EE5907 Pattern Recognition: Assignment #2
XU NUO, A0313771H

Part 1. Feature Extraction with PCA and LDA

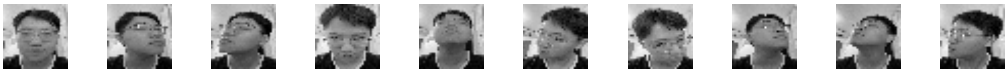
1. Data Preparation

I downloaded the CMU PIE dataset from Canvas, which contains 68 different classes, each with 170 portrait images captured under varying poses and lighting conditions. My metric number is A0313771H, and the las two digits are 71. Therefore, I used `random.seed(71)` and `random.sample(range(1, 69), 25)` to randomly select 25 classes from the dataset. After taking 10 selfies, I converted them to grayscale and resized them to 32*32 pixels to ensure consistency with CMU PIE images.

To maintain consistency in the code, I created a folder named "69" within the PIE dataset to store the preprocessed selfies (grayscaled and resized). Thus, class "69" corresponds to the selfie class. After randomly selecting 25 classes, I appended the selfie class to the list, resulting in a total of 26 classes. Upon running the code, the following 26 classes were selected:

`[42, 66, 2, 34, 20, 53, 49, 13, 17, 6, 61, 33, 4, 39, 58, 57, 67, 45, 65, 24, 30, 9, 23, 68, 11, 69]`

The selfie photos after preprocessing are shown below:



The relevant code is as follows:

selfie_prep.py

```
import os
import cv2

# ======= Process the Selfie Dataset =======
def process_selfie(root_path, target_path):
    selfie_files = sorted([
        file for file in os.listdir(root_path)
        if file.endswith(".jpg")
    ]) # Get all the sorted image files (according to their indices) in the folder

    for file in selfie_files:
        file_path = os.path.join(root_path, file) # Concatenate the file path
        img_current = cv2.imread(file_path) # Read the image
        img_current = cv2.cvtColor(img_current, cv2.COLOR_BGR2GRAY) # Convert the image
to grayscale
```

```
        img_current = cv2.resize(img_current, (32, 32)) # Resize the image to 32x32
        cv2.imwrite(os.path.join(target_path, file), img_current) # Save the processed
image
```

images_prep.py

```
import os
import cv2

# ======= Get Images from the Dataset =======
def get_images(root_path, selected_idx):
    images_training = []
    labels_training = []
    images_test = []
    labels_test = []

    # Iterate through each person folder, skip if not in selected_idx
    for person_idx in sorted(os.listdir(root_path)):
        if int(person_idx) not in selected_idx:
            continue

        person_folder = os.path.join(root_path, person_idx) # Contactenate the folder
path

        image_files = sorted([
            file for file in os.listdir(person_folder)
            if file.endswith(".jpg")
        ]) # Get all the sorted image files (according to their indices) in the folder

        total_images = len(image_files) # Get the total number of images in the folder
        train_count = int(0.7 * total_images) # 70% of the images are used for training,
the rest for testing

        for file in image_files[:train_count]: # Indices from 0 to train_count - 1
            file_path = os.path.join(person_folder, file) # Concatenate the file path
            img_current = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE) # Read the image in
grayscale
            images_training.append(img_current)# Append the image to the list
            labels_training.append(int(person_idx)) # Append the corresponding label to
the list

        for file in image_files[train_count:]:
            file_path = os.path.join(person_folder, file)
            img_current = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
            images_test.append(img_current)
```

```
            labels_test.append(int(person_idx))

    return images_training, labels_training, images_test, labels_test
```

To test the classification performance under different random seeds, my code is design to load all selected images directly into memory at runtime, rather than writing the selected images to a specific directory for later use. Therefore, the subsequent code typically includes the following section to enable the functionality of selecting and loading the images:

```
# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset
```

2. PCA for feature extraction and visualization

a. PCA from scratch implementation

Since each image has a size of 32*32, it is reshaped into a 1024-dimensional vector before processing. The PCA function takes a matrix `X` and a number `p` as input, where each row of matrix X represents the 1024 pixels of an image, and p indicates the number of principal components to retain. The function ultimately outputs the mean pixel image `X_mean`, the projection matrix `projection_matrix`, and the projected images `X_pca`. The manually implemented PCA function is shown below:

pca_from_scratch.py

```
import numpy as np

# ======= Perform PCA from Scratch =======
def mannual_pca(X, p):

    X_mean = np.mean(X, axis=0) # Calculate the mean of the pixels at the same position
of all images
    X_centered = X - X_mean # Subtract the mean from the images (Centralization)
    cov_matrix = np.cov(X_centered, rowvar=False, bias = True) # Calculate the covariance
matrix (rowvar=False for each column to represent a variable) (bias = True for biased
estimation)
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix) # Calculate the eigenvalues
and eigenvectors (eigh for symmetric matrix)
```

```
    sort_eigen_idx = np.argsort(eigenvalues)[::-1] # Sort the eigenvalues in descending
order
    eigenvalues = eigenvalues[sort_eigen_idx] # Sort the eigenvalues
    eigenvectors = eigenvectors[:, sort_eigen_idx] # Sort the eigenvectors
    projection_matrix = eigenvectors[:, :p] # Select the first p eigenvectors as the
projection matrix
    X_pca = X_centered @ projection_matrix # Project the images to the new space

    return X_mean, projection_matrix, X_pca # Return the mean (for reconstruction),
projection matrix, and the projected images
```

b. PCA with p=2/3 visualization

The eigenfaces obtained after reducing the dimensionality of the training images to 2D and 3D using PCA are shown in Figure 1 and Figure 2. It can be observed that the eigenface corresponding to the first principal component captures the most significant variance and represents the overall facial structure. The second eigenface reflects variations caused by lighting conditions, while the third eigenface captures the impact of facial orientation on facial features.

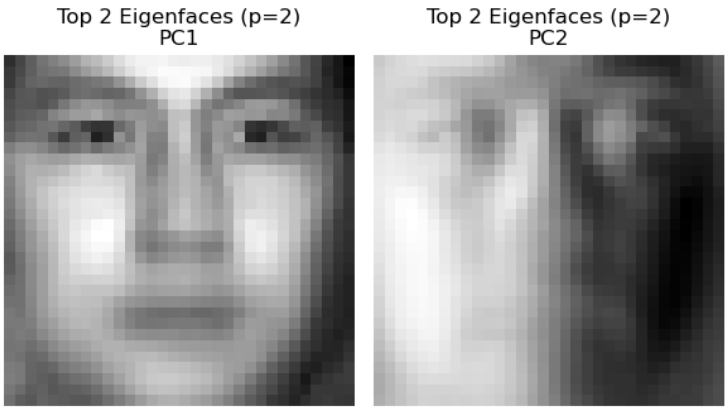


Fig. 1. Top 2 Eigenfaces (p=2)

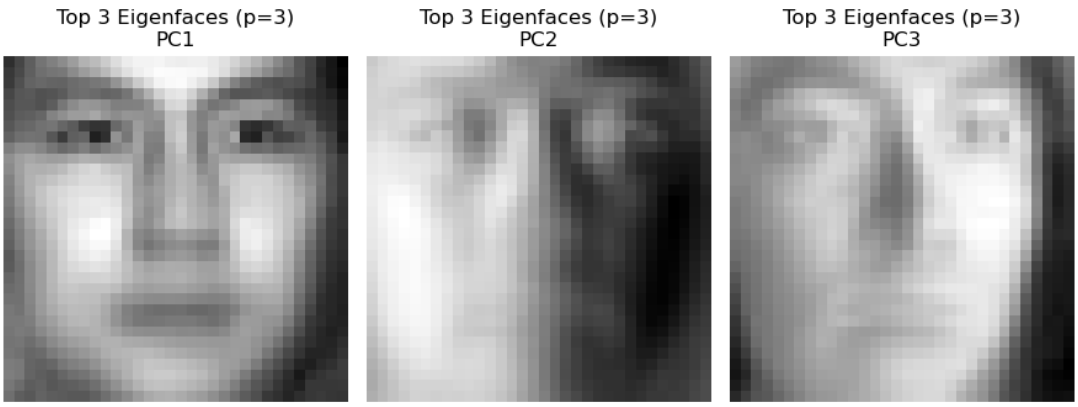


Fig. 2. Top 3 Eigenfaces (p=3)

Figures 3 and 4 show the projections of the training data in 2D and 3D space after dimensionality reduction using PCA. In the figures, the colored dots represent images from different classes, while the red stars indicate the positions of the selfie images in the projected space. It can be observed that most data points form a “cloud-like” distribution with significant overlap and unclear boundaries between classes, indicating that the first two or three principal components are insufficient for effective class separation. However, the red stars are well-clustered in both the 2D and 3D projections. I believe this is because my selfie images differ noticeably from the CMU PIE dataset images in terms of background, angle, and lighting, which allows the model to distinguish my selfies from the dataset images relatively well.

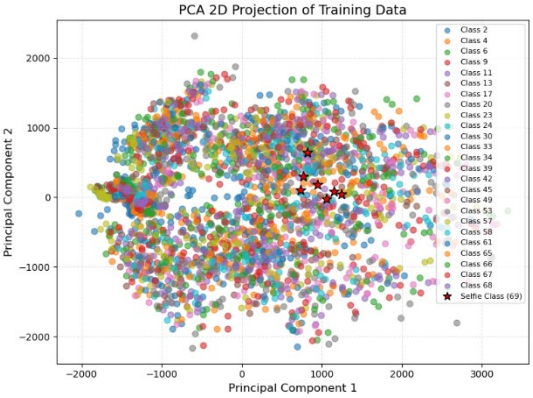


Fig. 3. PCA 2D projection (p=2)

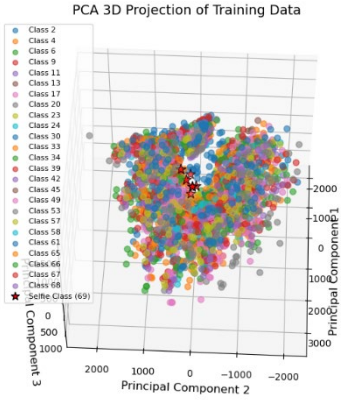


Fig. 4. PCA 3D projection (p=3)

The relevant code is as follows:

pca_2_3_visualization.py

```
import images_prep
import selfie_prep
import pca_from_scratch
import numpy as np
```

```
import matplotlib.pyplot as plt
import random

# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset
selfie_path = "./selfie_original" # Path to selfie
selfie_processed_path = "./PIE/69" # Path to processed PIE dataset

selfie_prep.process_selfie(selfie_path, selfie_processed_path) # Process the selfie
dataset (convert to grayscale, resize to 32x32)
training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array

X_mean_2, projection_matrix_2, X_pca_2 = pca_from_scratch.mannual_pca(X_train, 2) #
Perform PCA on the training images with 2 components
X_mean_3, projection_matrix_3, X_pca_3 = pca_from_scratch.mannual_pca(X_train, 3) #
Perform PCA on the training images with 3 components

# ======= 2D PCA Visualization =======
plt.figure(figsize=(8, 6)) # Set the figure size
classes = np.unique(Y_train) # Get the unique classes from the training labels, discard
the duplicates

for cls in classes:
    idx = (Y_train == cls) # Create a boolean array of the same length as Y_train, True
if the class is cls, False otherwise
    if cls == 69: # Selfie class is plotted as red stars
        plt.scatter(X_pca_2[idx, 0], X_pca_2[idx, 1],
                    color='red', marker='*', s=120, edgecolors='black',
                    label=f"Selfie Class ({cls})")
    else: # Other classes are plotted as circles
        plt.scatter(X_pca_2[idx, 0], X_pca_2[idx, 1],
```

```
                    alpha=0.6, s=40, label=f"Class {cls}")

plt.title("PCA 2D Projection of Training Data", fontsize=14)
plt.xlabel("Principal Component 1", fontsize=12)
plt.ylabel("Principal Component 2", fontsize=12)
plt.grid(True, linestyle="--", alpha=0.3)
plt.legend(fontsize=8, markerscale=0.8, loc='best')
plt.tight_layout()
plt.show(block=False)

# ======= 3D PCA Visualization =======
plt.figure(figsize=(8, 6)) # Set the figure size
ax = plt.axes(projection='3d') # Create a 3D plot

for cls in classes:
    idx = (Y_train == cls) # Create a boolean array of the same length as Y_train, True
if the class is cls, False otherwise
    if cls == 69: # Selfie class is plotted as red stars
        ax.scatter(X_pca_3[idx, 0], X_pca_3[idx, 1], X_pca_3[idx, 2],
                   color='red', marker='*', s=120, edgecolors='black',
                   label=f"Selfie Class ({cls})")
    else: # Other classes are plotted as circles
        ax.scatter(X_pca_3[idx, 0], X_pca_3[idx, 1], X_pca_3[idx, 2],
                   alpha=0.6, s=40, label=f"Class {cls}")

ax.set_title("PCA 3D Projection of Training Data", fontsize=14)
ax.set_xlabel("Principal Component 1", fontsize=12)
ax.set_ylabel("Principal Component 2", fontsize=12)
ax.set_zlabel("Principal Component 3", fontsize=12)
ax.grid(True, linestyle="--", alpha=0.3)
ax.legend(fontsize=8, markerscale=0.8, loc='best')
plt.tight_layout()
plt.show(block=False)

# ======= Eigenfaces Visualization =======
def plot_eigenfaces(projection_matrix, count, title_prefix="PCA Eigenfaces"):
    plt.figure(figsize=(3 * count, 4)) # Set the figure size
    for i in range(count):
        eigenface = projection_matrix[:, i].reshape(32, 32) # Reshape the eigenface to
32x32, evry column is an eigenface (1024, i)
        plt.subplot(1, count, i + 1)
        plt.imshow(eigenface, cmap='gray')
        plt.title(f"{title_prefix}\nPC{i + 1}", fontsize=12)
        plt.axis('off')
```

```
    plt.tight_layout()
    plt.show(block=False)

plot_eigenfaces(projection_matrix_2, 2, title_prefix="Top 2 Eigenfaces (p=2)")
plot_eigenfaces(projection_matrix_3, 3, title_prefix="Top 3 Eigenfaces (p=3)")
plt.show() # Display all the plots
```

c. PCA with p=80/200 preparation for Part 2

First, photos from different classes are randomly selected, and the functions pre-defined in *images_prep.py* are used to extract the training and testing images along with their corresponding labels. Then, each image is flattened into a 1024-dimensional vector. Using a self-implemented PCA function, the image mean, projection matrix, and projected images are obtained for p=80 and 200. The test images are then mean-centered using the training image mean, and dimensionality reduction is performed using the projection matrix derived from the training set.

The relevant code section is as follows:

```
# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array

X_mean_80, projection_matrix_80, X_pca_80 = pca_from_scratch.mannual_pca(X_train, 80) #
Perform PCA on the training images with 80 components
X_mean_200, projection_matrix_200, X_pca_200 = pca_from_scratch.mannual_pca(X_train,
200) # Perform PCA on the training images with 200 components

X_test_pca_80 = (X_test - X_mean_80) @ projection_matrix_80 # Project the test images to
the new space (centralization using the mean of training images)
X_test_pca_200 = (X_test -  X_mean_200) @ projection_matrix_200
```

3. LDA for feature extraction and visualization

a. LDA from scratch implementation

According to the definitions provided in the slides, I defined the total mean vector μ, class-specific mean vector μ_i, class-specific covariance matrix S_i and scatter matrix $\widehat{S_i}$, within-class covariance matrix S_w and scatter matrix $\widehat{S_w}$, between-class covariance matrix S_B and scatter matrix $\widehat{S_B}$, and total covariance matrix S_T and scatter matrix $\widehat{S_T}$.
The LDA function takes the data matrix `X` and a number `p` as input. Since LDA is a supervised learning method, label `Y` is also one of the inputs. The function ultimately outputs the projection matrix `projection_matrix`, and the projected images `X_lda`. The manually implemented LDA function is shown below:

lda_from_scratch.py

```
import numpy as np

# ======= Perform LDA from Scratch =======
def mannual_lda(X, Y, p):
    classes = np.unique(Y) # Get the unique classes from the labels
    class_mean = np.array([np.mean(X[Y == cls], axis=0) for cls in classes]) # Calculate
the mean of each class
    X_mean = np.mean(X, axis=0) # Calculate the overall mean

    S_i_list = [np.cov(X[Y == cls], rowvar=False, bias = True) for cls in classes] #
Calculate the class-specific covariance matrix (bias = True for biased estimation)
    S_i_hat_list = []
    for cls, S_i in zip(classes, S_i_list): # Iterate through each class and its
corresponding scatter matrix (zip to Pack two lists into pairs by position)
        N_i = len(X[Y == cls]) # Get the number of samples in the class
        S_i_hat = N_i * S_i # Calculate the class-specific scatter matrix
        S_i_hat_list.append(S_i_hat) # Append the scatter matrix to the list

    n_i_list = [len(X[Y == cls]) for cls in classes] # Get the number of samples in each
class
    N = sum(n_i_list) # Get the total number of samples
    S_w = np.zeros_like(S_i_list[0]) # Initialize the within-class covariance matrix
    for n_i, S_i in zip(n_i_list, S_i_list):
        P_i = n_i / N # Calculate the class-specific probability
        S_w += P_i * S_i # Calculate the within-class covariance matrix

    S_w_hat = np.sum(S_i_hat_list, axis=0) # Calculate the total within-class scatter
matrix
```

```
    S_b = np.zeros((X.shape[1], X.shape[1])) # Initialize the between-class covariance
matrix
    S_b_hat = np.zeros_like(S_b) # Initialize the between-class scatter matrix

    for n_i, mean_i in zip(n_i_list, class_mean):
        mean_diff = (mean_i - X_mean).reshape(-1, 1)
        S_b_hat += n_i * (mean_diff @ mean_diff.T)
        S_b += (n_i / N) * (mean_diff @ mean_diff.T)

    # S_T = S_w + S_b # Calculate the total covariance matrix
    # S_T_hat = S_w_hat + S_b_hat # Calculate the total scatter matrix

    # eigenvalues, eigenvectors = np.linalg.eigh(np.linalg.pinv(S_w_hat) @ S_b_hat) #
Calculate the eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.pinv(S_w_hat) @ S_b_hat) #
Calculate the eigenvalues and eigenvectors
    eigenvalues = eigenvalues.real
    eigenvectors = eigenvectors.real

    sort_eigen_idx = np.argsort(eigenvalues)[::-1] # Sort the eigenvalues in descending
order
    eigenvalues = eigenvalues[sort_eigen_idx] # Sort the eigenvalues
    eigenvectors = eigenvectors[:, sort_eigen_idx] # Sort the eigenvectors
    projection_matrix = eigenvectors[:, :p] # Select the first p eigenvectors as the
projection matrix
    X_lda = X @ projection_matrix # Project the images to the new space

    return projection_matrix, X_lda # Return the projection matrix and the projected
images[]
```

b. LDA with p=2/3 visualization

Fisherfaces are the projection vectors of LDA, each representing a feature direction that best separates the classes. Figures 5 and 6 show the fisherfaces obtained by applying LDA to reduce the training data to 2D and 3D, respectively. Among them, LD1 in both p=2 and p=3 exhibits relatively clear boundary structures, representing the direction with the largest between-class variation. LD2 and LD3 capture more detailed discriminatory directions, with more complex and texture-like patterns. Unlike eigenfaces generated by PCA, LDA is not intended for image reconstruction but rather for enhancing between-class separability. Therefore, these images may not exhibit clear facial structures, but they encode the directions with the strongest discriminative information.

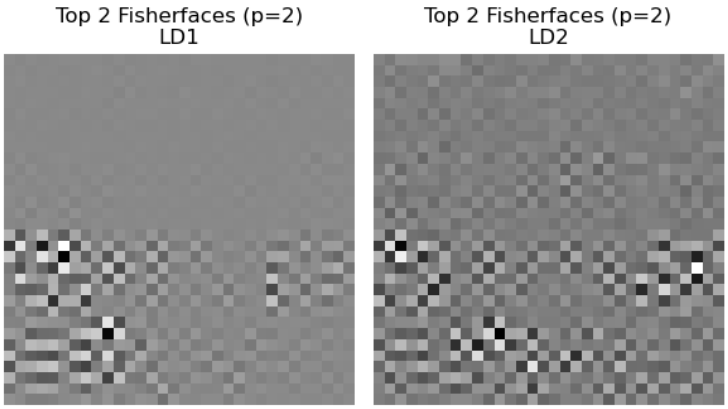


Fig. 5. Top 2 Fisherfaces (p=2)

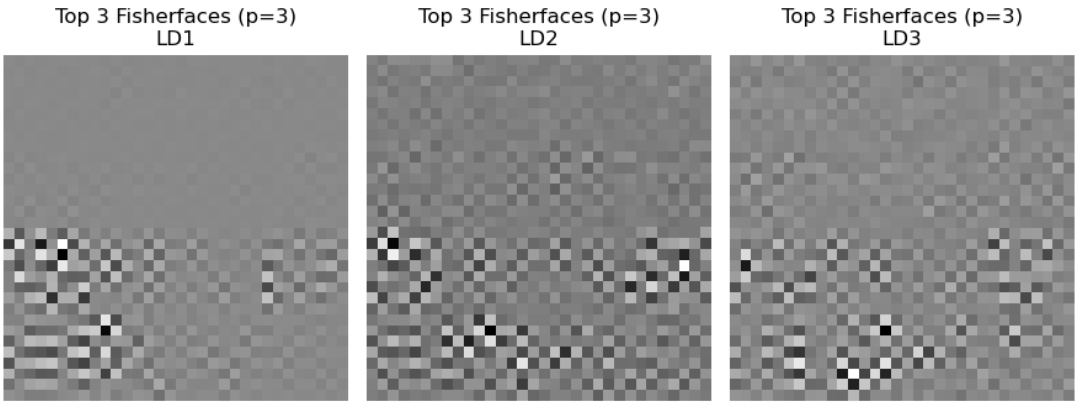


Fig. 6. Top 3 Fisherfaces (p=3)

Similar to the method of dimensionality reduction using PCA and plotting the projection results, Figures 7 and 8 show the projections of the training data in 2D and 3D space after dimensionality reduction using LDA. In Figure 7, all non-selfie classes are compressed into an approximately "vertical line," while the selfie class is distributed separately on the left side of the plot. This indicates that LDA successfully found a direction that clearly separates the selfie class from other classes, though its ability to distinguish among the other classes is limited. In the 3D space, the data points of different classes are more dispersed compared to the 2D case, forming “clusters” with greater spacing between classes and clearer inter-class boundaries.

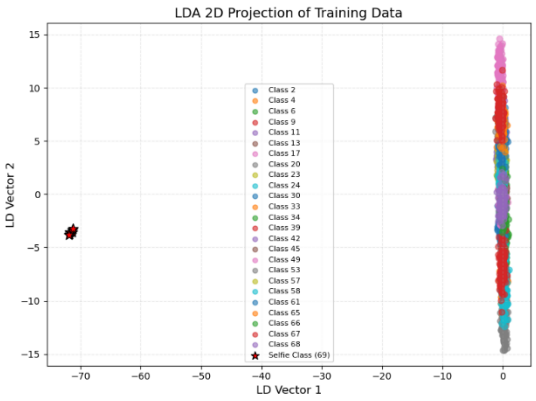


Fig. 7. LDA 2D projection (p=2)

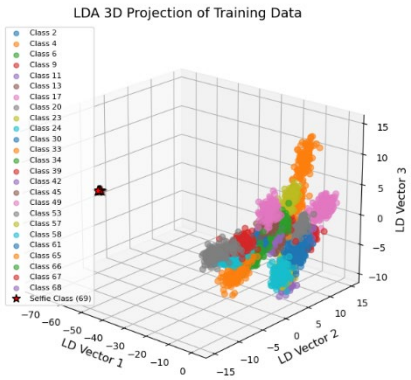


Fig. 8. LDA 3D projection (p=3)

The relevant code is as follows:

lda_2_3_visualization.py

```
import images_prep
import lda_from_scratch
import numpy as np
import matplotlib.pyplot as plt
import random

# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array

projection_matrix_2, X_lda_2 = lda_from_scratch.mannual_lda(X_train, Y_train, 2) #
Perform LDA on the training images with 2 components
projection_matrix_3, X_lda_3 = lda_from_scratch.mannual_lda(X_train, Y_train, 3) #
Perform LDA on the training images with 3 components
```

```
# ======= 2D LDA Visualization =======
plt.figure(figsize=(8, 6)) # Set the figure size
classes = np.unique(Y_train) # Get the unique classes from the training labels, discard
the duplicates

for cls in classes:
    idx = (Y_train == cls) # Create a boolean array of the same length as Y_train, True
if the class is cls, False otherwise
    if cls == 69: # Selfie class is plotted as red stars
        plt.scatter(X_lda_2[idx, 0], X_lda_2[idx, 1],
                    color='red', marker='*', s=120, edgecolors='black',
                    label=f"Selfie Class ({cls})")
    else: # Other classes are plotted as circles
        plt.scatter(X_lda_2[idx, 0], X_lda_2[idx, 1],
                    alpha=0.6, s=40, label=f"Class {cls}")

plt.title("LDA 2D Projection of Training Data", fontsize=14)
plt.xlabel("LD Vector 1", fontsize=12)
plt.ylabel("LD Vector 2", fontsize=12)
plt.grid(True, linestyle="--", alpha=0.3)
plt.legend(fontsize=8, markerscale=0.8, loc='best')
plt.tight_layout()
plt.show(block=False)

# ======= 3D LDA Visualization =======
plt.figure(figsize=(8, 6)) # Set the figure size
ax = plt.axes(projection='3d') # Create a 3D plot

for cls in classes:
    idx = (Y_train == cls) # Create a boolean array of the same length as Y_train, True
if the class is cls, False otherwise
    if cls == 69: # Selfie class is plotted as red stars
        ax.scatter(X_lda_3[idx, 0], X_lda_3[idx, 1], X_lda_3[idx, 2],
                    color='red', marker='*', s=120, edgecolors='black',
                    label=f"Selfie Class ({cls})")
    else: # Other classes are plotted as circles
        ax.scatter(X_lda_3[idx, 0], X_lda_3[idx, 1], X_lda_3[idx, 2],
                    alpha=0.6, s=40, label=f"Class {cls}")

ax.set_title("LDA 3D Projection of Training Data", fontsize=14)
ax.set_xlabel("LD Vector 1", fontsize=12)
ax.set_ylabel("LD Vector 2", fontsize=12)
ax.set_zlabel("LD Vector 3", fontsize=12)
ax.grid(True, linestyle="--", alpha=0.3)
```

```
ax.legend(fontsize=8, markerscale=0.8, loc='best')
plt.tight_layout()
plt.show(block=False)

# ======= Eigenfaces Visualization =======
def plot_eigenfaces(projection_matrix, count, title_prefix="Fisherfaces"):
    plt.figure(figsize=(3 * count, 4)) # Set the figure size
    for i in range(count):
        eigenface = projection_matrix[:, i].reshape(32, 32) # Reshape the fisherfaces to
32x32, evry column is an eigenface (1024, i)
        plt.subplot(1, count, i + 1)
        plt.imshow(eigenface, cmap='gray')
        plt.title(f"{title_prefix}\nLD{i + 1}", fontsize=12)
        plt.axis('off')
    plt.tight_layout()
    plt.show(block=False)

plot_eigenfaces(projection_matrix_2, 2, title_prefix="Top 2 Fisherfaces (p=2)")
plot_eigenfaces(projection_matrix_3, 3, title_prefix="Top 3 Fisherfaces (p=3)")
plt.show() # Display all the plots
```

c. LDA with p=9/15 preparation for Part 2

Similarly, the relevant code section is as follows:

```
# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array
```

```
projection_matrix_9, X_lda_9 = lda_from_scratch.mannual_lda(X_train, Y_train, 9) #
Perform LDA on the training images with 9 components
projection_matrix_15, X_lda_15 = lda_from_scratch.mannual_lda(X_train, Y_train, 15) #
Perform LDA on the training images with 15 components

X_test_lda_9 = X_test @ projection_matrix_9
X_test_lda_15 = X_test @ projection_matrix_15
```

Part 2. Pattern Recognition

1. KNN for classification

a. Apply KNN to PCA for p=80/200

In this section, I utilized the `KNeighborsClassifier` from `sklearn.neighbors` to build a KNN classifier with K=1. This means that for any given test sample, the KNN algorithm will search for its single nearest neighbor in the training set and assign the label of that neighbor as the prediction result. The training data was reduced in dimensionality using PCA to 80 and 200 dimensions, respectively, and two separate KNN models were trained on these reduced features. The test data was then fed into the trained models for prediction. The classification accuracy on the CMU PIE test set and the selfie test set is shown below:

```
[PCA-80] Accuracy on CMU PIE test images: 83.00%
[PCA-80] Accuracy on Selfie test images: 100.00%
[PCA-200] Accuracy on CMU PIE test images: 84.54%
[PCA-200] Accuracy on Selfie test images: 100.00%
```

For the PIE test set, the classification accuracy reaches 83% when using p = 80, and slightly higher at 84.54% when using p = 200. This indicates that the 200-dimensional representation retains more information and preserves more discriminative details, although the improvement is limited. It also shows that PCA effectively retains components with discriminative power even after dimensionality reduction.
For the selfie test set, both dimensionality reduction settings result in 100% classification accuracy, suggesting that the distribution of the selfie photos is noticeably different from the training samples. In the projected feature space, the nearest neighbors of the selfie samples are correctly identified as the same class, without being misclassified. However, this perfect accuracy is likely due to the small number of selfie test samples (only 3 images). If the number of selfie test images increases, the classification accuracy is expected to decrease.

The relevant code is shown below:

knn_pca_80_200.py

```python
import images_prep
import pca_from_scratch
import numpy as np
import random
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import os

os.environ["LOKY_MAX_CPU_COUNT"] = "20" # Set the maximum number of CPU cores to be used
by the joblib library
```

```
# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array

X_mean_80, projection_matrix_80, X_pca_80 = pca_from_scratch.mannual_pca(X_train, 80) #
Perform PCA on the training images with 80 components
X_mean_200, projection_matrix_200, X_pca_200 = pca_from_scratch.mannual_pca(X_train,
200) # Perform PCA on the training images with 200 components

X_test_pca_80 = (X_test - X_mean_80) @ projection_matrix_80 # Project the test images to
the new space (centralization using the mean of training images)
X_test_pca_200 = (X_test -  X_mean_200) @ projection_matrix_200

knn_80 = KNeighborsClassifier(n_neighbors=1) # Create a KNN classifier with 1 neighbor
knn_80.fit(X_pca_80, Y_train) # Fit the classifier with the training data
Y_pred_80 = knn_80.predict(X_test_pca_80) # Predict the test data

pie_mask = (Y_test != 69) # Boolean array to find photos that are not selfies
acc_pie_80 = accuracy_score(Y_test[pie_mask], Y_pred_80[pie_mask]) # Calculate the
accuracy of the model on non-selfie photos

selfie_mask = (Y_test == 69) # Boolean array to find the selfie photos
acc_selfie_80 = accuracy_score(Y_test[selfie_mask], Y_pred_80[selfie_mask]) # Calculate
the accuracy of the model on selfie photos

print(f"[PCA-80] Accuracy on CMU PIE test images: {acc_pie_80:.2%}")
print(f"[PCA-80] Accuracy on Selfie test images: {acc_selfie_80:.2%}")

knn_200 = KNeighborsClassifier(n_neighbors=1)
knn_200.fit(X_pca_200, Y_train)
```

```
Y_pred_200 = knn_200.predict(X_test_pca_200)

acc_pie_200 = accuracy_score(Y_test[pie_mask], Y_pred_200[pie_mask])
acc_selfie_200 = accuracy_score(Y_test[selfie_mask], Y_pred_200[selfie_mask])

print(f"[PCA-200] Accuracy on CMU PIE test images: {acc_pie_200:.2%}")
print(f"[PCA-200] Accuracy on Selfie test images: {acc_selfie_200:.2%}")
```

b. Apply KNN to LDA for p=9/15

Similarly, The classification accuracy on the CMU PIE test set and the selfie test set is shown below:

```
[LDA-9] Accuracy on CMU PIE test images: 75.00%
[LDA-9] Accuracy on Selfie test images: 33.33%
[LDA-15] Accuracy on CMU PIE test images: 78.08%
[LDA-15] Accuracy on Selfie test images: 33.33%
```

For the PIE test set, the classification accuracy reaches 75% when using p = 9, and slightly higher at 78.08% when using p = 15. This indicates that increasing the dimensionality from 9 to 15 enhances the separability between classes, allowing the model to better distinguish among the original categories. The classification accuracy for the selfie class is only 33.33%, meaning that only one out of the three selfie images was correctly classified, while the other two were misclassified as different classes. I believe this is because the selfie class was added separately and has very few training samples, resulting in poor discriminative ability for this class during LDA projection. It is also possible that the low accuracy is due to the limited number of test samples—only three images. If more selfie test images were included, the accuracy for this class would likely improve.
The relevant code is as follows:

knn_lda_9_15.py

```
import images_prep
import lda_from_scratch
import numpy as np
import random
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import os

os.environ["LOKY_MAX_CPU_COUNT"] = "20" # Set the maximum number of CPU cores to be used
by the joblib library

# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
```

```
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx) # Get the images and labels from the
dataset

X_train = np.array([img.flatten() for img in training_images]) # Flatten the traing
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images
Y_train = np.array(training_labels) # Convert the training labels to numpy array
Y_test = np.array(test_labels) # Convert the test labels to numpy array

projection_matrix_9, X_lda_9 = lda_from_scratch.mannual_lda(X_train, Y_train, 9) #
Perform LDA on the training images with 9 components
projection_matrix_15, X_lda_15 = lda_from_scratch.mannual_lda(X_train, Y_train, 15) #
Perform LDA on the training images with 15 components

X_test_lda_9 = X_test @ projection_matrix_9
X_test_lda_15 = X_test @ projection_matrix_15

knn_9 = KNeighborsClassifier(n_neighbors=1) # Create a KNN classifier with 1 neighbor
knn_9.fit(X_lda_9, Y_train) # Fit the classifier with the training data
Y_pred_9 = knn_9.predict(X_test_lda_9) # Predict the test data

pie_mask = (Y_test != 69) # Boolean array to find photos that are not selfies
acc_pie_9 = accuracy_score(Y_test[pie_mask], Y_pred_9[pie_mask]) # Calculate the
accuracy of the model on non-selfie photos

selfie_mask = (Y_test == 69) # Boolean array to find the selfie photos
acc_selfie_9 = accuracy_score(Y_test[selfie_mask], Y_pred_9[selfie_mask]) # Calculate
the accuracy of the model on selfie photos

print(f"[LDA-9] Accuracy on CMU PIE test images: {acc_pie_9:.2%}")
print(f"[LDA-9] Accuracy on Selfie test images: {acc_selfie_9:.2%}")

knn_15 = KNeighborsClassifier(n_neighbors=1)
knn_15.fit(X_lda_15, Y_train)
Y_pred_15 = knn_15.predict(X_test_lda_15)

acc_pie_15 = accuracy_score(Y_test[pie_mask], Y_pred_15[pie_mask])
acc_selfie_15 = accuracy_score(Y_test[selfie_mask], Y_pred_15[selfie_mask])
```

```
print(f"[LDA-15] Accuracy on CMU PIE test images: {acc_pie_15:.2%}")
print(f"[LDA-15] Accuracy on Selfie test images: {acc_selfie_15:.2%}")
```

2. GMM for clustering

a/b. Apply GMM to raw vectorized images and PCA for p=80/200

The construction of the GMM model mainly relies on `GaussianMixture` from `sklearn.mixture`. Figure 9 shows the result of applying GMM clustering directly to the original raw images without dimensionality reduction. The visualization is based on projecting the clustering results onto the first two principal components using PCA. From the figure, we can see that the boundaries among the three clusters are quite clear. Although the original data is high-dimensional, it retains all pixel details, which leads to well-separated clustering results.

Figure 10 shows the result of GMM clustering after reducing the data to 80 dimensions using PCA. After dimensionality reduction, the data is compressed and retains the main features, but a considerable amount of information is lost. As a result, the clustering outcome shows significant overlap and color mixing, indicating a decline in clustering quality.

Figure 11 shows the GMM clustering result after reducing the data to 200 dimensions using PCA. Compared to the 80-dimensional case, the cluster boundaries are much clearer, and the colored clusters are more distinctly separated. The clustering results are also more consistent with those of the raw image data. This indicates that higher-dimensional PCA representations preserve more class-discriminative features, which helps GMM better estimate accurate Gaussian distributions.

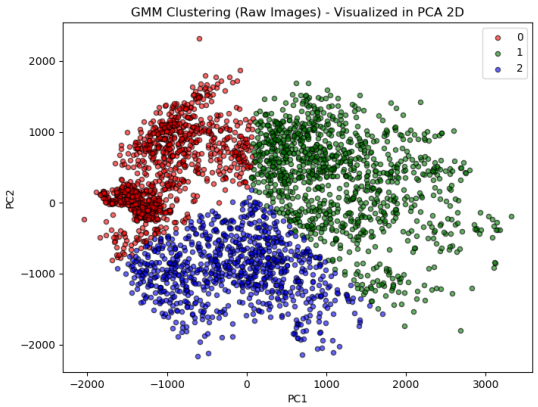


Fig. 9. GMM clustering on raw vectorized images

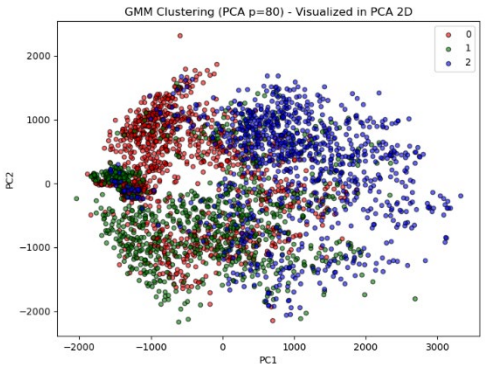


Fig. 10. GMM clustering (PCA p=80)

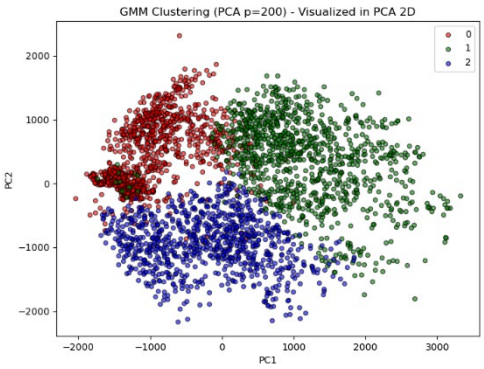


Fig. 11. GMM clustering on (PCA p=200)

The relevant code is as follows:

gmm_raw_80_200.py

```
import images_prep
import pca_from_scratch
import numpy as np
import random
from sklearn.mixture import GaussianMixture
import os
import matplotlib.pyplot as plt

os.environ["LOKY_MAX_CPU_COUNT"] = "20" # Set the maximum number of CPU cores to be used
by the joblib library

# Metric number: A0313771H
random.seed(71)  # For reproducibility
selected_idx = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE
dataset
selected_idx.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset
training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_idx)

X_train = np.array([img.flatten() for img in training_images]) # Flatten the training
images
X_test = np.array([img.flatten() for img in test_images]) # Flatten the test images

colors = ['red', 'green', 'blue']

# ======= GMM on raw vectorized images =======
```

```
gmm_raw = GaussianMixture(n_components=3, covariance_type='full', random_state=71) #
Create a GMM classifier with 3 components
gmm_raw.fit(X_train) # Fit the classifier with the training data
Y_pred_raw = gmm_raw.predict(X_train) # Predict the training data

_, _, X_raw_pca_2d = pca_from_scratch.mannual_pca(X_train, 2) # Perform PCA on the
training images with 2 principal components

plt.figure(figsize=(8, 6))
for cluster_id, color in enumerate(colors):
    idx = (Y_pred_raw == cluster_id)
    plt.scatter(X_raw_pca_2d[idx, 0], X_raw_pca_2d[idx, 1], c=color,
label=str(cluster_id),
                s=20, alpha=0.6, edgecolors='k')
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("GMM Clustering (Raw Images) - Visualized in PCA 2D")
plt.legend()

# ======= GMM on PCA p=80 images =======
_, _, X_pca_80 = pca_from_scratch.mannual_pca(X_train, 80)
gmm_80 = GaussianMixture(n_components=3, covariance_type='full', random_state=71)
gmm_80.fit(X_pca_80)
Y_pred_80 = gmm_80.predict(X_pca_80)

_, _, X_pca_80_2d = pca_from_scratch.mannual_pca(X_pca_80, 2)

plt.figure(figsize=(8, 6))
for cluster_id, color in enumerate(colors):
    idx = (Y_pred_80 == cluster_id)
    plt.scatter(X_pca_80_2d[idx, 0], X_pca_80_2d[idx, 1], c=color, label=str(cluster_id),
                s=20, alpha=0.6, edgecolors='k')
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("GMM Clustering (PCA p=80) - Visualized in PCA 2D")
plt.legend()

# ======= GMM on PCA p=200 images =======
_, _, X_pca_200 = pca_from_scratch.mannual_pca(X_train, 200)
gmm_200 = GaussianMixture(n_components=3, covariance_type='full', random_state=71)
gmm_200.fit(X_pca_200)
Y_pred_200 = gmm_200.predict(X_pca_200)

_, _, X_pca_200_2d = pca_from_scratch.mannual_pca(X_pca_200, 2)
```

```
plt.figure(figsize=(8, 6))
for cluster_id, color in enumerate(colors):
    idx = (Y_pred_200 == cluster_id)
    plt.scatter(X_pca_200_2d[idx, 0], X_pca_200_2d[idx, 1], c=color,
label=str(cluster_id),
                s=20, alpha=0.6, edgecolors='k')
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("GMM Clustering (PCA p=200) - Visualized in PCA 2D")
plt.legend()
plt.show()
```

3. SVM for classification

a/b/c/d. Apply linear SVM to raw vectorized images, PCA for p=80/200 and LDA for p=9/15

The construction of the SVM mainly relies on `OneVsRestClassifier` from `sklearn.multiclass`. The classification accuracy results of using different dimensionality reduction methods to process images and constructing SVM with different penalty coefficients C are shown below:

```
====== Raw Features ======
[Raw] C=0.01 | Train Acc: 100.00% | Test Acc: 94.55%
[Raw] C=0.1 | Train Acc: 100.00% | Test Acc: 94.55%
[Raw] C=1 | Train Acc: 100.00% | Test Acc: 94.55%

====== PCA Features ======
[PCA-80] C=0.01 | Train Acc: 100.00% | Test Acc: 87.95%
[PCA-200] C=0.01 | Train Acc: 100.00% | Test Acc: 93.55%
[PCA-80] C=0.1 | Train Acc: 100.00% | Test Acc: 87.95%
[PCA-200] C=0.1 | Train Acc: 100.00% | Test Acc: 93.55%
[PCA-80] C=1 | Train Acc: 100.00% | Test Acc: 87.95%
[PCA-200] C=1 | Train Acc: 100.00% | Test Acc: 93.55%

====== LDA Features ======
[LDA-9] C=0.01 | Train Acc: 93.37% | Test Acc: 63.16%
[LDA-15] C=0.01 | Train Acc: 98.48% | Test Acc: 70.61%
[LDA-9] C=0.1 | Train Acc: 93.27% | Test Acc: 63.78%
[LDA-15] C=0.1 | Train Acc: 98.04% | Test Acc: 69.61%
[LDA-9] C=1 | Train Acc: 94.99% | Test Acc: 63.24%
[LDA-15] C=1 | Train Acc: 97.90% | Test Acc: 64.54%
```

The effect of data dimension on the classification accuracy:

From the PCA results, we can see that the test accuracy at p=200 (93.55%) is higher than that at p=80 (87.95%), indicating that using more principal components allows the model to retain more

discriminative information, thus improving classification accuracy. Similarly, for LDA, the test accuracy at p=15 is also noticeably higher than at p=9, further suggesting that higher dimensionality can enhance the model's discriminative ability. Therefore, we can conclude that increasing the feature dimensionality within a reasonable range can improve the model's performance on the test set.

The effect of feature learning method on the classification accuracy:
When the data is not reduced in dimensionality, the test accuracy reaches 94.55%, which is the best performance, but the computational cost is high. After applying PCA for dimensionality reduction, there is a slight drop in accuracy (with a maximum of 93.55%), but the model still maintains high precision, indicating that PCA can effectively compress data while preserving discriminative power. In contrast, when using LDA for dimensionality reduction, although the training accuracy is high, the test accuracy is significantly lower than PCA (with a maximum of 70.61%), suggesting that LDA focuses more on between-class separability but tends to overfit when the number of classes is large.

The effect of parameter C on the classification accuracy:
When no dimensionality reduction or PCA is applied, the results under different values of C (0.01, 0.1, 1) show almost no variation, indicating that the model is relatively insensitive to C under raw or PCA features. However, when LDA is used for dimensionality reduction, the test accuracy changes significantly with C. As C increases, the test accuracy decreases, suggesting that the model is more sensitive to the strength of regularization under LDA features.
From a theoretical perspective, when C is small, the SVM tends to tolerate more classification errors on the training set and aims to find a larger margin, which generally leads to better generalization. As C becomes larger, the penalty for misclassification increases, making the model more likely to overfit. This aligns well with the experimental results.

The relevant code is as follows:

svm_raw_80_200_9_15.py

```
import images_prep
import pca_from_scratch
import lda_from_scratch
import numpy as np
import random
from sklearn.multiclass import OneVsRestClassifier
from sklearn import svm
from sklearn.metrics import accuracy_score

# ======= Data preparation =======
random.seed(71)
selected_indices = random.sample(range(1, 69), 25)
selected_indices.append(69)

PIE_path = "./PIE"
```

```
training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_indices)

X_train_raw = np.array([img.flatten() for img in training_images])
X_test_raw = np.array([img.flatten() for img in test_images])
Y_train = np.array(training_labels)
Y_test = np.array(test_labels)

mean_80, proj_80, X_train_pca_80 = pca_from_scratch.mannual_pca(X_train_raw, 80)
mean_200, proj_200, X_train_pca_200 = pca_from_scratch.mannual_pca(X_train_raw, 200)
X_test_pca_80 = (X_test_raw - mean_80) @ proj_80
X_test_pca_200 = (X_test_raw - mean_200) @ proj_200

proj_lda_9, X_train_lda_9 = lda_from_scratch.mannual_lda(X_train_raw, Y_train, 9)
proj_lda_15, X_train_lda_15 = lda_from_scratch.mannual_lda(X_train_raw, Y_train, 15)
X_test_lda_9 = X_test_raw @ proj_lda_9
X_test_lda_15 = X_test_raw @ proj_lda_15

# ======= Define a general SVM training evaluation function =======
def train_and_evaluate_svm(X_train, Y_train, X_test, Y_test, C_value):
    ovr_svm = OneVsRestClassifier(svm.SVC(kernel='linear', C=C_value, random_state=71))
    ovr_svm.fit(X_train, Y_train)
    y_train_pred = ovr_svm.predict(X_train)
    y_test_pred = ovr_svm.predict(X_test)
    train_accuracy = accuracy_score(Y_train, y_train_pred)
    test_accuracy = accuracy_score(Y_test, y_test_pred)
    return train_accuracy, test_accuracy

# ======= Evaluating SVM classification performance under different dimensionality
reduction methods =======
C_values = [1e-2, 1e-1, 1]

print("====== Raw Features ======")
for C in C_values:
    train_acc, test_acc = train_and_evaluate_svm(X_train_raw, Y_train, X_test_raw,
Y_test, C)
    print(f"[Raw] C={C} | Train Acc: {train_acc:.2%} | Test Acc: {test_acc:.2%}")

print("\n====== PCA Features ======")
for C in C_values:
    train_acc_80, test_acc_80 = train_and_evaluate_svm(X_train_pca_80, Y_train,
X_test_pca_80, Y_test, C)
    print(f"[PCA-80] C={C} | Train Acc: {train_acc_80:.2%} | Test Acc:
{test_acc_80:.2%}")
```

```
    train_acc_200, test_acc_200 = train_and_evaluate_svm(X_train_pca_200, Y_train,
X_test_pca_200, Y_test, C)
    print(f"[PCA-200] C={C} | Train Acc: {train_acc_200:.2%} | Test Acc:
{test_acc_200:.2%}")

print("\n====== LDA Features ======")
for C in C_values:
    train_acc_9, test_acc_9 = train_and_evaluate_svm(X_train_lda_9, Y_train,
X_test_lda_9, Y_test, C)
    print(f"[LDA-9] C={C} | Train Acc: {train_acc_9:.2%} | Test Acc: {test_acc_9:.2%}")

    train_acc_15, test_acc_15 = train_and_evaluate_svm(X_train_lda_15, Y_train,
X_test_lda_15, Y_test, C)
    print(f"[LDA-15] C={C} | Train Acc: {train_acc_15:.2%} | Test Acc:
{test_acc_15:.2%}")
```

4. CNN for classification

The construction, training and evaluation of the CNN model mainly relies on `PyTorch`.
The CNN model was constructed strictly following the task requirements. The optimizer used is `Adam` with a `learning rate of 0.001`. The loss function is `cross-entropy loss`, and the `batch size is set to 32`. The model was trained for `20 epochs`. The accuracy and loss on both the training and test sets are shown in Figures 12 and 13. Final evaluation metrics such as accuracy, precision, recall, and F1 score are also presented below. The trained model weights are saved in the current working directory with the filename cnn_model.pth.

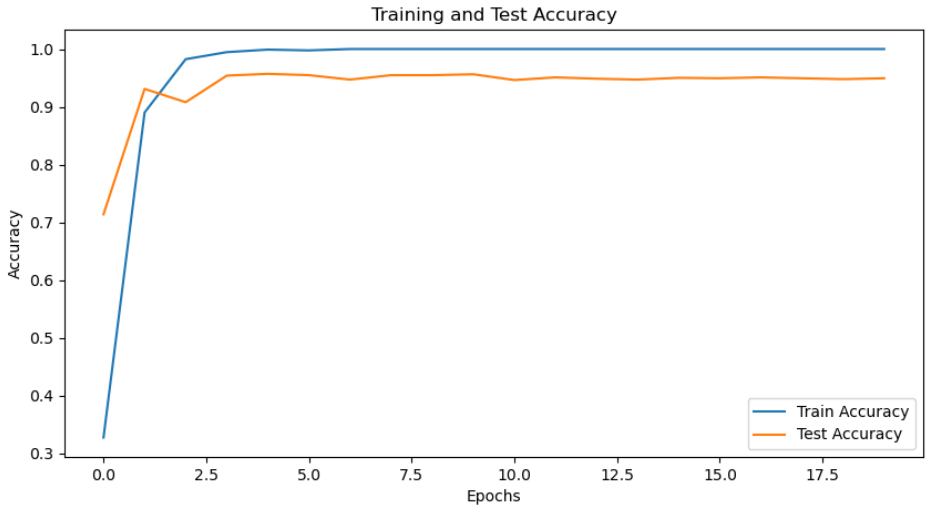


Fig. 12. Training and test set accuracy using CNN

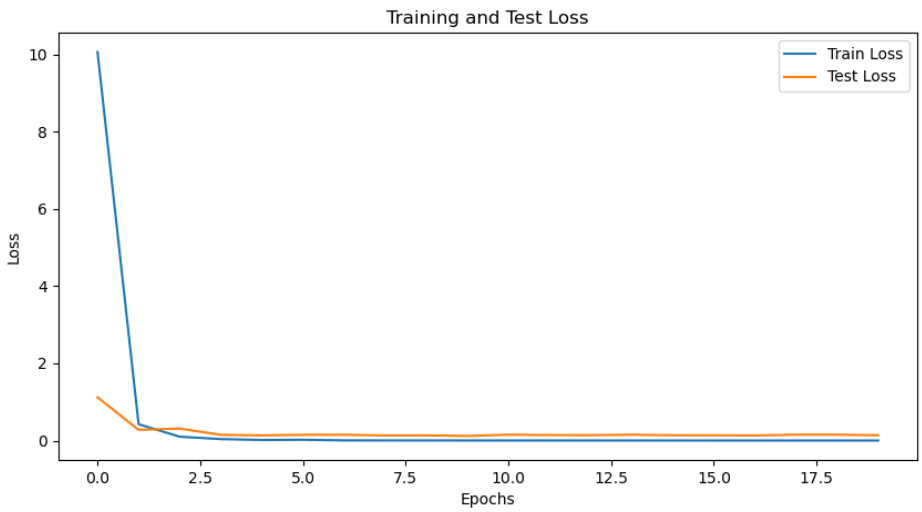


Fig. 13. Training and test set loss using CNN

```
=== Final Evaluation on Test Set ===
Accuracy: 96.93%
Precision: 97.11%
Recall: 94.63%
F1 Score: 95.13%
Model saved to ./cnn_model.pth
```

From the figures, it can be observed that the training accuracy increases rapidly and approaches saturation around the 3rd to 4th epoch. The test accuracy also rises accordingly, reaching over 90% by the 3rd epoch and then stabilizing at approximately 95%–97%. This indicates that the model has strong fitting capability, quickly learning effective features in the early stages and performing well on the test set without showing obvious signs of overfitting. Additionally, the performance metrics on the test set are all better than those achieved by previous PCA/LDA + SVM/KNN combination models, demonstrating that the CNN has learned more discriminative features through end-to-end training.
The relevant code is as follows:

cnn.py

```
import images_prep
import random
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
from sklearn.preprocessing import LabelEncoder

# ======= Data Preparation =======
# Metric number: A0313771H
random.seed(71) # Random seed for reproducibility
selected_indices = random.sample(range(1, 69), 25) # Randomly select 25 people from PIE dataset
selected_indices.append(69) # Add selfie photos to the list

PIE_path = "./PIE" # Path to PIE dataset

training_images, training_labels, test_images, test_labels =
images_prep.get_images(PIE_path, selected_indices) # Get the images and labels from the dataset

X_train = np.array(training_images).reshape(-1, 1, 32, 32) # Reshape the training images to (N, C, H, W)
X_test = np.array(test_images).reshape(-1, 1, 32, 32) # Reshape the test images to (N, C, H, W)
X_train = torch.tensor(X_train, dtype=torch.float32) # Convert the training images to PyTorch tensor
X_test = torch.tensor(X_test, dtype=torch.float32) # Convert the test images to PyTorch tensor

label_encoder = LabelEncoder() # Create a label encoder (original labels are between 1-69, we need to convert them to 0-25)
Y_train_encoded = label_encoder.fit_transform(training_labels) # Encode the training labels
Y_test_encoded = label_encoder.transform(test_labels) # Encode the test labels

Y_train = torch.tensor(Y_train_encoded, dtype=torch.long) # Convert the training labels to PyTorch tensor
Y_test = torch.tensor(Y_test_encoded, dtype=torch.long) # Convert the test labels to PyTorch tensor

train_dataset = TensorDataset(X_train, Y_train) # Create a training dataset
test_dataset = TensorDataset(X_test, Y_test) # Create a test dataset

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True) # Create a training data loader
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False) # Create a test data loader

# ======= Define a CNN According to the Requirements =======
```

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, kernel_size=5, stride=1, padding=0) # Convolutional
layer with 20 filters. Kernel size 5x5, stride 1, padding 0
        self.conv2 = nn.Conv2d(20, 50, kernel_size=5, stride=1, padding=0) #
Convolutional layer with 50 filters. Kernel size 5x5, stride 1, padding 0
        self.fc = nn.Linear(50 * 5 * 5, 500) # Fully connected layer with 500 units, 5 *
5 because of two convolutional layers and two max-pooling layers
        self.op = nn.Linear(500, 26) # Output layer with 26 units (26 classes)

    def forward(self, x):
        x = self.conv1(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2) # Max pooling layer with kernel size
2x2, stride 2
        x = self.conv2(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2) # Max pooling layer with kernel size
2x2, stride 2
        x = x.view(-1, 50 * 5 * 5)
        x = F.relu(self.fc(x))
        x = self.op(x) # Output layer without softmax activation because CrossEntropyLoss
includes it
        return x

# ======= Define the Training Function =======
def train(model, train_loader, optimizer, criterion, device, verbose=False):
    model.train() # Set the model to training mode
    running_loss = 0.0 # Initialize the running loss
    total_correct = 0 # Initialize the total correct predictions
    total_samples = 0 # Initialize the total samples

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device) # Move the inputs and
labels to the device
        optimizer.zero_grad() # Zero the parameter gradients
        outputs = model(inputs) # Forward pass
        loss = criterion(outputs, labels) # Calculate the loss
        loss.backward() # Backward pass
        optimizer.step() # Optimize
        running_loss += loss.item() # Accumulate the loss
        _, preds = torch.max(outputs, dim=1) # Get the predicted labels
        total_correct += torch.sum(preds == labels).item() # Accumulate the correct
predictions
        total_samples += labels.size(0) # Accumulate the total samples
```

```
    avg_loss = running_loss / len(train_loader) # Calculate the average loss
    accuracy = total_correct / total_samples # Calculate the accuracy

    if verbose:
        print(f"Test Loss: {avg_loss:.4f}, Test Accuracy: {accuracy:.2%}")

    return avg_loss, accuracy

# ======= Define the Evaluation Function =======
def evaluate(model, test_loader, criterion, device, verbose=False):
    model.eval()
    running_loss = 0.0
    total_correct = 0
    total_samples = 0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item()
            _, preds = torch.max(outputs, dim=1)
            total_correct += torch.sum(preds == labels).item()
            total_samples += labels.size(0)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    avg_loss = running_loss / len(test_loader)
    accuracy = total_correct / total_samples

    if verbose:
        print(f"Test Loss: {avg_loss:.4f}, Test Accuracy: {accuracy:.2%}")

    return avg_loss, accuracy, all_preds, all_labels

# ======= Train the Model =======
num_epochs = 20
train_losses = []
test_losses = []
train_accuracies = []
test_accuracies = []
```

```
# Define the training parameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # Use GPU if
available
model = CNN().to(device) # Move the model to the device
optimizer = optim.Adam(model.parameters(), lr=0.001) # Adam optimizer
criterion = nn.CrossEntropyLoss() # Cross-entropy loss

for epoch in range(num_epochs):
    train_loss, train_accuracy = train(model, train_loader, optimizer, criterion, device)
    test_loss, test_accuracy, eva_preds, eva_labels = evaluate(model, test_loader,
criterion, device)
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Accuracy:
{train_accuracy:.2%}, Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2%}")

# Calculate the evaluation metrics: accuracy score, precision score, recall score, f1
score
accuracy = accuracy_score(eva_labels, eva_preds)
precision = precision_score(eva_labels, eva_preds, average='macro')
recall = recall_score(eva_labels, eva_preds, average='macro')
f1 = f1_score(eva_labels, eva_preds, average='macro')

print(f"\n=== Final Evaluation on Test Set ===")
print(f"Accuracy: {accuracy:.2%}")
print(f"Precision: {precision:.2%}")
print(f"Recall: {recall:.2%}")
print(f"F1 Score: {f1:.2%}")

# Save the model
model_path = "./cnn_model.pth"
torch.save(model.state_dict(), model_path)
print(f"Model saved to {model_path}")

# Plot the training and test losses
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
plt.title('Training and Test Loss')
plt.show(block=False)

# Plot the training and test accuracies
plt.figure(figsize=(10, 5))
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Test Accuracy')
plt.show()
```