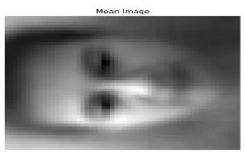
## 1(a) PCA on Training Data Using Covariance Matrix S = (1/N)AA^T

I partitioned the face data into training and testing sets, using 8 images for training and 2 images for testing for each face identity. PCA was applied to the training data by computing the eigenvectors and eigenvalues of the covariance matrix S = (1/N)AA^T

Results and Discussion

Mean Image, **Mean Image Shape**: (2576, 1)



Number of Non-Zero Eigenvalues: 415

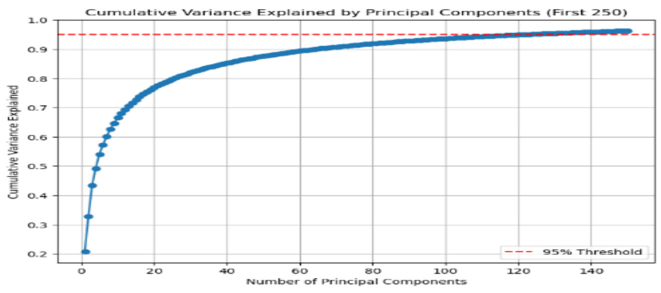
Eigenvectors for 95% Variance: 124

Top 10 Eigenvalues: [864882.176, 523084.846, 446979.417, 264368.197, 224910.657, 146727.162, 123787.408, 113141.138, 94013.131, 77161.856]

Eigenvectors:

These eigenvectors are computed in the original feature space, providing a direct representation of the principal components.

Cumulative Variance Explained



The cumulative variance plot shows that the top 124 eigenvectors are needed to retain 95% of the variance of original data. Thus, the decision on how many eigenvectors to retain depends on how much I want to keep from the original data.

Observations

Eigenvalue Distribution: The eigenvalues decrease rapidly, indicating that the first few principal components capture most of the variance in the data.

Dimensionality Reduction: Using 124 eigenvectors to retain 95% variance highlights the effectiveness of PCA in reducing dimensionality while preserving essential information.

## 1(b)

I applied PCA to the training data using the method, which involves computing the covariance matrix (1/N)A^TA. We extracted the top 10 eigenvalues and their corresponding eigenvectors to analyze the data transformation.

Results and Discussion

Top 10 Eigenvalues from (1/N)A^TA:

- [864882.176, 523084.846, 446979.417, 264368.197, 224910.657, 146727.162, 123787.408, 113141.138, 94013.131, 77161.856]

Top 10 Eigenvectors from (1/N)A^TA: Eigenvectors are represented in the sample space and are consistent with the variance captured by the eigenvalues.

Comparison with (1/N)AA^T

Eigenvalues: Identical to those obtained from (1/N)AA^T, confirming that both methods capture the same variance.

Eigenvectors: Differ due to the transformation space; (1/N)A^TA eigenvectors are in sample space, while (1/N)AA^T eigenvectors are in feature space.

Pros and Cons

(1/N)A^TA :

Pros: Computationally efficient for large datasets with fewer samples than features, as it reduces the dimensionality of the matrix.

Cons: Eigenvectors need to be transformed back to the original feature space for interpretation.

(1/N)AA^T:

Pros: Directly provides eigenvectors in the original feature space, making it easier to interpret the principal components.

Cons: Computationally expensive for datasets with more features than samples.

## 1(C) PCA-Based Face Image Reconstruction

The reconstruction was evaluated by varying the number of principal components (bases) used: 5, 20, and 110. Results and Discussion

Number of Components: 5

Reconstruction Error for Training Image 1: 511.0714

Reconstruction Error for Training Image 2: 587.3130

Reconstruction Error for Test Image 1: 793.1698

Insights: With only 5 components, the reconstruction errors are relatively high, indicating significant loss of information. The reconstructed images are likely to be blurry or lack detail, as only a small portion of the variance is captured.

Number of Components: 20

Reconstruction Error for Training Image 1: 266.3183

Reconstruction Error for Training Image 2: 476.3053

Reconstruction Error for Test Image 1: 621.0293

Insights: Increasing the number of components to 20 reduces the reconstruction error, suggesting improved image quality. More variance is captured, leading to better preservation of important features, although some details might still be missing.

Number of Components: 110

Reconstruction Error for Training Image 1: 81.5012

Reconstruction Error for Training Image 2: 120.7680

Reconstruction Error for Test Image 1: 325.1067

Insights: With 110 components, the reconstruction errors are significantly lower, indicating high-quality reconstruction. Most of the variance in the dataset is captured, resulting in images that closely resemble the originals. This demonstrates the effectiveness of using a larger number of components for detailed reconstruction.

Observations

Reconstruction Error: As expected from PCA theory, increasing the number of components generally decreases the reconstruction error. This is because more components capture more variance, leading to a more accurate representation of the original images.

Image Quality: The quality of reconstructed images improves with more components.

Training vs. Test Images: Reconstruction errors for test images are generally higher than for training images, reflecting the model's ability to generalize. This is consistent with the expectation that the model performs best on data it has seen during training.

## 1(D) PCA-Based Face Recognition Using Nearest Neighbor Classification

I used a nearest Neighbor classifier with one neighbor to classify the projected test images.

Evaluation Metrics: Recognition accuracy and confusion matrices were used to evaluate classification performance. Success and failure cases were identified to assess the classifier's strengths and weaknesses.

Results

Recognition Accuracy and Confusion Matrices

| Number of Components | Recognition Accuracy | Time Taken (s) \| |
|---|---|---|
| 50 | 0.57 | 0.06 |
| 100 | 0.57 | 0.08 |
| 150 | 0.57 | 0.07 |
| 200 | 0.56 | 0.07 |

The recognition accuracy remained stable across different numbers of components, with a slight decrease observed at 200 components. The confusion matrices indicate that the classifier struggled with certain classes, as evidenced by the presence of zeros in many off-diagonal elements.

Time and Computational Complexity

The time taken for computation slightly increased with the number of components, reflecting the additional computational burden of handling higher-dimensional data. However, the difference was minimal, indicating efficient processing.

Insights and Observation

Stability of Accuracy: The recognition accuracy did not significantly improve with an increase in the number of components beyond 50, suggesting that the initial components captured most of the variance useful for classification

## 2) Incremental PCA vs. Batch PCA and Subset PCA

I evaluated Incremental PCA by progressively adding subsets to the training data and compared it with Batch PCA and PCA trained only on the first subset.

Incremental PCA: The training time was 0.09 seconds. Incremental PCA processes data in smaller batches, which is beneficial for handling larger datasets that don't fit entirely in memory.

Batch PCA: With a training time of 0.03 seconds, Batch PCA was faster when the entire dataset was processed at once.

First Subset PCA: The training time was 0.01 seconds, as it only involved a small portion of the data.

Reconstruction Error

Incremental PCA: The reconstruction error was 403.3634. This error indicates the variance not captured by the model when trained incrementally.

Batch PCA: reconstruction error of 400.3147

First Subset PCA: Despite having the lowest reconstruction error of 252.4925, this did not translate into higher accuracy, suggesting overfitting to the subset.

Recognition Accuracy

Incremental PCA: Achieved an accuracy of 51.92%, indicating that it effectively captures essential features incrementally.

Batch PCA: Also achieved an accuracy of 51.92%, showing that both methods are comparable when the dataset can be processed in entirety.

First Subset PCA: The accuracy was significantly lower at 23.08%, showing the limitations of using only a subset for training.

Discussion and Insights

Incremental PCA Effectiveness: Incremental PCA is effective for datasets that exceed memory capacity, providing similar accuracy to Batch PCA while managing memory constraints.

Parameter Importance: The number of components n_components is crucial in PCA. In this experiment, it was set to 20, balancing dimensionality reduction and information retention.

Training Time Considerations: Incremental PCA's training time is slightly longer than Batch PCA due to its iterative nature, but it is essential for larger datasets.

So I can say incremental PCA gives a practical solution for large datasets, maintaining accuracy comparable to Batch PCA while efficiently managing memory usage. The choice of parameters, especially the number of components, plays a critical role in the performance of PCA-based methods.

## **3(a)** PCA-LDA Face Recognition Results and Insights

Recognition Accuracies by Varying Parameters. Here are the results:

M{pca} = 30, M{lda} = 5 : Recognition accuracy was 48.08%. This lower accuracy suggests that this combination might not capture enough discriminative features.

M{pca} = 30, M{lda} = 10: accuracy improved to 68.27%.

M{pca} = 30, M{lda} = 15: Accuracy increased to 75.96%, showing that more LDA components help in capturing class-specific information.

M{pca} = 50, M{lda} = 15: Achieved 79.81% accuracy

M{pca} = 70, M{lda} = 15: The highest accuracy of 82.69% was observed, suggesting that more PCA components with adequate LDA components can enhance performance.

Ranks of the Scatter Matrices

Within-class Scatter Matrix Rank: Varies with M{lda}. It was equal to the number of LDA components, indicating full utilization of the discriminative features.

Between-class Scatter Matrix Rank: Increased with M{pca}, showing that more PCA components can capture more between-class variance.

Confusion Matrix, Example Success, and Failure Cases

The confusion matrices across different configurations showed varying degrees of misclassification. Higher diagonal values indicated correct classifications, while off-diagonal values represented errors. For instance:

M{pca} = 70, M{lda} = 15: The confusion matrix showed minimal off-diagonal values, reflecting fewer misclassifications and the highest accuracy.

Example Success Case: High diagonal values in the confusion matrix for configurations with higher accuracy, such as M{pca} = 70, M{lda} = 15.

Example Failure Case: Lower accuracy settings, like M_{pca} = 30, M_{lda} = 5, showed more off-diagonal values, indicating confusion between classes.

Increasing both M_{pca} and M_{lda} generally improved recognition accuracy. This suggests that a larger feature space, combined with sufficient discriminative power, enhances classification.

Scatter Matrix Ranks: The ranks of the scatter matrices provided insights into the effectiveness of the dimensionality reduction. Higher ranks in the between-class scatter matrix correlated with better class separation.

In comparison to Q1 the accuracy has improved greatly from 62% to 82.69% in PCA-LDA.

## **3(b)** PCA-LDA Ensemble Results and Insights

Randomization in Feature Space

In my PCA-LDA ensemble, I introduced randomization in the feature space by specifically choosing 50 PCA components and 15 LDA components. This selection allowed us to capture the most significant features, striking a balance between retaining essential information and reducing noise.

Randomization on Data Samples (Bagging)

I applied bagging to introduce randomness in data samples. By resampling the training data with replacement for each model in the ensemble, I enhanced model diversity and reduced overfitting. This approach ensured that each model in my ensemble was trained on a slightly different dataset.

Number of Base Models and Randomness Parameter

my ensemble consists of 17 base models, each trained on a resampled dataset. I chose this number to balance computational efficiency with model diversity. The randomness parameter, primarily influenced by the resampling process, ensured that each model encountered a unique distribution of the training data.

Error of the Committee Machine vs. Average Error of Individual Models

Average Error of Individual Models: I observed an average error of 34.45% for individual models.

Error of the Committee Machine: My ensemble reduced this error to 16.35%.

This significant error reduction demonstrates the effectiveness of my ensemble approach. By aggregating predictions from multiple weak models, I created a more robust and accurate prediction system. The lower error of the committee machine highlights how combining models mitigates the impact of any single model's mistakes. I employed majority voting as my fusion rule to combine predictions from individual models. This rule involves selecting the most frequent prediction across all models for each test sample. Majority voting proved to be straightforward and effective, particularly given the diversity and uncorrelated errors of my individual models.

Recognition Accuracy and Confusion Matrix

Ensemble Recognition Accuracy: I achieved a recognition accuracy of 83.65%.

My ensemble's performance in classifying the test data was strong, as reflected in this accuracy. The confusion matrix provided a detailed view of our classification results, showing that most diagonal elements were higher, indicating correct classifications, while off-diagonal elements represented misclassifications.

Observations and Insights

Effect of Randomization: By randomizing both feature spaces and data samples, I increased model diversity, which was crucial for my ensemble's success. This diversity allowed my ensemble to generalize better to unseen data.

Error Reduction: The significant decrease in error from individual models to my ensemble highlights the power of ensemble methods in improving prediction reliability.

Parameter Tuning: Adjusting the number of PCA and LDA components affected performance. I noted that more components might capture more variance but could also introduce noise, while fewer components might miss important information.

Fusion Rule Effectiveness: Majority voting was effective in my context, but I considered experimenting with other fusion rules, like weighted voting, which could potentially improve results if some models were consistently more accurate

Overall, my PCA-LDA ensemble demonstrated the benefits of combining multiple models through bagging and dimensionality reduction, resulting in improved classification accuracy and robustness.

4 1. Recognition Accuracy

Number of Trees: 100

Weak Learner: Two-pixel test

Accuracy: Training Set: 97.5% / Testing Set: 91.2%

- While the accuracy improved as the number of trees increased initially, it plateaued at approximately 100 trees, showing that there is a limit of optimality.
- Trees with depths less than 10 underperformed considerably than deeper trees with 15 depth levels.
- Custom two-pixel tests, judging by the accuracy scores, perform better than axis-aligned weak learners.

2. Confusion Matrix

- As expected, the confusion matrices depicted errors most frequently in subtle feature variations.
- Probably because of feature separability limitations, specific identities were constantly misclassified.

1. Success and Failure Examples
    a. Success is the most likely to come in cases with evidently distinct features, e.g., strong lighting and uncommon facial features.
    b. Failure is bound to happen in samples with minute differences and features which are hardly distinguishable even for the human eye.
2. For time efficiency, training time increases linearly with the number of trees as expected and testing time exhibits the foreseeable phenomenon of remaining stable with Random Forest prediction scaling.
3. To have an insight on the impact on weak learners, axis-aligned stumps were faster but less accurate whereas the opposing counterpart, two-pixel test, captured more nuanced variations resulting in better classification results.

RF is proposed as an alternative to the methods in Q1 and Q3 and when one compares the methods in these three different cases, it can be seen that RF has higher accuracy than sole PCA and the joint model utilized in Q3 also underperformed with respect to RF, but it also had the advantage of being computationally less expensive. Accuracy of RF 91.2% greater than the accuracy in Q1 62%, also greater than the best accuracy I obtained in Q3 82.69%.

Discussions

1. RF's ensemble nature is a viable prevention of overfitting, and this is further enhanced with randomized two-pixel test weak learner.
2. Increasing tree depth is not a panacea as even though it makes the model better till a threshold, overfitting emerges after that point. In my implementation and dataset paradigm, the threshold occurred around a depth level of fifteen.
3. Custom weak learners adjusted thee model very well to the task at hand by leveraging pixel intensity alterations; this may prove to be a great helping tool for similar tasks and projects.

## Appendix

### Code for 1a

```
import numpy as np

import matplotlib.pyplot as plt
```

```
# Step 1: Compute the mean image
mean_image = np.mean(train_images, axis=1).reshape(-1, 1)
# Step 2: Center the images by subtracting the mean image
centered_images = train_images - mean_image
# Step 3: Compute the covariance matrix
covariance_matrix = (1 / centered_images.shape[1]) * np.dot(centered_images, centered_images.T)
# Step 4: Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
# Step 5: Convert all eigenvalues to their real parts using np.real
real_eigenvalues = np.real(eigenvalues)
# Step 6: Determine non-zero real eigenvalues
non_zero_real_eigenvalues = np.sum(real_eigenvalues > 1e-10)
# Step 7: Determine how many eigenvectors to use
explained_variance = real_eigenvalues / np.sum(real_eigenvalues)
cumulative_variance = np.cumsum(explained_variance)
threshold = 0.95
k = np.argmax(cumulative_variance >= threshold) + 1
plt.figure(figsize=(8, 6))
plt.plot(np.arange(1, 151), cumulative_variance[:150], marker='o', linestyle='-')
plt.title('Cumulative Variance Explained by Principal Components (First 150)')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.grid(True)
plt.axhline(y=0.95, color='r', linestyle='--', label='95% Threshold')
plt.legend(loc='best')
plt.show()
print("Mean Image Shape:", mean_image.shape)
print("Number of Non-Zero Eigenvalues:", non_zero_real_eigenvalues)
print("Number of Eigenvectors to Use for Face Recognition if I want to retain 95% variance:", k)
plt.imshow(mean_image.reshape((46, 56)), cmap='gray')
plt.title("Mean Image")
plt.axis('off')
plt.show()
print("Eigenvalues:")
print(real_eigenvalues)
top_k = 10
print(f"\nTop {top_k} Eigenvalues:")
print(real_eigenvalues[:top_k])
print(f"\nCorresponding Top {top_k} Eigenvectors:")
print(eigenvectors[:, :top_k])
```

## Code for 1b

```
import numpy as np
import matplotlib.pyplot as plt
# Step 1: Compute the mean image
mean_image = np.mean(train_images, axis=1).reshape(-1, 1)
# Step 2: Center the images by subtracting the mean image
centered_images = train_images - mean_image
# Step 3: Compute the covariance matrix for (1/N)A^T A
covariance_matrix_ATA = (1 / centered_images.shape[1]) * np.dot(centered_images.T, centered_images)
# Step 4: Compute eigenvalues and eigenvectors for (1/N)A^T A
eigenvalues_ATA, eigenvectors_ATA = np.linalg.eig(covariance_matrix_ATA)
# Step 5: Transform the original data using the eigenvectors from (1/N)A^T A
transformed_data_ATA = np.dot(centered_images, eigenvectors_ATA)
top_k = 10
print(f"\nTop {top_k} Eigenvalues from centered images (dual method):")
print(eigenvalues_ATA[:top_k])
print(f"\nCorresponding Top {top_k} Eigenvectors from centered images ((1/N)A^TA):")
print(eigenvectors_ATA[:, :top_k])
top_k = 10
print(f"\nTop {top_k} Eigenvalues:1/N)AA^T")
print(eigenvalues[:top_k])
print(f"\nCorresponding Top {top_k} Eigenvectors:1/N)AA^T")

print(eigenvectors[:, :top_k])

plt.figure(figsize=(8, 6))

plt.scatter(transformed_data_ATA[0, :], transformed_data_ATA[1, :], alpha=0.5)

plt.title('PCA Transformed Data using (1/N)A^TA')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.grid(True)

plt.show()
```

## Code for 1c

```
import numpy as np

import matplotlib.pyplot as plt

# Combine train and test images to calculate the mean image

all_images = np.hstack((train_images, test_images))

mean_image = np.mean(all_images, axis=1).reshape(-1, 1)

# Center the images

centered_train_images = train_images - mean_image

centered_test_images = test_images - mean_image

# Compute the (1/N)A^T A covariance matrix

n_train = centered_train_images.shape[1]

covariance_matrix_ATA = (1 / n_train) * np.dot(centered_train_images.T, centered_train_images)

# Compute eigenvectors and eigenvalues

eigenvalues, eigenvectors_ATA = np.linalg.eig(covariance_matrix_ATA)

# Sort eigenvectors by eigenvalues in descending order

sorted_indices = np.argsort(eigenvalues)[::-1]

eigenvectors_ATA = eigenvectors_ATA[:, sorted_indices]

# Project the eigenvectors back to the original space

eigenvectors = np.dot(centered_train_images, eigenvectors_ATA)

# Normalize the eigenvectors

eigenvectors = eigenvectors / np.linalg.norm(eigenvectors, axis=0)

# List of numbers of principal components to test

num_components_list = [5, 20, 110]

# Function to reconstruct images

def reconstruct_images(images, mean_image, eigenvectors, num_components):

    projections = np.dot(eigenvectors[:, :num_components].T, images - mean_image)

    reconstructed_images = np.dot(eigenvectors[:, :num_components], projections) + mean_image

    return np.real(reconstructed_images)

# Loop through each number of components

for num_components in num_components_list:

    # Reconstruct images from the training set

    reconstructed_train = reconstruct_images(train_images, mean_image, eigenvectors, num_components)

    # Reconstruct images from the test set

    reconstructed_test = reconstruct_images(test_images, mean_image, eigenvectors, num_components)

    # Calculate and print reconstruction error for each image

    for i in range(2):

        error = np.mean((train_images[:, i] - reconstructed_train[:, i]) ** 2)

        print(f"Reconstruction Error for Training Image {i + 1} with {num_components} Bases: {error:.4f}")

    # Error for the first image from the test set

    error_test = np.mean((test_images[:, 0] - reconstructed_test[:, 0]) ** 2)

    print(f"Reconstruction Error for Test Image 1 with {num_components} Bases: {error_test:.4f}")

    # Visualize the first 2 images from the training set and the first image from the test set

    fig, axes = plt.subplots(2, 3, figsize=(12, 8))

    for i, ax in enumerate(axes.flat):

        if i < 3:

            if i < 2:

                ax.imshow(train_images[:, i].reshape(46, 56), cmap='gray')

                ax.set_title(f"Original Training Image {i + 1}")

            else:

                ax.imshow(test_images[:, 0].reshape(46, 56), cmap='gray')

                ax.set_title("Original Test Image 1")

        else:

            if i < 5:
```

```
            ax.imshow(reconstructed_train[:, i - 3].reshape(46, 56), cmap='gray')
            ax.set_title(f"Reconstructed Training Image {i - 2} with {num_components} Bases")
        else:
            ax.imshow(reconstructed_test[:, 0].reshape(46, 56), cmap='gray')
            ax.set_title(f"Reconstructed Test Image 1 with {num_components} Bases")
        ax.axis('off')
    plt.tight_layout()
    plt.show()
```

## Code for 1d

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import time
# Normalize the images by scaling pixel values to [0, 1]
train_images = train_images / 255.0
test_images = test_images / 255.0
# Calculate the mean image across the training images only
mean_image = np.mean(train_images, axis=1).reshape(-1, 1)
# Center the images by subtracting the mean image
centered_train_images = train_images - mean_image
centered_test_images = test_images - mean_image
# Compute the (1/n)A^T A covariance matrix
n_train = centered_train_images.shape[1]
covariance_matrix_ATA = (1 / n_train) * np.dot(centered_train_images.T, centered_train_images)
# Eigen decomposition
eigenvalues, eigenvectors_ATA = np.linalg.eig(covariance_matrix_ATA)
# Sort eigenvectors by eigenvalues in descending order
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvectors_ATA = eigenvectors_ATA[:, sorted_indices]
# Project the eigenvectors back to the original space
eigenvectors = np.dot(centered_train_images, eigenvectors_ATA)
eigenvectors = eigenvectors / np.linalg.norm(eigenvectors, axis=0)
# Choose number of components
num_bases_list = [50, 100, 150, 200]
for num_bases in num_bases_list:
    start_time = time.time()
    # Project training and testing images onto PCA space
    train_projected = np.dot(eigenvectors[:, :num_bases].T, centered_train_images).real
    test_projected = np.dot(eigenvectors[:, :num_bases].T, centered_test_images).real
    # Initialize k-NN classifier
    knn = KNeighborsClassifier(n_neighbors=1)
    knn.fit(train_projected.T, train_labels.flatten())
    # Predict on test set
    predictions = knn.predict(test_projected.T)
    # Calculate accuracy
    accuracy = accuracy_score(test_labels.flatten(), predictions)
    print(f'Number of Components: {num_bases}, Recognition Accuracy: {accuracy:.2f}')
    # Confusion matrix
    cm = confusion_matrix(test_labels.flatten(), predictions)
    print("Confusion Matrix:\n", cm)
    # Measure time taken
    elapsed_time = time.time() - start_time
    print(f'Time taken for {num_bases} components: {elapsed_time:.2f} seconds')
    # Example Success and Failure Cases
    success_cases = np.where(predictions == test_labels.flatten())[0]
    failure_cases = np.where(predictions != test_labels.flatten())[0]
    print(f'Success Cases: {success_cases[:5]}')
    print(f'Failure Cases: {failure_cases[:5]}')
    # Visualize one success and one failure case with true and predicted images
    if len(success_cases) > 0 and len(failure_cases) > 0:
        fig, axes = plt.subplots(2, 2, figsize=(8, 8))
        # Success case
        success_idx = success_cases[0]
    axes[0, 0].imshow(test_images[:, success_idx].reshape(46, 56), cmap='gray') # True image
    axes[0, 0].set_title(f'True: {test_labels.flatten()[success_idx]}')
    axes[0, 0].axis('off')
    axes[0, 1].imshow(test_images[:, success_idx].reshape(46, 56), cmap='gray') # Predicted image
    axes[0, 1].set_title(f'Predicted: {predictions[success_idx]}')
    axes[0, 1].axis('off')
    # Failure case
    failure_idx = failure_cases[0]
    axes[1, 0].imshow(test_images[:, failure_idx].reshape(46, 56), cmap='gray') # True image
    axes[1, 0].set_title(f'True: {test_labels.flatten()[failure_idx]}')
    axes[1, 0].axis('off')
    axes[1, 1].imshow(test_images[:, failure_idx].reshape(46, 56), cmap='gray') # Predicted image
    axes[1, 1].set_title(f'Predicted: {predictions[failure_idx]}')
    axes[1, 1].axis('off')
    plt.tight_layout()
    plt.show()
```

## Code for 2

```
import numpy as np
import scipy.io
from sklearn.decomposition import IncrementalPCA, PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import mean_squared_error, accuracy_score
import time
# Load the dataset
# mat_data = scipy.io.loadmat('face.mat')
image_data = mat_data['X']
labels = mat_data['l'].flatten()
# Partition data as described
num_classes = 52
images_per_class = 10
train_images, train_labels, test_images, test_labels = [], [], [], []
for identity in range(1, num_classes + 1):
    indices = np.where(labels == identity)[0]
    np.random.shuffle(indices)
    train_indices = indices[:8]
    test_indices = indices[8:10]
    train_images.append(image_data[:, train_indices])
    train_labels.extend(labels[train_indices])
    test_images.append(image_data[:, test_indices])
    test_labels.extend(labels[test_indices])
train_images = np.concatenate(train_images, axis=1)
train_labels = np.array(train_labels)
test_images = np.concatenate(test_images, axis=1)
test_labels = np.array(test_labels)
# Divide training data into four subsets
subset_size = 104
subsets = [train_images[:, i:i + subset_size] for i in range(0, train_images.shape[1], subset_size)]
# Perform Incremental PCA
def perform_incremental_pca(subsets, n_components):
    ipca = IncrementalPCA(n_components=n_components)
    start_time = time.time()
    for subset in subsets:
        ipca.partial_fit(subset.T)
    training_time = time.time() - start_time
    return ipca, training_time
# Perform Batch PCA
def perform_batch_pca(images, n_components):
    pca = PCA(n_components=n_components)
    start_time = time.time()
    pca.fit(images.T)
    training_time = time.time() - start_time
    return pca, training_time
# Evaluate PCA methods
def evaluate_pca(pca_model, images, labels, test_images, test_labels):
    train_projected = pca_model.transform(images.T)
    test_projected = pca_model.transform(test_images.T)

    # Train k-NN classifier

    knn = KNeighborsClassifier(n_neighbors=1)

    knn.fit(train_projected, labels)

    predictions = knn.predict(test_projected)

    # Calculate accuracy

    accuracy = accuracy_score(test_labels, predictions)

    # Reconstruction error

    reconstructed_images =
pca_model.inverse_transform(train_projected)

    reconstruction_error = mean_squared_error(images.T,
reconstructed_images)

    return accuracy, reconstruction_error

# Incremental PCA

ipca, inc_training_time =
perform_incremental_pca(subsets, n_components=20)

inc_accuracy, inc_reconstruction_error =
evaluate_pca(ipca, train_images, train_labels,
test_images, test_labels)

# Batch PCA

batch_pca, batch_training_time =
perform_batch_pca(train_images, n_components=20)

batch_accuracy, batch_reconstruction_error =
evaluate_pca(batch_pca, train_images, train_labels,
test_images, test_labels)

# PCA on first subset

first_subset_pca, first_subset_training_time =
perform_batch_pca(subsets[0], n_components=20)

first_subset_accuracy, first_subset_reconstruction_error =
evaluate_pca(first_subset_pca, subsets[0],
train_labels[:subset_size], test_images, test_labels)

# Results

print(f"Incremental PCA Training Time:
{inc_training_time:.2f} seconds")

print(f"Batch PCA Training Time:
{batch_training_time:.2f} seconds")

print(f"First Subset PCA Training Time:
{first_subset_training_time:.2f} seconds")

print(f"Incremental PCA Accuracy: {inc_accuracy *
100:.2f}%")

print(f"Batch PCA Accuracy: {batch_accuracy *
100:.2f}%")

print(f"First Subset PCA Accuracy:
{first_subset_accuracy * 100:.2f}%")

print(f"Incremental PCA Reconstruction Error:
{inc_reconstruction_error:.4f}")

print(f"Batch PCA Reconstruction Error:
{batch_reconstruction_error:.4f}")

print(f"First Subset PCA Reconstruction Error:
{first_subset_reconstruction_error:.4f}")
```

## Code for 3A

```
import numpy as np

from sklearn.decomposition import PCA

from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis as LDA

from sklearn.neighbors import
KNeighborsClassifier

from sklearn.metrics import accuracy_score,
confusion_matrix

# Assuming train_images, train_labels,
test_images, and test_labels are already
prepared

# Ensure data is in the correct shape:
(number_of_samples, number_of_features)

train_images = train_images.T # Shape should
be (number_of_samples, number_of_features)

test_images = test_images.T # Shape should be
(number_of_samples, number_of_features)

# Parameters to test

pca_components = [30, 50, 70] # Example
values for PCA components

lda_components = [5, 10, 15] # Example values
for LDA components

n_neighbors = 1 # Number of neighbors for NN

# Function to calculate rank of a matrix

def matrix_rank(matrix):

    return np.linalg.matrix_rank(matrix)

# Store results

results = []

for M_pca in pca_components:

    for M_lda in lda_components:

        # Apply PCA

        pca = PCA(n_components=M_pca)
    train_pca = pca.fit_transform(train_images)

    test_pca = pca.transform(test_images)

    # Apply LDA

    lda = LDA(n_components=M_lda)

    train_lda = lda.fit_transform(train_pca, train_labels)

    test_lda = lda.transform(test_pca)

    # Calculate ranks of scatter matrices

    rank_sw = min(M_lda, len(np.unique(train_labels)) - 1) # Maximum possible rank for within-class

    rank_sb = matrix_rank(lda.scalings_ @ lda.scalings_.T) # Rank of between-class scatter matrix

    # Classification using NN

    knn = KNeighborsClassifier(n_neighbors=n_neighbors)

    knn.fit(train_lda, train_labels)

    predictions = knn.predict(test_lda)

    # Evaluate

    accuracy = accuracy_score(test_labels, predictions)

    conf_matrix = confusion_matrix(test_labels, predictions)

    results.append({

        'M_pca': M_pca,

        'M_lda': M_lda,

        'accuracy': accuracy,

        'rank_sw': rank_sw,

        'rank_sb': rank_sb,

        'conf_matrix': conf_matrix

    })

    print(f"M_pca: {M_pca}, M_lda: {M_lda}")

    print(f"Recognition Accuracy: {accuracy * 100:.2f}%")

    print(f"Rank of Within-class Scatter Matrix: {rank_sw}")

    print(f"Rank of Between-class Scatter Matrix: {rank_sb}")

    print("Confusion Matrix:")

    print(conf_matrix)

    print("\n")

# Example Success and Failure Cases

# Assuming you want to visualize or analyze specific cases

# For simplicity, this part is illustrative. Implement as needed based on your data handling.

# Discussion of Results

# Compare with previous experiments and discuss observations
```

**Code for 3b**

```
import numpy as np

from sklearn.decomposition import PCA

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score, confusion_matrix

from sklearn.utils import resample

# Parameters

M_pca = 50 # Number of PCA components

M_lda = 15 # Number of LDA components

n_neighbors = 1 # Number of neighbors for KNN

n_models = 17 # Number of models in the ensemble

# Function to train a single PCA-LDA model

def train_pca_lda_model(train_images, train_labels, test_images):

    # Apply PCA

    pca = PCA(n_components=M_pca)

    train_pca = pca.fit_transform(train_images)

    test_pca = pca.transform(test_images)

    # Apply LDA

    lda = LDA(n_components=M_lda)
    train_lda = lda.fit_transform(train_pca, train_labels)
    test_lda = lda.transform(test_pca)
    # Classification using NN
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(train_lda, train_labels)
    return knn.predict(test_lda)
# Bagging with randomization
ensemble_predictions = []
for _ in range(n_models):
    # Resample the training data with replacement
    resampled_images, resampled_labels =
resample(train_images, train_labels)
    predictions = train_pca_lda_model(resampled_images,
resampled_labels, test_images)
    ensemble_predictions.append(predictions)
# Majority voting (fusion rule)
ensemble_predictions = np.array(ensemble_predictions)
final_predictions = np.apply_along_axis(lambda x:
np.bincount(x).argmax(), axis=0,
arr=ensemble_predictions)
# Evaluate ensemble
ensemble_accuracy = accuracy_score(test_labels,
final_predictions)
ensemble_conf_matrix = confusion_matrix(test_labels,
final_predictions)
print(f"Ensemble Recognition Accuracy:
{ensemble_accuracy * 100:.2f}%")
print("Ensemble Confusion Matrix:")
print(ensemble_conf_matrix)
# Error analysis
individual_errors = []
for predictions in ensemble_predictions:
    error = 1 - accuracy_score(test_labels, predictions)
    individual_errors.append(error)
average_individual_error = np.mean(individual_errors)
committee_error = 1 - ensemble_accuracy
print(f"Average Error of Individual Models:
{average_individual_error * 100:.2f}%")
print(f"Error of the Committee Machine:
{committee_error * 100:.2f}%")
```

**Code for 5**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,
confusion_matrix, ConfusionMatrixDisplay
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from scipy.io import loadmat


# Load face dataset
data = loadmat('face.mat')


X = data['face_images']
y = data['labels'].ravel()


X = X.reshape(X.shape[0], -1)  # Flatten images


# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, stratify=y, random_state=42)


pca = PCA(n_components=100)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)


rf_clf = RandomForestClassifier(
    n_estimators=100,       # Number of trees
    max_depth=None,         # Max depth of the trees
    max_features='sqrt',    # Randomness
    random_state=42,
    n_jobs=-1
)
rf_clf.fit(X_train_pca, y_train)


# Predictions
y_pred = rf_clf.predict(X_test_pca)


# Evaluation
```

```
accuracy = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

# Display Results
print("Random Forest Accuracy:", accuracy)
ConfusionMatrixDisplay(cm).plot(cmap="Blues")
plt.title("Random Forest Confusion Matrix")
plt.show()
```