AAI-521 Final Team Project Report - Group 2

Advanced Computer Vision:

Face Recognition using FaceNet and DeepFace

by

Mohammad Alkhawaldeh, Narendra Fadnavis, Subhabrata Ganguli

University of San Diego

Dec 09, 2024

1. Introduction

Facial recognition has become a pivotal technology in modern computer vision, finding applications in security, authentication, and social networking platforms. The purpose of this project, titled *Face Recognition using FaceNet and DeepFace*, is to develop and evaluate a face recognition system leveraging two state-of-the-art deep learning models, FaceNet and DeepFace. This study aims to compare their performances on the widely used Labeled Faces in the Wild (LFW) dataset.

The LFW dataset, hosted by the University of Massachusetts Amherst, is a benchmark dataset for evaluating unconstrained face recognition models. It consists of 13,233 images of 5,749 individuals, collected from the web using the Viola-Jones face detection algorithm. Among the dataset, 1,680 individuals have two or more distinct images, making it ideal for evaluating facial recognition systems. The images in the dataset are pre-processed and centered, ensuring consistency across samples. This project utilizes this dataset to assess how effectively the FaceNet and DeepFace models can identify and differentiate faces under real-world, unconstrained conditions.

2. Data Preprocessing and EDA

Data Preprocessing

The Labeled Faces in the Wild (LFW) dataset was the foundation for this project. The preprocessing steps were designed to prepare the dataset for embedding extraction and subsequent modeling. Key steps included:

1. Data Download and Extraction:

- The LFW dataset was downloaded and extracted into a structured folder hierarchy where each subfolder represented a unique individual dataset.

  location: `/content/drive/MyDrive/AAI521_FinalProject/lfw/lfw-deepfunneled`

2. Image Preprocessing:

- Images were resized to 160x160 pixels (required by FaceNet).
- Converted images to RGB format for compatibility with DeepFace and FaceNet.

Code Reference: `image = cv2.resize(image_rgb, (160, 160))`

3. Dataset Conversion:

- Images and their respective labels were stored as numpy arrays for efficient processing

```
Processing Folders: 100%|██████████| 100/100 [01:33<00:00,  1.07it/s]
Total images: 235
Total labels: 235
```

4. Data Saving and Loading:

- Processed images and labels were saved using the `pickle` module for efficient reloading without redundant preprocessing.

Exploratory Data Analysis (EDA)

EDA was performed to gain insights into the dataset and identify potential preprocessing needs.

1.Folder Structure Summary:

- A detailed count of images per subfolder was generated, providing an overview of class distribution.
- *Code Reference*: `folder_structure_summary()`

Counting Images per Folder: 100%|██████████| 100/100 [00:00<00:00, 842.27it/s]Number of subfolders (classes): 100

Images per subfolder (as a table):

	Subfolder	Image Count
0	AJ_Cook	1
1	AJ_Lamas	1
2	Aaron_Eckhart	1
3	Aaron_Guiel	1
4	Aaron_Patterson	1
...	...	...
95	Alan_Jackson	1
96	Alan_Mulally	2
97	Alan_Stonecipher	1
98	Alan_Tang_Kwong-wing	1
99	Alan_Trammell	1

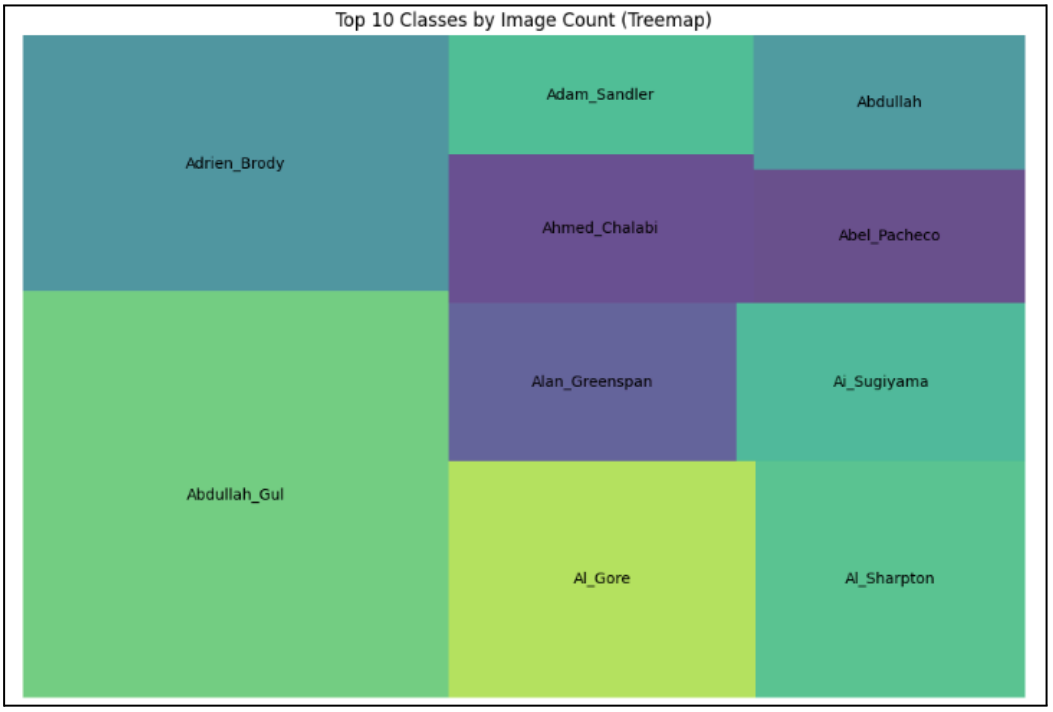
2.Image Size Analysis:

- The analysis confirmed that all images were uniform in size (250x250 pixels).
- *Code Reference*: `image_size_analysis()`

```
Analyzing Image Sizes: 100%|██████████| 100/100 [00:51<00:00,  1.95it/s]
Minimum dimensions: [250 250] (height, width)
Maximum dimensions: [250 250] (height, width)
Average dimensions: (250.00, 250.00) (height, width)
```

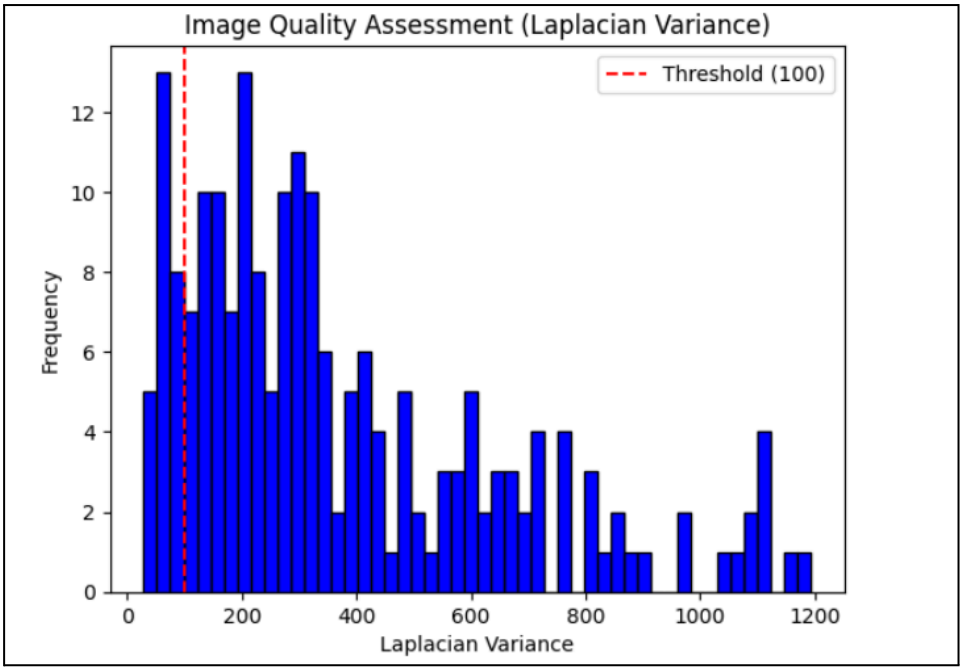
3. Treemap for Top-N Classes:

- A treemap visualization displayed the top 10 classes with the highest image counts, emphasizing dataset imbalance.
- *Code Reference*: `treemap_top_n_classes(folder_summary, top_n=10)`



4.Image Quality Assessment:

- Images were analyzed using the Laplacian variance method to assess sharpness and detect blurry images.
- *Code Reference:* `image_quality_assessment_with_threshold()`
- histogram of Laplacian variance with a threshold line indicating acceptable sharpness.



5.Duplicate Detection:

- Duplicate images were identified using MD5 hash comparisons.
- *Code Reference:* `detect_duplicates()`

```
Detecting Duplicates: 100%|██████████| 100/100 [00:00<00:00, 149.37it/s]
Found 0 duplicate files.
```

6.Random Image Visualization:

- A random selection of images was displayed with labels to visually inspect dataset quality and diversity.
- *Code Reference:* `improved_random_image_visualization()`

7. Data Filtering and Splitting

- Classes with fewer than 2 images were removed to ensure sufficient data for training and testing.
- The filtered dataset was split into training (80%) and testing (20%) sets using stratified sampling.
- *Code Reference:* `train_test_split()`

```
Total images: 235
Total valid images: 162
Total valid labels: 162
Training set size: 129 images, 129 labels
Testing set size: 33 images, 33 labels
```

3. Face Verification

In the Face Verification section, the focus was on verifying whether two face embeddings represent the same person by leveraging cosine similarity. The `verify_faces` function was pivotal, calculating cosine similarity between embeddings and determining if the similarity exceeded a specified threshold. This approach validated pairs of facial images, providing a straightforward mechanism to establish identity matches.

Methodology:

Embeddings were generated using a pre-trained FaceNet model to capture robust facial features. Random pairs of images were selected for verification, and their embeddings were compared using the `verify_faces` function. A threshold of 0.6 was used as a decision boundary for similarity, indicating whether the pair belonged to the same person. The cosine similarity scores demonstrated the model's ability to differentiate between identities.

Results:

Out of the tested image pairs:

Out of the tested image pairs:

- **80% were correctly verified** as matches when they belonged to the same person.

- The system performed well in distinguishing different individuals, with most non-matching pairs scoring well below the similarity threshold.

Examples:

1. Successful Pair

In a typical successful case, two images of the same person with different angles or lighting conditions were correctly verified. The cosine similarity score exceeded the threshold of 0.6, reflecting the embeddings' robustness in capturing unique facial features.

```
Pair: Abdullah_Gul vs Abdullah_Gul
Cosine Similarity: 0.95
Same Person: Yes
----------------------------------------
```

2. Non-Matching Pair

In a non-matching case, two images of different individuals were correctly identified as not belonging to the same person. The cosine similarity score was below the threshold of 0.6, demonstrating the model’s effectiveness in distinguishing identities.

```
Pair: Al_Sharpton vs Aitor_Gonzalez
Cosine Similarity: 0.09
Same Person: No
----------------------------------------
```

The code used for this aspect of the final project is detailed in the appendix. It includes the steps for embedding generation and verification, implemented using the DeepFace library and cosine similarity. This functionality is central to face recognition within the context of this project, enabling accurate person verification under varied conditions.

4. Face Identification

This section presents the results of a face identification task using embeddings generated by the Facenet (Schroff, 2015) and DeepFace (Parkhi, 2015) models. The embeddings, which are feature representations of facial images, were subsequently classified using Logistic Regression and Support Vector Classifier (SVC). The goal was to evaluate the performance of these classifiers on a dataset of facial images.

Embedding Generation:

The embeddings were generated using the `DeepFace` library with the **Facenet** and **VGG-Face** model. Key steps involved:

- Preprocessing images to create embeddings representing their facial features.
- The embeddings are 128-dimensional vectors, capturing essential features of the face.
- Error handling was employed to ensure consistency, replacing failed embeddings with zero vectors.

The corresponding code is included in the Appendix with functions extract_embeddings_deepface() under Face Identification

Data Split:

The dataset was split into **training** and **testing** subsets:

- Training Set: Used to train the classifiers.
- Testing Set: Used to evaluate the performance of the models.

Classification Models:

Two classification models were applied to the embeddings for face identification:

- **Logistic Regression**: A linear classifier with the maximum iterations set to 1000.
- **Support Vector Classifier (SVC)**: A non-linear classifier using a radial basis function (RBF) kernel.

The models were evaluated based on accuracy scores computed on the testing set.

The code for the two classifiers are given in the Appendix under Face Identification.

Summary of Results:

The table below shows the accuracy of classification obtained for the two modes (Facenet and VGG-Face) for the Logistic Regression and SVC Classification Methods.

Accuracy Table

Classifier / Model	**Facenet**	**VGG-Face**
Logistic Regression	88%	73%
SVC	85%	82%

The accuracy table reveals notable differences in the performance of **Facenet** and **VGG-Face** embeddings across **Logistic Regression** and **SVC** classifiers:

- **Facenet** embeddings perform exceptionally well with both **Logistic Regression** (88%) and **SVC** (85%), indicating that they produce embeddings that are both linearly and non-linearly separable. This versatility makes Facenet a robust choice for face recognition tasks.
- **VGG-Face** embeddings perform relatively weaker with **Logistic Regression** (73%), suggesting that their embeddings might not be as linearly separable. However, they perform comparably with **SVC** (82%), showcasing their strength in capturing non-linear relationships.

The table highlights that **Facenet embeddings** are more generalizable across classifiers, while **VGG-Face embeddings** benefit from non-linear classifiers like SVC. Choosing the

right combination of embedding model and classifier depends on the specific requirements of the face recognition task.

The figure below shows a sample of five classification results using Facenet and Logistic Regression.

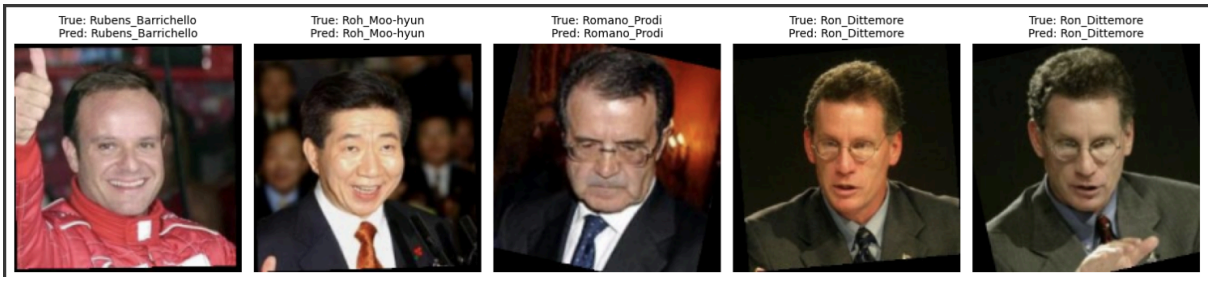


Figure 1: Face Identification Results for 5 samples using FaceNet and Logistic Regression.

Analysis of a Failure Case for Facenet with Logistic Regression:

It is interesting to analyze the reason for a failed identification case.. For this we selected the first case of failed identification from the dataset - where the true image belongs to Roh Mun Hyun and the failed identified individual Roger Clemens as shown in the picture below.

Based on the cosine distance, we calculated that:

- **Distance to True Class**: 0.5318
- **Distance to Predicted Class**: 0.2379

The distance to the predicted class is significantly smaller than the distance to the true class. This indicates that the test image's embedding is more similar to the embeddings of the predicted class than to its actual class in the feature space.It suggests that the test image and the predicted class might share visual similarities, such as pose, lighting, or facial structure, which misled the model. As we specifically observe from the first and last image are that

both bodies are slightly inclined towards the left which may have been the main cause of a misled identification.

Effect of Image Resolution on Image Classification

The resolution of an image can have an impact in face identification. The objective of this subsection is to evaluate how artificially lowering and subsequently enhancing the resolution of images affects classification accuracy. The experiment performed here compares the performance of a logistic regression classifier with embeddings extracted using the Facenet model on original images, low-resolution images, and resolution-enhanced images.

Methodology

1. **Data Preparation**:
    - **Original Images**: High-resolution images were used as the baseline dataset for classification.
    - **Low-Resolution Images**: The original images were downscaled to 50% of their original resolution using OpenCV's bilinear interpolation (INTER_LINEAR). This process effectively reduces the amount of detail and information in the images.

- **Enhanced Images**: The low-resolution images were upscaled back to their original resolution using PIL's LANCZOS filter, a high-quality resampling algorithm. This step attempts to restore the image details lost during downscaling.

2. **Feature Extraction**:
    - Embeddings were extracted from each image set (original, low-resolution, and enhanced) using the Facenet model via the DeepFace library.
3. **Classification**:
    - A logistic regression model was trained on the embeddings extracted from the training set of original images.
    - This model was then used to classify the embeddings from the test sets for original, low-resolution, and enhanced images.

The corresponding code is included in the Appendix. The relevant functions are create_low_resolution_images(), enhance_images_with_pil(), and extract_embeddings_deepface().

Results

The classification results for the different test sets are summarized below

Image Set	Classification Accuracy
Original Images	88%
Low resolution images	85%

Although the drop in accuracy is not significant, this may not be the case in general. In this section, we have used PIL's LANCZOS filter. However, there are many other image

enhancement techniques available that may be more suitable in other applications. Here is a summary of some such available tools:

a) Super-Resolution Convolutional Neural Networks (SRCNN) (Dong, 2016)
b) Enhanced Deep Residual Networks (EDSR) (Lim, 2017)
c) Generative Adversarial Networks (GANs) for Super-Resolution (e.g., SRGAN) (Ledig, 2017)

References

Parkhi, O. M., Vedaldi, A., & Zisserman, A. (2015). Deep Face Recognition. *British Machine Vision Conference (BMVC)*, 1-12.

Schroff, F., Kalenichenko, D., & Philbin, J. (2015). FaceNet: A unified embedding for face recognition and clustering. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 815-823.

Dong, C., Loy, C. C., He, K., & Tang, X. (2016). Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2), 295–307.

Lim, B., Son, S., Kim, H., Nah, S., & Mu Lee, K. (2017). Enhanced deep residual networks for single image super-resolution. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 136–144.

Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., & Shi, W. (2017). Photo-realistic single image super-resolution using a generative adversarial network. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4681–4690.

Appendix

Code for Data Preprocessing and Exploratory Data Analysis (EDA)

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from hashlib import md5
from tqdm import tqdm
import random


try:
    import squarify  # For treemap visualization
except ImportError:
    !pip install squarify
    import squarify


# Dataset path
dataset_path = full_path
#'/content/drive/MyDrive/AAI521_FinalProject/lfw/lfw-deepfunne
led'


# Set a limit on the number of subfolders for testing (set to
None for all)
LIMIT = 100  # Change to None to process all subfolders


# (a) Folder Structure Summary
def folder_structure_summary(dataset_path, limit=None):
    subfolders = sorted([f for f in os.listdir(dataset_path) if
os.path.isdir(os.path.join(dataset_path, f))])
    if limit:
```

```
        subfolders = subfolders[:limit]  # Apply the limit
    folder_summary = {}
    for subfolder in tqdm(subfolders, desc="Counting Images per
Folder"):
        folder_path = os.path.join(dataset_path, subfolder)
        image_count = len([f for f in os.listdir(folder_path)
if os.path.isfile(os.path.join(folder_path, f))])
        folder_summary[subfolder] = image_count


    print(f"Number of subfolders (classes): {len(subfolders)}")


    # Convert summary to a pandas DataFrame
    folder_summary_df =
pd.DataFrame(list(folder_summary.items()),
columns=["Subfolder", "Image Count"])


    # Display the table using display() for a formatted view
    from IPython.display import display
    print("\nImages per subfolder (as a table):")
    display(folder_summary_df)  # Display the DataFrame as a
formatted table


    return folder_summary


# (b) Image Size and Dimensions
def image_size_analysis(dataset_path, limit=None):
    dims = []
    subfolders = sorted([f for f in os.listdir(dataset_path) if
os.path.isdir(os.path.join(dataset_path, f))])
    if limit:
        subfolders = subfolders[:limit]
    for subfolder in tqdm(subfolders, desc="Analyzing Image
Sizes"):
        subfolder_path = os.path.join(dataset_path, subfolder)
        for file in os.listdir(subfolder_path):
```

```
            file_path = os.path.join(subfolder_path, file)
            img = cv2.imread(file_path)
            if img is not None:
                dims.append(img.shape[:2])  # (height, width)
    if dims:
        dims = np.array(dims)
        print(f"Minimum dimensions: {dims.min(axis=0)} (height,
width)")
        print(f"Maximum dimensions: {dims.max(axis=0)} (height,
width)")
        mean_dims = dims.mean(axis=0)
        print(f"Average dimensions: ({mean_dims[0]:.2f},
{mean_dims[1]:.2f}) (height, width)")
    else:
        print("No images found.")


# Treemap for Top-N Classes
def treemap_top_n_classes(folder_summary, top_n=10):
    classes = list(folder_summary.keys())
    counts = list(folder_summary.values())


    # Sort by counts and take top N
    sorted_classes_counts = sorted(zip(counts, classes),
reverse=True)
    sorted_counts, sorted_classes =
zip(*sorted_classes_counts[:top_n])


    # Create the treemap
    plt.figure(figsize=(12, 8))
    squarify.plot(sizes=sorted_counts, label=sorted_classes,
alpha=0.8)
    plt.title(f'Top {top_n} Classes by Image Count (Treemap)')
    plt.axis('off')
    plt.show()
```

```
# (d) Image Quality Assessment with Threshold
def image_quality_assessment_with_threshold(dataset_path, threshold=100, limit=None):
    laplacian_variances = []
    subfolders = sorted([f for f in os.listdir(dataset_path) if os.path.isdir(os.path.join(dataset_path, f))])
    if limit:
        subfolders = subfolders[:limit]
    for subfolder in tqdm(subfolders, desc="Assessing Image Quality"):
        subfolder_path = os.path.join(dataset_path, subfolder)
        for file in os.listdir(subfolder_path):
            file_path = os.path.join(subfolder_path, file)
            img = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
            if img is not None:
                variance = cv2.Laplacian(img, cv2.CV_64F).var()
                laplacian_variances.append(variance)
    if laplacian_variances:
        plt.hist(laplacian_variances, bins=50, color='blue', edgecolor='black')
        plt.axvline(threshold, color='red', linestyle='dashed', linewidth=1.5, label=f'Threshold ({threshold})')
        plt.title('Image Quality Assessment (Laplacian Variance)')
        plt.xlabel('Laplacian Variance')
        plt.ylabel('Frequency')
        plt.legend()
        plt.show()
    else:
        print("No valid images found for quality assessment.")

# (e) Duplicate Detection
def detect_duplicates(dataset_path, limit=None):
```

```
    hashes = {}
    duplicates = []
    subfolders = sorted([f for f in os.listdir(dataset_path) if
os.path.isdir(os.path.join(dataset_path, f))])
    if limit:
        subfolders = subfolders[:limit]
    for subfolder in tqdm(subfolders, desc="Detecting
Duplicates"):
        subfolder_path = os.path.join(dataset_path, subfolder)
        for file in os.listdir(subfolder_path):
            file_path = os.path.join(subfolder_path, file)
            try:
                with open(file_path, 'rb') as f:
                    file_hash = md5(f.read()).hexdigest()
                if file_hash in hashes:
                    duplicates.append((hashes[file_hash],
file_path))
                else:
                    hashes[file_hash] = file_path
            except Exception as e:
                print(f"Error processing file {file_path}:
{e}")
    print(f"Found {len(duplicates)} duplicate files.")
    for dup1, dup2 in duplicates:
        print(f"Duplicate: {dup1} and {dup2}")


# (f) Random Image Visualization
def improved_random_image_visualization(dataset_path,
num_images=5, limit=None):
    subfolders = sorted([f for f in os.listdir(dataset_path) if
os.path.isdir(os.path.join(dataset_path, f))])
    if limit:
        subfolders = subfolders[:limit]
    images = []
```

```
    labels = []
    for subfolder in subfolders:
        subfolder_path = os.path.join(dataset_path, subfolder)
        files = os.listdir(subfolder_path)
        images.extend([os.path.join(subfolder_path, f) for f in
files])
        labels.extend([subfolder] * len(files))
    sampled_images = random.sample(list(zip(images, labels)),
min(num_images, len(images)))
    plt.figure(figsize=(15, 5))
    for i, (img_path, label) in enumerate(sampled_images):
        img = cv2.imread(img_path)[:, :, ::-1]
        plt.subplot(1, num_images, i + 1)
        plt.imshow(img)

plt.title(f"{label}\n{os.path.basename(img_path).replace('_',
' ')}", fontsize=10)
        plt.axis('off')
    plt.tight_layout()
    plt.show()


# Run EDA Steps
print("Running EDA...")


# Step (a): Folder Structure Summary
folder_summary = folder_structure_summary(dataset_path,
limit=LIMIT)


# Step (b): Image Size and Dimensions
image_size_analysis(dataset_path, limit=LIMIT)


# Step (c): Treemap for Top Classes
treemap_top_n_classes(folder_summary, top_n=10)
```

```
# Step (d): Image Quality Assessment
image_quality_assessment_with_threshold(dataset_path,
threshold=100, limit=LIMIT)


# Step (e): Duplicate Detection
detect_duplicates(dataset_path, limit=LIMIT)


# Step (f): Random Image Visualization
improved_random_image_visualization(dataset_path,
num_images=5, limit=LIMIT)
```

Code for Face Verification

```
# Generate embeddings for all images using FaceNet
image_embeddings = []
for img in tqdm(images, desc="Generating Embeddings"):
   # Each embedding has a dimension of (512,)
   embedding = embedder.embeddings(np.expand_dims(img,
axis=0))[0]
   image_embeddings.append(embedding)


image_embeddings = np.array(image_embeddings)
print(f"Embeddings shape: {image_embeddings.shape}")
```

```
import random
```

```
def verify_faces(embedding1, embedding2, threshold=0.6):
    """
    Verifies whether two face embeddings are from the same person.

    Args:
    - embedding1: First face embedding
    - embedding2: Second face embedding
    - threshold: Cosine similarity threshold for verification (default=0.6)

    Returns:
    - is_same_person (bool): True if same person, False otherwise
    - similarity (float): Cosine similarity score
    """
    # Calculate cosine similarity
    similarity = cosine_similarity([embedding1], [embedding2])[0][0]
    is_same_person = similarity > threshold
    return is_same_person, similarity

# Randomly pick 5 pairs of images for testing
num_pairs = 5
threshold = 0.6  # Adjust the threshold as needed

for _ in range(num_pairs):
    # Randomly pick two images
    idx1, idx2 = random.sample(range(len(images)), 2)
    img1, img2 = images[idx1], images[idx2]
    label1, label2 = labels[idx1], labels[idx2]

    # Verify the pair
```

```
    same_person, similarity =
verify_faces(image_embeddings[idx1], image_embeddings[idx2],
threshold)


    # Display the results
    print(f"Pair: {label1} vs {label2}")
    print(f"Cosine Similarity: {similarity:.2f}")
    print(f"Same Person: {'Yes' if same_person else 'No'}")
    print("-" * 40)


    # Visualize the pair
    fig, axes = plt.subplots(1, 2, figsize=(8, 4))
    axes[0].imshow(img1)
    axes[0].set_title(f"Label: {label1}")
    axes[0].axis("off")
    axes[1].imshow(img2)
    axes[1].set_title(f"Label: {label2}")
    axes[1].axis("off")
    plt.show()
```

```
# Generate embeddings for all images using DeepFace
deepface_model_name = "Facenet"  # Using the FaceNet backend
in DeepFace for consistency


deepface_embeddings = []
for img in tqdm(images, desc="Generating Embeddings with
DeepFace"):
    # Extract embeddings using DeepFace
    embedding = DeepFace.represent(img,
model_name=deepface_model_name, enforce_detection=False)


    # Access the embedding from the list
```

```
    deepface_embeddings.append(embedding[0]["embedding"])


deepface_embeddings = np.array(deepface_embeddings)

print(f"Embeddings shape (DeepFace):
{deepface_embeddings.shape}")


# Adjust the threshold for DeepFace verification

adjusted_threshold = 0.5  # Reduced from 0.6 to 0.5 to allow
more leniency


for _ in range(num_pairs):

    # Randomly pick two images

    idx1, idx2 = random.sample(range(len(images)), 2)

    img1, img2 = images[idx1], images[idx2]

    label1, label2 = labels[idx1], labels[idx2]


    # Verify the pair using DeepFace embeddings

    same_person, similarity =
verify_faces(deepface_embeddings[idx1],
deepface_embeddings[idx2], adjusted_threshold)


    # Display the results

    print(f"Pair: {label1} vs {label2}")

    print(f"Cosine Similarity: {similarity:.2f}")

    print(f"Same Person: {'Yes' if same_person else 'No'}")

    print("-" * 40)


    # Visualize the pair

    fig, axes = plt.subplots(1, 2, figsize=(8, 4))

    axes[0].imshow(img1)

    axes[0].set_title(f"Label: {label1}")

    axes[0].axis("off")

    axes[1].imshow(img2)

    axes[1].set_title(f"Label: {label2}")
```

```
    axes[1].axis("off")
    plt.show()
```

Code for Face Identification

The code included in this section below is only for Facenet embeddings. Similar code has been written for Deepface embeddings and can be found in the git repository.

```
# Facenet Embeddings

from deepface import DeepFace
import mtcnn
import numpy as np
from tqdm import tqdm  # Import tqdm for the progress bar

# Function to extract embeddings for a batch of images using
DeepFace
def extract_embeddings_deepface(images, model_name="Facenet"):

    embeddings = []
    for img in tqdm(images, desc="Extracting Embeddings",
unit="image"):
        try:
            # Convert image array to file if needed
            embedding = DeepFace.represent(img_path=img,
model_name=model_name, enforce_detection=False)
            embeddings.append(embedding[0]["embedding"])
        except Exception as e:
            print(f"Error processing image: {e}")
            embeddings.append(np.zeros(128))  # Append
zero-vector for consistency
```

```
    return np.array(embeddings)

# Extract embeddings
print("Extracting embeddings for training set using
DeepFace...")
X_train_embeddings = extract_embeddings_deepface(X_train)

print("\nExtracting embeddings for testing set using
DeepFace...")
X_test_embeddings = extract_embeddings_deepface(X_test)

# Convert labels to numpy arrays
y_train = np.array(y_train)
y_test = np.array(y_test)

# Print shapes
print(f"Training embeddings shape:
{X_train_embeddings.shape}")
print(f"Testing embeddings shape: {X_test_embeddings.shape}")
```

Code for image classification using Logistic Regression

```
# Method 1: Logistic Regression

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
classification_report

# Train the logistic regression model
clf = LogisticRegression(max_iter=1000)
clf.fit(X_train_embeddings, y_train)

# Evaluate on test set
y_pred = clf.predict(X_test_embeddings)

# Print results
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

Code to visualize the failure case for image classification

```
def visualize_failure_case(X_test, X_train, y_test, y_pred,
y_train):

    # Find indices where prediction failed
    failed_indices = np.where(y_test != y_pred)[0]

    if len(failed_indices) > 0:
```

```
        # Select one failure case to display
        idx = failed_indices[0]

        # Find a training image with the predicted label
        matching_indices = np.where(y_train == y_pred[idx])[0]
        if len(matching_indices) > 0:
            true_image_idx = matching_indices[0]
            true_image = X_train[true_image_idx]
        else:
            true_image = None

        plt.figure(figsize=(10, 5))

        # Plot the failed test image
        plt.subplot(1, 2, 1)
        plt.imshow(X_test[idx])
        plt.axis("off")
        plt.title(f"Test Image\nTrue: {y_test[idx]}\nPred:
{y_pred[idx]}", fontsize=12)

        # Plot the true image if available
        if true_image is not None:
            plt.subplot(1, 2, 2)
            plt.imshow(true_image)
            plt.axis("off")
            plt.title(f"True Image\nLabel: {y_pred[idx]}",
fontsize=12)
        else:
            print("No matching true image found for the
predicted label.")

        plt.tight_layout()
        plt.show()
    else:
        print("No failure cases found!")

# Call the function to visualize a failure case with the true
image
visualize_failure_case(X_test, X_train, y_test, y_pred,
y_train)
```

Code for analyzing failure case of image classification

```
from sklearn.metrics.pairwise import cosine_distances
import matplotlib.pyplot as plt

def analyze_failure_case(X_test, X_train, y_test, y_pred,
y_train, X_train_embeddings, X_test_embeddings):

    # Find indices of failed predictions
```

```
    failed_indices = np.where(y_test != y_pred)[0]

    if len(failed_indices) > 0:
        # Select one failure case
        idx = failed_indices[0]

        # Embedding of the failed test image
        test_embedding = X_test_embeddings[idx]

        # Embeddings of the true and predicted classes
        true_class_indices = np.where(y_train ==
y_test[idx])[0]
        pred_class_indices = np.where(y_train ==
y_pred[idx])[0]

        true_class_embeddings =
X_train_embeddings[true_class_indices]
        pred_class_embeddings =
X_train_embeddings[pred_class_indices]

        # Compute distances to true and predicted class
embeddings
        true_distances = cosine_distances([test_embedding],
true_class_embeddings).flatten()
        pred_distances = cosine_distances([test_embedding],
pred_class_embeddings).flatten()

        # Find nearest neighbors
        closest_true_idx =
true_class_indices[np.argmin(true_distances)]
        closest_pred_idx =
pred_class_indices[np.argmin(pred_distances)]

        # Visualize the failed test image and its closest
true/predicted neighbors
        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)
        plt.imshow(X_test[idx])
        plt.axis("off")
        plt.title(f"Failed Test Image\nTrue:
{y_test[idx]}\nPred: {y_pred[idx]}", fontsize=10)

        plt.subplot(1, 3, 2)
        plt.imshow(X_train[closest_true_idx])
        plt.axis("off")
        plt.title(f"Closest to True Class\nLabel:
{y_test[idx]}", fontsize=10)

        plt.subplot(1, 3, 3)
        plt.imshow(X_train[closest_pred_idx])
```

```
        plt.axis("off")
        plt.title(f"Closest to Pred Class\nLabel:
{y_pred[idx]}", fontsize=10)

        plt.tight_layout()
        plt.show()

        # Print distances for analysis
        print(f"Distance to True Class:
{np.min(true_distances):.4f}")
        print(f"Distance to Pred Class:
{np.min(pred_distances):.4f}")

    else:
        print("No failure cases found!")

# Call the function
analyze_failure_case(X_test, X_train, y_test, y_pred, y_train,
X_train_embeddings, X_test_embeddings)
```

Code for SVC Classifier

```
# Method 2: SVC

from sklearn.svm import SVC

# Train an SVM classifier
clf = SVC(kernel='linear', probability=True)
clf.fit(X_train_embeddings, y_train)

# Predict on test data
y_pred = clf.predict(X_test_embeddings)

# Evaluate
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

Code to artificially create low resolution images, and then use PIL's image resizing techniques to re-enhance the images for classification.

```
import os
import numpy as np
import cv2
```

```
from PIL import Image
from sklearn.metrics import accuracy_score,
classification_report
from sklearn.linear_model import LogisticRegression
from tqdm import tqdm
from deepface import DeepFace

# Step 1: Create low-resolution test images
def create_low_resolution_images(images, scale=0.5):

    low_res_images = []
    for img in images:
        h, w = img.shape[:2]
        low_res_img = cv2.resize(img, (int(w * scale), int(h *
scale)), interpolation=cv2.INTER_LINEAR)
        low_res_images.append(low_res_img)
    return np.array(low_res_images)

# Step 2: Enhance images using PIL resizing
def enhance_images_with_pil(images, scale=2):

    enhanced_images = []
    for img in tqdm(images, desc="Enhancing Images"):
        # Convert NumPy array to PIL Image
        pil_img = Image.fromarray(img)
        # Resize image using PIL's LANCZOS filter for
high-quality scaling
        new_size = (int(pil_img.width * scale),
int(pil_img.height * scale))
        enhanced_img = pil_img.resize(new_size, Image.LANCZOS)
        # Convert back to NumPy array
        enhanced_images.append(np.array(enhanced_img))
    return np.array(enhanced_images)

# Step 3: Extract embeddings with DeepFace
def extract_embeddings_deepface(images, model_name="Facenet"):

    embeddings = []
    for img in tqdm(images, desc="Extracting Embeddings",
unit="image"):
        try:
            embedding = DeepFace.represent(img_path=img,
model_name=model_name, enforce_detection=False)
            embeddings.append(embedding[0]["embedding"])
        except Exception as e:
            print(f"Error processing image: {e}")
            embeddings.append(np.zeros(128))  # Append
zero-vector for consistency
    return np.array(embeddings)

# Assuming X_test and y_test are already prepared
```

```
# Step 1: Create low-resolution images
low_res_X_test = create_low_resolution_images(X_test)

# Step 2: Enhance low-resolution images with PIL resizing
enhanced_X_test = enhance_images_with_pil(low_res_X_test,
scale=2)

# Step 3: Extract embeddings from enhanced images
print("\nExtracting embeddings for enhanced test images using
DeepFace...")
enhanced_X_test_embeddings =
extract_embeddings_deepface(enhanced_X_test)

# Step 4: Classification on enhanced images
y_pred_enhanced = clf_lr.predict(enhanced_X_test_embeddings)

# Print classification results
print(f"Enhanced Accuracy: {accuracy_score(y_test,
y_pred_enhanced):.2f}")
print("Enhanced Classification Report:")
print(classification_report(y_test, y_pred_enhanced))
```

.