gallery,
            method=method,
            pair_models=pair_models
        )

        candidate_scores = collect_candidate_scores_from_precomputed(
            known_results=known_results,
            unknown_results=unknown_results
        )

        thresholds = build_threshold_grid(
            scores=candidate_scores,
            num_thresholds=NUM_THRESHOLDS_PER_METHOD
        )

        if len(thresholds) == 0:
            print("No thresholds generated; skipping method.")
```

```
            continue

        method_results = []

        for threshold in thresholds:
            TIR, FIR, FAR = evaluate_threshold_from_precomputed(
                method=method,
                threshold=threshold,
                known_results=known_results,
                unknown_results=unknown_results
            )

            method_results.append([
                run_idx + 1,
                method,
                threshold,
                TIR,
                FIR,
                FAR
            ])

        run_df = pd.DataFrame(
            method_results,
            columns=["Run", "Method", "Threshold", "TIR", "FIR", "FAR"]
        )
        all_run_results.append(run_df)

        print(
            run_df.round({
                "Threshold": 6,
                "TIR": 4,
                "FIR": 4,
                "FAR": 4
            }).to_string(index=False, col_space=14)
        )

# ----------------------
# COMBINE ALL RUNS
# ----------------------
if len(all_run_results) == 0:
```

```
    raise RuntimeError("No results were generated.")

all_runs_df = pd.concat(all_run_results, ignore_index=True)
all_runs_df_rounded = all_runs_df.round({
    "Threshold": 6,
    "TIR": 4,
    "FIR": 4,
    "FAR": 4
})
all_runs_df_rounded.to_csv(ALL_RUNS_CSV, index=False)
print(f"\nSaved all run results to {ALL_RUNS_CSV}")

# -----------------------
# AVERAGE ACROSS RUNS
# -----------------------
avg_results_df = (
    all_runs_df
    .groupby(["Method", "Threshold"], as_index=False)[["TIR", "FIR",
"FAR"]]
    .mean()
)

avg_results_df = avg_results_df.round({
    "Threshold": 6,
    "TIR": 4,
    "FIR": 4,
    "FAR": 4
})

print("\n===== AVERAGED VALIDATION RESULTS BY METHOD =====")
for method in avg_results_df["Method"].unique():
    print(f"\n----- {method} -----")
    method_df = avg_results_df[avg_results_df["Method"] == method].copy()
    print(method_df.to_string(index=False, col_space=14))

avg_results_df.to_csv(AVG_RUNS_CSV, index=False)
print(f"\nSaved averaged validation results to {AVG_RUNS_CSV}")

# -----------------------
# BEST THRESHOLD PER METHOD
```

```
# Sort: highest TIR, then lowest FAR, then lowest FIR
# ----------------------
best_thresholds_df = (
    avg_results_df
    .sort_values(by=["Method", "TIR", "FAR", "FIR"], ascending=[True,
False, True, True])
    .groupby("Method", as_index=False)
    .first()
)

best_thresholds_df.to_csv(BEST_THRESHOLDS_CSV, index=False)

print("\n===== BEST THRESHOLD PER METHOD =====")
print(best_thresholds_df.to_string(index=False, col_space=14))
print(f"\nSaved best thresholds to {BEST_THRESHOLDS_CSV}")

# ----------------------
# SAVE FIXED TEST SPLIT FOR LATER
# ----------------------
split_info = {
    "base_seed": BASE_SEED,
    "min_images_per_id": MIN_IMAGES_PER_ID,
    "max_unknown_identities": MAX_UNKNOWN_IDENTITIES,
    "num_val_runs": NUM_VAL_RUNS,
    "methods": METHODS,
    "known_test_paths": {person: data["test_path"] for person, data in
known_fixed_test.items()},
    "unknown_paths": unknown_valid_paths
}

with open(SPLIT_INFO_JSON, "w") as f:
    json.dump(split_info, f, indent=2)

print(f"Saved fixed test split info to {SPLIT_INFO_JSON}")
print("\nDone. Compare methods using TIR, FIR, and FAR.")
```

compute accuracy using selected threshold Against test set

```
import os
```

```
import json
import torch
import random
import numpy as np
from PIL import Image
from itertools import combinations
from facenet_pytorch import InceptionResnetV1, MTCNN
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import pandas as pd

# -----------------------
# CONFIG
# -----------------------
DATASET_PATH = r"C:\Users\muras\OneDrive\Desktop\FaceNet\lfw_funneled"
PRETRAINED_MODEL_PATH =
r"C:\Users\muras\Downloads\20180402-114759-vggface2.pt"
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

MIN_IMAGES_PER_ID = 5
MAX_UNKNOWN_IDENTITIES = 600

# Clustered template settings
MAX_PROTOTYPES_PER_ID = 2
CLUSTER_MIN_SAMPLES = 4
KMEANS_RANDOM_STATE = 42

# Pairwise classifier settings
MAX_POSITIVE_PAIRS_PER_ID = 20
NEGATIVE_TO_POSITIVE_RATIO = 2
MAX_TOTAL_POSITIVE_PAIRS = 12000

# Reproducibility
BASE_SEED = 42
np.random.seed(BASE_SEED)
random.seed(BASE_SEED)
torch.manual_seed(BASE_SEED)
```

```
if torch.cuda.is_available():
    torch.cuda.manual_seed(BASE_SEED)
    torch.cuda.manual_seed_all(BASE_SEED)

# Files
EMBEDDINGS_CACHE_FILE = "lfw_facenet_embeddings_cache.npz"
SPLIT_INFO_JSON = "open_set_fixed_test_split.json"
TEST_RESULTS_CSV = "open_set_test_results_selected_thresholds.csv"

# -----------------------
# CHOSEN THRESHOLDS
# -----------------------
SELECTED_THRESHOLDS = {
    "cosine_distance": 0.268603,
    "euclidean_distance": 0.773416,
    "linear_svm": 2.267476,
    "logistic_regression": 9.914608,
    "manhattan_distance": 14.113981,
    "squared_euclidean_distance": 0.603021
}

METHODS = list(SELECTED_THRESHOLDS.keys())

# -----------------------
# LOAD FACENET
# -----------------------
print("Loading FaceNet model...")
facenet = InceptionResnetV1(pretrained=None, classify=False).to(DEVICE)
state_dict = torch.load(PRETRAINED_MODEL_PATH, map_location=DEVICE)
state_dict = {k: v for k, v in state_dict.items() if not
k.startswith('logits.')}
facenet.load_state_dict(state_dict, strict=False)
facenet.eval()
print("FaceNet loaded successfully!")

# -----------------------
# INIT MTCNN
# -----------------------
mtcnn = MTCNN(image_size=160, margin=20, device=DEVICE)
```

```python
# ----------------------
# HELPERS
# ----------------------
def l2_normalize(v: np.ndarray) -> np.ndarray:
    return v / (np.linalg.norm(v) + 1e-12)

def embed_image(path):
    try:
        img = Image.open(path).convert('RGB')
    except Exception:
        return None

    aligned = mtcnn(img)
    if aligned is None:
        return None

    aligned = aligned.unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        emb = facenet(aligned)
        emb = torch.nn.functional.normalize(emb, p=2, dim=1)
        emb = emb.cpu().numpy()[0]

    return emb.astype(np.float32)

def build_clustered_templates(
    embs: np.ndarray,
    max_prototypes: int = 3,
    cluster_min_samples: int = 6
) -> list:
    n = len(embs)
    if n == 0:
        return []

    if n < cluster_min_samples or max_prototypes <= 1:
        centroid = l2_normalize(np.mean(embs, axis=0))
        return [centroid.astype(np.float32)]

    k = min(max_prototypes, n)
    if k < 2:
        centroid = l2_normalize(np.mean(embs, axis=0))
```

```
        return [centroid.astype(np.float32)]

    km = KMeans(n_clusters=k, n_init=10, random_state=KMEANS_RANDOM_STATE)
    labels = km.fit_predict(embs)

    prototypes = []
    for c in range(k):
        cluster_embs = embs[labels == c]
        if len(cluster_embs) == 0:
            continue
        proto = l2_normalize(np.mean(cluster_embs, axis=0))
        prototypes.append(proto.astype(np.float32))

    if len(prototypes) == 0:
        centroid = l2_normalize(np.mean(embs, axis=0))
        prototypes = [centroid.astype(np.float32)]

    return prototypes

def pair_feature_absdiff(a: np.ndarray, b: np.ndarray) -> np.ndarray:
    return np.abs(a - b).astype(np.float32)

def method_is_distance(method: str) -> bool:
    return method in {
        "cosine_distance",
        "euclidean_distance",
        "squared_euclidean_distance",
        "manhattan_distance"
    }

def sample_combinations(index_pairs, max_samples, rng):
    if len(index_pairs) <= max_samples:
        return index_pairs
    chosen = rng.choice(len(index_pairs), size=max_samples, replace=False)
    return [index_pairs[i] for i in chosen]

def build_pairwise_training_data(gallery_raw_embs: dict, rng:
np.random.Generator):
    X = []
    y = []
```

```
    positive_count = 0

    for person, embs in gallery_raw_embs.items():
        if len(embs) < 2:
            continue

        all_pos_pairs = list(combinations(range(len(embs)), 2))
        sampled_pos_pairs = sample_combinations(all_pos_pairs,
MAX_POSITIVE_PAIRS_PER_ID, rng)

        for i, j in sampled_pos_pairs:
            X.append(pair_feature_absdiff(embs[i], embs[j]))
            y.append(1)
            positive_count += 1

    if positive_count == 0:
        return None, None

    if positive_count > MAX_TOTAL_POSITIVE_PAIRS:
        pos_indices = [i for i, label in enumerate(y) if label == 1]
        keep_pos_indices = rng.choice(pos_indices,
size=MAX_TOTAL_POSITIVE_PAIRS, replace=False)
        X = [X[i] for i in keep_pos_indices]
        y = [1] * len(X)
        positive_count = len(X)

    persons = [p for p, embs in gallery_raw_embs.items() if len(embs) > 0]
    target_negative_count = NEGATIVE_TO_POSITIVE_RATIO * positive_count

    negatives_added = 0
    attempts = 0
    max_attempts = max(target_negative_count * 10, 1000)

    while negatives_added < target_negative_count and attempts <
max_attempts:
        attempts += 1
        p1, p2 = rng.choice(persons, size=2, replace=False)
        emb1 = gallery_raw_embs[p1][rng.integers(0,
len(gallery_raw_embs[p1]))]
```

```
        emb2 = gallery_raw_embs[p2][rng.integers(0,
len(gallery_raw_embs[p2]))]

        X.append(pair_feature_absdiff(emb1, emb2))
        y.append(0)
        negatives_added += 1

    X = np.array(X, dtype=np.float32)
    y = np.array(y, dtype=np.int32)
    return X, y

def train_pairwise_models(gallery_raw_embs: dict, rng:
np.random.Generator):
    X, y = build_pairwise_training_data(gallery_raw_embs, rng)
    models = {
        "logistic_regression": None,
        "linear_svm": None
    }

    if X is None or y is None:
        print("Pairwise models skipped: not enough positive pairs.")
        return models

    unique_classes = np.unique(y)
    if len(unique_classes) < 2:
        print("Pairwise models skipped: only one class present.")
        return models

    try:
        lr_model = make_pipeline(
            StandardScaler(),
            LogisticRegression(
                solver="liblinear",
                max_iter=300,
                class_weight="balanced",
                random_state=BASE_SEED
            )
        )
        lr_model.fit(X, y)
        models["logistic_regression"] = lr_model
```

```
        print(f"Logistic regression trained on {len(y)} pairs.")
    except Exception as e:
        print(f"Logistic regression training failed: {e}")

    try:
        svm_model = make_pipeline(
            StandardScaler(),
            LinearSVC(
                class_weight="balanced",
                max_iter=5000,
                random_state=BASE_SEED
            )
        )
        svm_model.fit(X, y)
        models["linear_svm"] = svm_model
        print(f"Linear SVM trained on {len(y)} pairs.")
    except Exception as e:
        print(f"Linear SVM training failed: {e}")

    return models

def build_gallery_arrays(gallery: dict):
    proto_list = []
    proto_persons = []

    for person, protos in gallery.items():
        for p in protos:
            proto_list.append(p.astype(np.float32))
            proto_persons.append(person)

    if len(proto_list) == 0:
        return None, None

    proto_matrix = np.stack(proto_list).astype(np.float32)
    return proto_matrix, proto_persons

def best_match_distance(query_emb: np.ndarray, proto_matrix: np.ndarray,
proto_persons: list, method: str):
    if method == "cosine_distance":
        scores = 1.0 - np.dot(proto_matrix, query_emb)
```

```
    elif method == "euclidean_distance":
        diff = proto_matrix - query_emb
        scores = np.sqrt(np.sum(diff * diff, axis=1))

    elif method == "squared_euclidean_distance":
        diff = proto_matrix - query_emb
        scores = np.sum(diff * diff, axis=1)

    elif method == "manhattan_distance":
        scores = np.sum(np.abs(proto_matrix - query_emb), axis=1)

    else:
        raise ValueError(f"Unknown distance method: {method}")

    best_idx = int(np.argmin(scores))
    return proto_persons[best_idx], float(scores[best_idx])

def best_match_classifier(query_emb: np.ndarray, proto_matrix: np.ndarray,
proto_persons: list, model):
    feats = np.abs(proto_matrix - query_emb).astype(np.float32)
    scores = model.decision_function(feats)

    best_idx = int(np.argmax(scores))
    return proto_persons[best_idx], float(scores[best_idx])

def identify_with_threshold(query_emb, proto_matrix, proto_persons,
method, threshold, pair_models):
    if method_is_distance(method):
        best_person, best_score = best_match_distance(query_emb,
proto_matrix, proto_persons, method)
        pred = best_person if best_score < threshold else "unknown"
        return pred, best_score
    else:
        model = pair_models[method]
        best_person, best_score = best_match_classifier(query_emb,
proto_matrix, proto_persons, model)
        pred = best_person if best_score > threshold else "unknown"
        return pred, best_score
```

```python
def load_embeddings_cache(filename):
    print("Loading embeddings cache...")
    data = np.load(filename, allow_pickle=True)
    known_embs = data["known_embs"].item()
    unknown_embs = data["unknown_embs"]
    unknown_paths = data["unknown_paths"].tolist()
    print("Embeddings loaded successfully!")
    return known_embs, unknown_embs, unknown_paths

# -----------------------
# LOAD EMBEDDINGS CACHE
# -----------------------
if not os.path.exists(EMBEDDINGS_CACHE_FILE):
    raise FileNotFoundError(
        f"Missing embeddings cache: {EMBEDDINGS_CACHE_FILE}\n"
        f"Run your validation script first so embeddings are saved."
    )

known_person_to_embs, unknown_embs, unknown_valid_paths =
load_embeddings_cache(EMBEDDINGS_CACHE_FILE)

# -----------------------
# LOAD / REBUILD FIXED TEST SPLIT
# -----------------------
print("\nCreating fixed held-out test split...")
rng_test = np.random.default_rng(BASE_SEED)

known_fixed_test = {}
known_remaining_for_test = {}

for person, data in known_person_to_embs.items():
    embs = data["embs"]
    paths = data["paths"]

    idxs = np.arange(len(embs))
    rng_test.shuffle(idxs)

    test_idx = idxs[0]
    rem_idxs = idxs[1:]
```

```
    if len(rem_idxs) < 1:
        continue

    known_fixed_test[person] = {
        "test_emb": embs[test_idx],
        "test_path": paths[test_idx]
    }
    known_remaining_for_test[person] = {
        "embs": [embs[j] for j in rem_idxs],
        "paths": [paths[j] for j in rem_idxs]
    }

print(f"Known identities with held-out test image:
{len(known_fixed_test)}")
print(f"Unknown embeddings available: {len(unknown_embs)}")

# Optional consistency check with saved split info if present
if os.path.exists(SPLIT_INFO_JSON):
    with open(SPLIT_INFO_JSON, "r") as f:
        split_info = json.load(f)

    saved_test_paths = split_info.get("known_test_paths", {})
    mismatches = 0
    for person, data in known_fixed_test.items():
        saved_path = saved_test_paths.get(person, None)
        if saved_path is not None and saved_path != data["test_path"]:
            mismatches += 1

    if mismatches == 0:
        print("Fixed test split matches saved split info.")
    else:
        print(f"Warning: {mismatches} test-path mismatches found vs saved
split info.")

# ----------------------
# BUILD FINAL GALLERY FROM ALL NON-TEST IMAGES
# ----------------------
print("\nBuilding final gallery from all non-test images...")
gallery = {}
gallery_raw_embs = {}
```

```
for person, data in known_remaining_for_test.items():
    embs = np.array(data["embs"], dtype=np.float32)

    if len(embs) == 0:
        continue

    protos = build_clustered_templates(
        embs,
        max_prototypes=MAX_PROTOTYPES_PER_ID,
        cluster_min_samples=CLUSTER_MIN_SAMPLES
    )

    if len(protos) > 0:
        gallery[person] = protos
        gallery_raw_embs[person] = list(embs)

print(f"Gallery identities: {len(gallery)}")
if len(gallery) == 0:
    raise RuntimeError("Gallery is empty.")

avg_protos = np.mean([len(v) for v in gallery.values()])
print(f"Average prototypes per identity: {avg_protos:.2f}")

proto_matrix, proto_persons = build_gallery_arrays(gallery)
if proto_matrix is None:
    raise RuntimeError("Failed to build gallery prototype matrix.")

# ----------------------
# TRAIN LEARNED METHODS ON FINAL GALLERY
# ----------------------
print("\nTraining pairwise models on final gallery...")
rng_models = np.random.default_rng(BASE_SEED)
pair_models = train_pairwise_models(gallery_raw_embs, rng_models)

# ----------------------
# EVALUATE TEST SET
# ----------------------
print("\nEvaluating test set with selected thresholds...")
results = []
```

```
known_test_count = len(known_fixed_test)
unknown_count = len(unknown_embs)

for method in METHODS:
    threshold = SELECTED_THRESHOLDS[method]

    if method in {"logistic_regression", "linear_svm"} and
pair_models.get(method) is None:
        print(f"Skipping {method}: trained model unavailable.")
        continue

    true_id = 0
    misidentified = 0
    false_accept = 0

    # Known held-out test images
    for person, data in known_fixed_test.items():
        emb = data["test_emb"]

        pred, score = identify_with_threshold(
            query_emb=emb,
            proto_matrix=proto_matrix,
            proto_persons=proto_persons,
            method=method,
            threshold=threshold,
            pair_models=pair_models
        )

        if pred == person:
            true_id += 1
        elif pred != "unknown":
            misidentified += 1

    # Unknown identities
    for emb in unknown_embs:
        pred, score = identify_with_threshold(
            query_emb=emb,
            proto_matrix=proto_matrix,
            proto_persons=proto_persons,
```

```
            method=method,
            threshold=threshold,
            pair_models=pair_models
        )

        if pred != "unknown":
            false_accept += 1

    TIR = true_id / known_test_count if known_test_count > 0 else 0.0
    FIR = misidentified / known_test_count if known_test_count > 0 else
0.0
    FAR = false_accept / unknown_count if unknown_count > 0 else 0.0

    results.append([
        method,
        threshold,
        true_id,
        misidentified,
        false_accept,
        known_test_count,
        unknown_count,
        TIR,
        FIR,
        FAR
    ])

# -----------------------
# SAVE + PRINT RESULTS
# -----------------------
results_df = pd.DataFrame(
    results,
    columns=[
        "Method",
        "Threshold",
        "True_ID",
        "Misidentified",
        "False_Accept",
        "Num_Known_Test",
        "Num_Unknown",
        "TIR",
```

```
        "FIR",
        "FAR"
    ]
)

results_df = results_df.sort_values("Method").reset_index(drop=True)

print("\n===== TEST SET RESULTS USING SELECTED THRESHOLDS =====")
print(
    results_df.round({
        "Threshold": 6,
        "TIR": 4,
        "FIR": 4,
        "FAR": 4
    }).to_string(index=False, col_space=14)
)

results_df.to_csv(TEST_RESULTS_CSV, index=False)
print(f"\nSaved test results to: {TEST_RESULTS_CSV}")
```