MACHINE LEARNING

FACE RECOGNITION

TEAM:

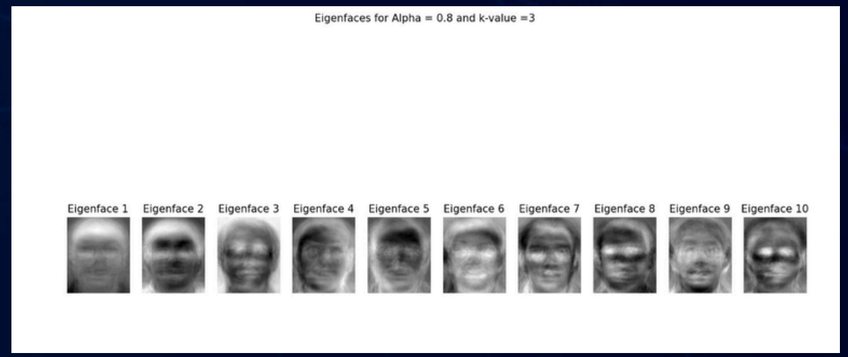
ZEANAB OSAMA : 2305289
MARIAM AHMED : 2305300
AHMED SADEK : 2305355

PCA

```python
import numpy as np
#Principal Component Analysis
def PCA(training_DataMatrix, alpha):
    # Step 1: Compute the mean
    mean = np.mean(training_DataMatrix, axis=0)

    # Step 2: Center the data (m)
    training_data_centralized = training_DataMatrix - mean

    # Step 3: Compute the covariance matrix
    cov_matrix = training_data_centralized @ training_data_centralized.T

    # Step 4: Compute the eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

1. Imports the NumPy library and aliases it as **np** for numerical operations
2. Comment indicating this is a Principal Component Analysis implementation.
3. Defines a function named **PCA** that takes two parameters:
   - **training_DataMatrix**: The input data matrix (samples x features)
   - **alpha**: Threshold for cumulative explained variance ratio (typically 0.95 for 95%)
4. Computes the mean vector of the data matrix along axis=0 (column-wise mean)
5. Centers the data by subtracting the mean from each feature (mean normalization)
6. Computes the covariance matrix using matrix multiplication (@ operator)
   - Note: This computes XX^T (samples x samples) rather than the traditional X^TX (features x features)
7. Computes eigenvalues and eigenvectors of the covariance matrix using NumPy's linear algebra module

python



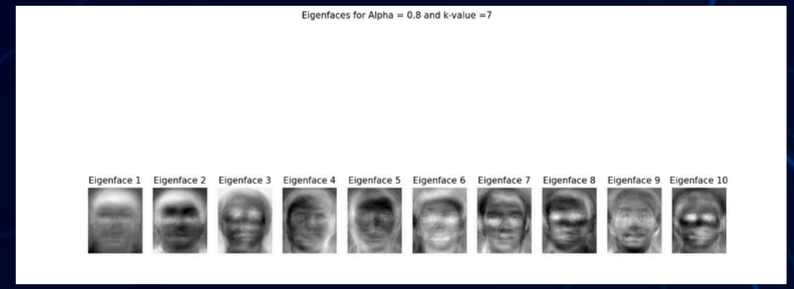

```
# Step 5: Sort the eigenvectors descendingly by eigenvalues
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

# Step 6: Compute the cumulative explained variance ratio
explained_variance_ratio = np.cumsum(eigenvalues) / np.sum(eigenvalues)

# Step 7: Determine the number of components to keep
no_components = np.argmax(explained_variance_ratio >= alpha) + 1
```

1. Sorts eigenvalues and corresponding eigenvectors in descending order:
   - np.argsort(eigenvalues) gets indices for ascending sort
   - [::-1] reverses the order to get descending sort
   - Applies the sorted indices to both eigenvalues and eigenvectors
2. Computes the cumulative explained variance ratio:
   - **np.cumsum** calculates cumulative sum of eigenvalues
   - Divides by total sum to get normalized ratios
   - Finds the minimum number of components needed to achieve the desired explained variance (alpha):
   - Creates boolean mask where ratios >= alpha
   - **np.argmax** finds first True occurrence (plus 1 for 1-based count)
   -

```python
# Step 8: Reduce the basis
eigenvectors_converted = training_data_centralized.T @ eigenvectors
eigenfaces = eigenvectors_converted / np.linalg.norm(eigenvectors_converted, axis=0)

# Step 9: Reduce the dimensionality of the data
projected_data = training_data_centralized @ eigenfaces[:, :no_components]

return mean, eigenfaces[:, :no_components], projected_data
```

Converts eigenvectors from sample space to feature space:
First line projects eigenvectors back to original feature space
Second line normalizes each eigenvector to unit length
rojects the centered data onto the principal components:

- Uses only the top **no_components** eigenvectors
- Result is the reduced-dimension representation
- Returns three values:
- **mean**: The mean vector of the original data
- Reduced set of eigenvectors (principal components)
- Projected data in the reduced space
-



INITIAL IMPORTS

```python
import numpy as np
#from PIL import Image
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from data_preprocessing import process
from pca import PCA
```

mports NumPy for numerical operations
(Commented out) PIL import for image processing
Matplotlib for visualization
Scikit-learn's KNN classifier
Custom process() function from data_preprocessing module
Your PCA implementation from pca module

PCA WITH HYPERPARAMETER TUNING

```
X_train, y_train, X_test, y_test = process()
#PCA (With Tuning)
# Perform PCA and classify for each alpha value
print("\nAccuracy for every value of alpha and k:")
alpha_values = [0.8, 0.85, 0.9, 0.95]
k_values = [1, 3, 5, 7]
accuracies = []

for alpha in alpha_values:
    mean_face, eigenfaces, projected_data = PCA(X_train, alpha)
    projected_train = (X_train - mean_face) @ eigenfaces
    projected_test = (X_test - mean_face) @ eigenfaces
```

Loads and splits the dataset into training and test sets using the process() function

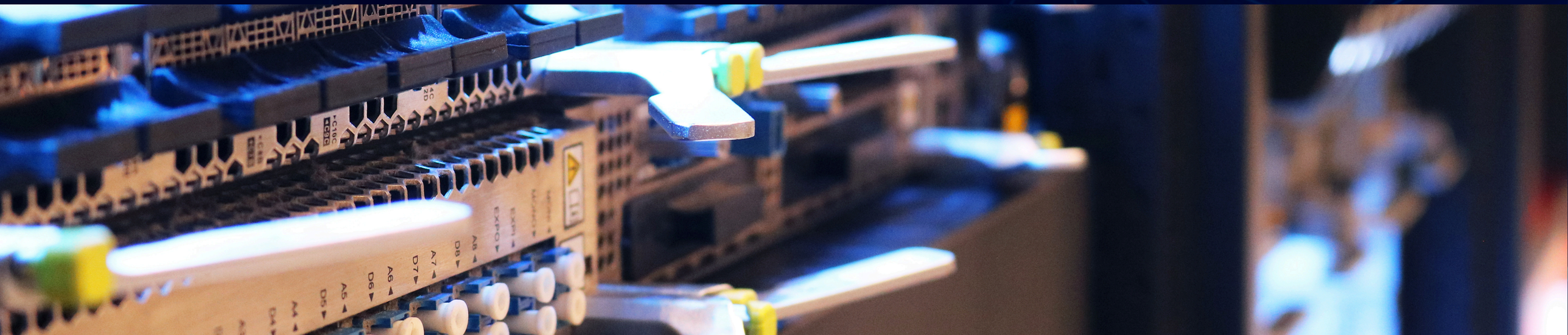
1. Prints header for tuning results
2. Defines alpha values (variance thresholds) to test
3. Defines k values (neighbors) for KNN
4. Initializes empty list to store accuracies
5. Loops through each alpha value
6. Performs PCA on training data
7. Projects both training and test data using PCA components
8.

```
Label vector shape: (400,)
Data Matrix shape: (400, 10304)
Data Matrix shape:: (400, 10304)
Label vector shape: (400,)

Training set shapes:
X_train shape: (200, 10304)
y_train shape: (200,)

Test set shapes:
X_test shape: (200, 10304)
y_test shape: (200,)

Accuracy for every value of alpha and k:
Alpha: 0.8, K: 1, Accuracy: 95.00%
Alpha: 0.8, K: 3, Accuracy: 89.50%
Alpha: 0.8, K: 5, Accuracy: 85.00%
Alpha: 0.8, K: 7, Accuracy: 80.50%
Alpha: 0.85, K: 1, Accuracy: 95.00%
```

Nested loop through each k value
Initializes KNN classifier with current k
Trains classifier on projected training data
Makes predictions on projected test data
Calculates accuracy percentage
Stores accuracy result

1. Prints current alpha, k, and accuracy
2. Creates figure for 10 eigenfaces
3. Loops through first 10 principal components
4. Reshapes each eigenvector to image dimensions (112×92)
5. Displays each eigenface
6. Sets title and removes axes
7. Shows the plot

```python
for k in k_values:
    classifier = KNeighborsClassifier(n_neighbors=k)
    classifier.fit(projected_train, y_train)
    y_pred = classifier.predict(projected_test)
    accuracy = np.mean(y_pred == y_test) * 100
    accuracies.append(accuracy)
    print(f"Alpha: {alpha}, K: {k}, Accuracy: {accuracy:.2f}%")
    # Plot the first 10 eigenfaces for this alpha
    fig, axs = plt.subplots(1, 10, figsize=(16, 10))
    for i in range(10):
        image_array = np.reshape(eigenfaces[:, i], (112, 92))
        axs[i].imshow(image_array, cmap="gray")
        axs[i].set_title("Eigenface " + str(i + 1))
        axs[i].axis("off")
    plt.suptitle(f"Eigenfaces for Alpha = {alpha} and k-value ={k}")
    plt.show()
```



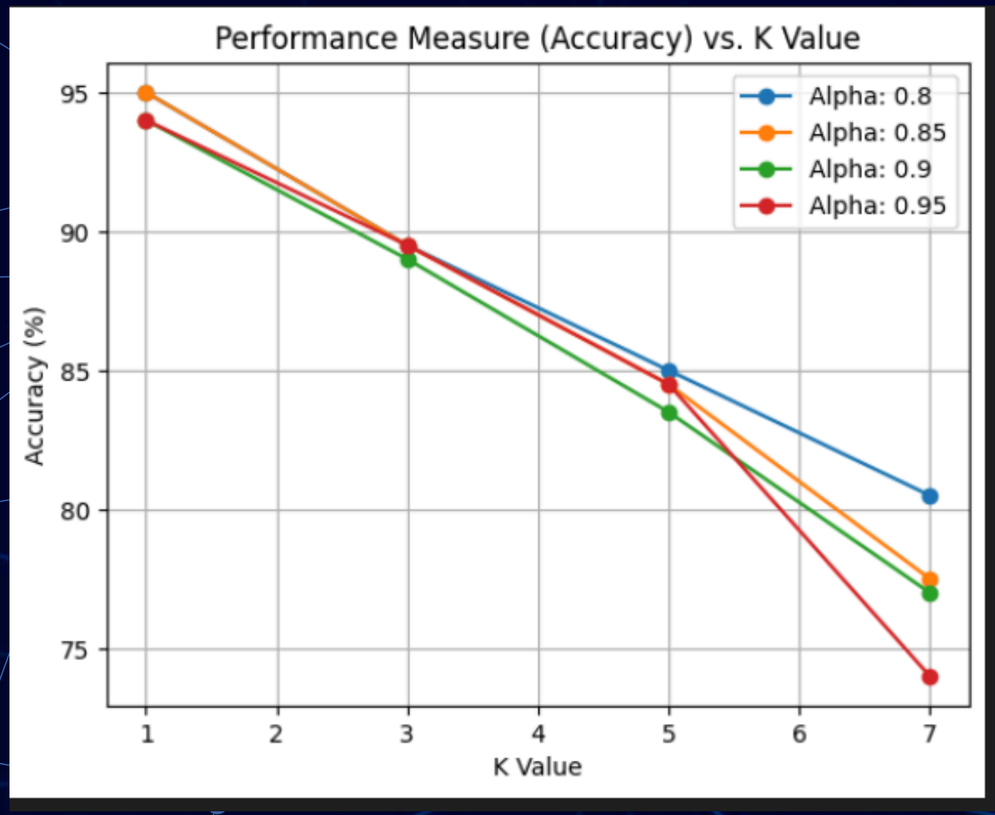
Reshapes accuracy list into 2D array (alphas × k-values)

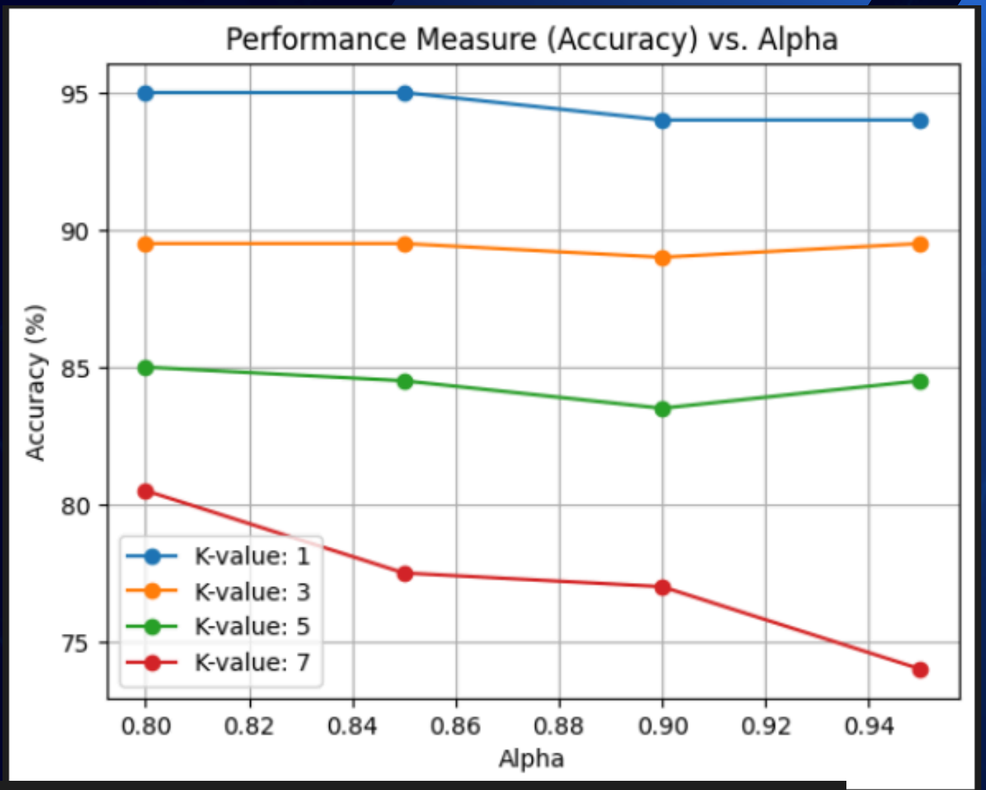
1. Plots accuracy vs k-values for each alpha
2. Adds title and axis labels
3. Shows legend and grid
4. Displays the plot
5. Plots accuracy vs alpha for each k-value
6. Similar formatting as previous plot
7. Displays the comparative results
8.

```python
accuracies = np.array(accuracies).reshape(len(alpha_values), len(k_values))

# Plot the performance measure (accuracy) against K value for each alpha
for i, alpha in enumerate(alpha_values):
    plt.plot(k_values, accuracies[i], marker='o', label=f"Alpha: {alpha}")
plt.title("Performance Measure (Accuracy) vs. K Value")
plt.xlabel("K Value")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.grid(True)
plt.show()

# It's the same plot but this mesure the performance (accuracy) against alpha for each K value
for i, k in enumerate(k_values):
    plt.plot(alpha_values, accuracies[:, i], marker='o', label=f"K-value: {k}")
plt.title("Performance Measure (Accuracy) vs. Alpha")
plt.xlabel("Alpha")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.grid(True)
plt.show()
```

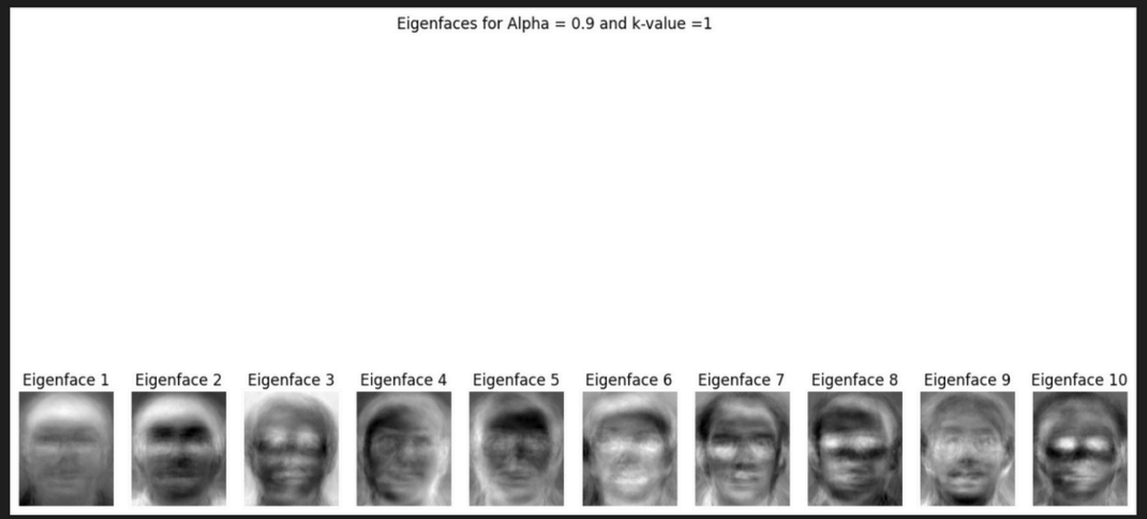
The code evaluates PCA performance by testing different variance thresholds (alpha)
For each PCA configuration, it tests different KNN neighbor counts (k)
Visualizes both the eigenfaces and accuracy trends
Uses two complementary visualization approaches to analyze the hyperparameter space
The commented section at the top shows a simpler version without k-tuning
The implementation provides comprehensive evaluation of how PCA dimensionality reduction affects classification performance with different KNN configuratio

OUT PUT:






```
Alpha: 0.9, K: 1, Accuracy: 94.00%
```



LDA

- This function performs Linear Discriminant Analysis (LDA) on the training data to compute a projection matrix that maximizes class separation.Explanation: Removes redundant dimensions from y_train (e.g., converts shape (n, 1) to (n,)).
- For each class (1 to 40), compute the mean vector of all samples belonging to that class.
- **X_train[y_train == i]** selects all rows in **X_train** where the label is **i**.
- **np.mean(..., axis=0)** computes the mean along columns (feature-wise mean).
- For each class (1 to 40), count the number of samples (np.sum(y_train == i)).
- Computes the global mean vector of all training data (mean of all samples, regardless of class).

```python
def LDA(X_train, y_train):
    y_train = np.squeeze(y_train)
    class_means = np.array([np.mean(X_train[y_train == i], axis=0) for i in range(1, 41)])
    class_sizes = np.array([np.sum(y_train == i) for i in range(1, 41)])

    # Compute overall mean
    # We Need To Compute Overall Mean To
    # Centering the Data (class_means[i(current_class) - 1(because it 0 index)] - overall_mean)
    # Centering The Data Helping For Removing Any Bias Or Shift In The Data Distribution
    overall_mean = np.mean(X_train, axis=0)
```

S_W is initialized as a zero matrix of shape (n_features, n_features).
For each class:
class_data selects samples belonging to the current class.
centered_data centers the data by subtracting the class mean.
np.dot(centered_data.T, centered_data) computes the covariance matrix for the class.

- S_W accumulates the sum of all class covariances.Adds a small value (1e-7) to the diagonal of S_W to ensure it is invertible (avoiding numerical instability).

```
# Sum All the Covariances = S_W
S_W = np.zeros((X_train.shape[1], X_train.shape[1]))
for i in range(1, 41):
    # Use boolean index to select rows from X_train
    class_data = X_train[y_train == i]
    centered_data = class_data - class_means[i - 1]
    S_W += np.dot(centered_data.T, centered_data)

# Regularize S_W
S_W += 1e-7 * np.identity(X_train.shape[1])
```

```python
S_B = np.zeros((X_train.shape[1], X_train.shape[1]))
for i in range(1, 41):
    # Use boolean index to select rows from X_train
    # Select All Vector Images In The Current Class
    class_data = X_train[y_train == i]
    # Subtracted From Overall_mean
    class_diff = class_means[i - 1] - overall_mean
    S_B += class_sizes[i - 1] * np.outer(class_diff, class_diff)

# Solve generalized eigenvalue problem
eigenvalues, eigenvectors = np.linalg.eig(np.dot(np.linalg.inv(S_W), S_B))
```

S_B is initialized as a zero matrix.
For each class:
class_diff computes the difference between the class mean and the overall mean.
np.outer(class_diff, class_diff) computes the outer product (rank-1 matrix).

- S_B accumulates the weighted sum (bySolves the generalized eigenvalue problem S_W^{-1} * S_B * v = λ * v.
- Returns eigenvalues (eigenvalues) and eigenvectors (eigenvectors).

```python
idx = np.argsort(eigenvalues)[::-1]
sorted_eigenvectors = eigenvectors[:, idx]

# Take Only The First39 Dominant eigenvectors [In The PDF He Want The First Dominant Eigenvectors]
projection_matrix = sorted_eigenvectors[:, :39]
return np.real(projection_matrix)
```

Sorts eigenvalues in descending order ([::-1] reverses the order).
Reorders eigenvectors accordingly.

- Selects the top 39 eigenvectors (most discriminative directions).
- **np.real()** ensures the result is real-valued (discards imaginary parts if any).

- Transforms training_data using the projection matrix (training_data @ projection_matrix).
- ransforms **test_data** using the same projection matrix.
- Returns the transformed training and test data.
-

```
def LDA_projected_data(training_data,test_data,projection_matrix):
    projected_X_train = np.dot(training_data, projection_matrix)
    projected_X_test = np.dot(test_data, projection_matrix)
    return projected_X_train, projected_X_test
```

```
def LDA2 (train_data, train_labels, k=1):
    # mean of each class
    mean1 = np.mean(train_data[train_labels.ravel() == 1], axis=0)
    mean0 = np.mean(train_data[train_labels.ravel() == 0], axis=0)
```

Computes mean vectors for class 1 and class 0.

```
# within class scatter matrix
Sw = np.dot((train_data[train_labels.ravel() == 1] - mean1).T,
            (train_data[train_labels.ravel() == 1] - mean1)) + np.dot((train_data[train_labels.ravel() == 0] - mean0).T,
                                                                     (train_data[train_labels.ravel() == 0] - mean0))
# between class scatter matrix
Sb = np.dot((mean1 - mean0).reshape(-1,1), (mean1 - mean0).reshape(-1,1).T)

# calculate eigenvalues and eigenvectors
eig_values, eig_vectors = np.linalg.eigh(np.dot(np.linalg.inv(Sw), Sb))
eig_values = np.real( eig_values)
eig_vectors = np.real( eig_vectors)
```

Computes Sw as the sum of covariances for both classes
Computes **Sb** as the outer product of the difference between class means.
olves **Sw^{-1} * Sb * v = λ * v** using **eigh** (for Hermitian matrices).

```
idx = np.argsort(eig_values)[::-1]
eig_values = eig_values[idx]
eig_vectors = eig_vectors[:,idx]
return eig_vectors[:,:k]
```

Sorts eigenvalues and eigenvectors in descending order.

- Returns the top **k** eigenvectors (default **k=1**).

LDA(): Multi-class LDA (40 classes) with regularization and eigenvalue decomposition.
LDA_projected_data(): Projects data onto the LDA subspace.
LDA2(): Binary LDA (2 classes) with simplified computations.
This code is commonly used for dimensionality reduction in classification tasks, ensuring

```python
from lda import LDA, LDA_projected_data
from data_preprocessing import process
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
```

mports custom LDA functions (LDA, LDA_projected_data) from lda.py.
Imports process() from data_preprocessing.py (likely loads/splits the dataset).
Imports scikit-learn's KNeighborsClassifier for k-NN classification and accuracy_score for evaluation.
Imports pandas for results visualization.

```python
def Test_LDA(k , LDA_projection_matrix):
    projected_X_train, projected_X_test = LDA_projected_data(X_train,X_test,LDA_projection_matrix)
```

Projects X_train and X_test into the LDA subspace using the provided LDA_projection_matrix.

```python
def Test_LDA(k , LDA_projection_matrix):
    projected_X_train, projected_X_test = LDA_projected_data(X_train,X_test,LDA_projection_matrix)
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(projected_X_train, y_train.ravel())
    y_pred = knn.predict(projected_X_test)
    accuracy = accuracy_score(y_test, y_pred.ravel())
    return accuracy

X_train, y_train, X_test, y_test = process()
LDA_projection_matrix = LDA(X_train,y_train)
print(LDA_projection_matrix.shape)
print("LDA Accuracy: " + str(Test_LDA(1 , LDA_projection_matrix))) # ====> 0.965

LDA_projection_matrix = LDA(X_train,y_train)
print(LDA_projection_matrix.shape)
print(LDA_projection_matrix)
print("LDA Accuracy: " + str(Test_LDA(1 , LDA_projection_matrix))) # ====> 0.965
```

nitializes a k-NN classifier with k neighbors.
Fits the model on the projected training data (projected_X_train) and flattened labels (y_train.ravel()).

- Predicts labels for **projected_X_test**.
- Computes accuracy by comparing predictions (**y_pred**) to true labels (**y_test**).
- Calls **process()** to load and split the dataset into training/testing sets.
- Computes the LDA projection matrix using the training data.
- Prints the shape of the projection matrix (e.g., **(10304, 39)**).
- Evaluates LDA + 1-NN and prints accuracy (reported as 96.5%).
- repeats LDA computation and evaluation (same accuracy expected).
-
-

- Defines k_values to test for k-NN.
  Initializes results list to store accuracies.
  Projects the data once (reused for all k values).
  For each **k**:
  - Initializes k-NN with **distance** weighting (closer neighbors have more influence).
  - Trains the classifier and computes accuracy.
  - reates a DataFrame to display accuracies for each **k**.
  - Sets the index name to "**k**" for clarity.
  -

```python
k_values = [1, 3, 5, 7, 9] # HyperParameters

# Initialize a list to store the results
results = []
projected_X_train, projected_X_test = LDA_projected_data(X_train,X_test,LDA_projection_matrix)
# Loop over the values of k
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k, weights="distance")
    knn.fit(projected_X_train, y_train.ravel())
    y_pred = knn.predict(projected_X_test)
    accuracy = accuracy_score(y_test, y_pred.ravel())
    results.append({"accuracy": accuracy})

# Convert the results to a DataFrame
df = pd.DataFrame(results, index=k_values)
df.index.name = "k"
print(df)
```

the output:

```
Label vector shape: (400,)
Data Matrix shape: (400, 10304)
Data Matrix shape:: (400, 10304)
Label vector shape: (400,)

Training set shapes:
X_train shape: (200, 10304)
y_train shape: (200,)

Test set shapes:
X_test shape: (200, 10304)
y_test shape: (200,)
```

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from pca import PCA
from lda import LDA2
```

os: For interacting with the operating system (e.g., reading files).
numpy (np): For numerical operations (arrays, matrices).
matplotlib.pyplot (plt): For plotting graphs/images.
PIL.Image: For image processing (opening, converting, resizing).
KNeighborsClassifier: K-Nearest Neighbors classifier from scikit-learn.
accuracy_score: To compute classification accuracy.
PCA & LDA2: Custom PCA and LDA implementations (likely for dimensionality reduction).

```python
def load_images(path):
    images = []
    labels = []
    #nonfaces --> change to greyscal , resize and flatten
    if "non" in path:
        for i, dir in enumerate(os.listdir(path)):
            for file in os.listdir(os.path.join(path, dir)):
                img = Image.open(os.path.join(path, dir, file)).convert('L')
                img = img.resize((92,112))
                images.append(np.array(img).flatten())
                labels.append(i+1)

    #faces --> flatten
    else:
        for i, dir in enumerate(os.listdir(path)):
            for file in os.listdir(os.path.join(path, dir)):
                img = Image.open(os.path.join(path, dir, file))
                images.append(np.array(img).flatten())
                labels.append(i+1)
    return np.array(images), np.array(labels).reshape(-1,1)
```

images: Stores flattened image arrays.
labels: Stores corresponding labels.

1. **"non" in path**: Checks if the path contains non-face images.
2. **os.listdir(path)**: Lists subdirectories (e.g., different non-face categories).
3. **Image.open(...).convert('L')**: Opens an image and converts it to grayscale.
4. **img.resize((92,112))**: Resizes to match face image dimensions.
5. **flatten()**: Converts the 2D image to a 1D array.
6. Labels: Assigned incrementally (**i+1**).
7. Similar to non-faces, but no grayscale conversion or resizing (assumed already preprocessed).
8. Converts lists to NumPy arrays and reshapes labels to column vectors.
9.
10.

```
faces, labels = load_images('Data')
non_faces, non_labels = load_images('nonfaces')

'''
binary labels --> 1 for faces, 0 for non-faces
'''
faces_labels = np.ones((len(faces),1))
non_faces_labels = np.zeros((len(non_faces),1))
print("Faces= ",faces.shape, faces_labels.shape)
print("NonFaces= ",non_faces.shape, non_faces_labels.shape)
```

faces_labels: All 1 (face).
non_faces_labels: All 0 (non-face).

- **np.arange**: Creates an array of indices.
- **np.random.shuffle**: Randomly shuffles indices.
- **data[idx]**: Returns shuffled data and labels.

```
def shuffle_data(data, labels):
    idx = np.arange(data.shape[0])
    np.random.shuffle(idx)
    return data[idx], labels[idx]

faces, labels = shuffle_data(faces, labels)
non_faces, non_labels = shuffle_data(non_faces, non_labels)
```

```python
def plot_data(faces, labels, n=100):
    num_rows = n // 10 #10 columns
    fig, axs = plt.subplots(num_rows, 10, figsize=(15, 1.5 * num_rows), gridspec_kw={'hspace': 0.3})
    axs = axs.ravel()
    for i in range(n):
        axs[i].imshow(faces[i].reshape((112, 92)), cmap="gray")
        axs[i].set_title(f"Label: {labels[i]}")
        axs[i].axis("off")
    plt.show()

plot_data(faces, labels,20) #show 20 images
plot_data(non_faces, non_labels,20)
```

plt.subplots: Creates a grid of subplots.
imshow: Displays images in grayscale.
set_title: Shows the label above each image.
axis("off"): Removes axes for clarity.

alpha: Fraction of data for training (e.g., 0.5 for 50%).
non_face_precentage_in_train: Controls non-faces in training.

- Uses stride slicing to split data.
- Randomly splits data into train/test sets.
- Concatenates face and non-face data.

```python
# function to split the data into training and testing which alpha is the percentage of the training data
def split_data(faces, faces_labels, non_faces, non_faces_labels, non_faces_count, alpha, non_face_precentage_in_train=1):
    if alpha == 0.5:
        faces_train = faces[::2]
        faces_train_labels = faces_labels[::2]
        faces_test = faces[1::2]
        faces_test_labels = faces_labels[1::2]

        non_faces_train = non_faces[:int(non_faces_count*non_face_precentage_in_train):2]
        non_faces_train_labels = non_faces_labels[:int(non_faces_count*non_face_precentage_in_train):2]

        non_faces_test = non_faces[1:non_faces_count:2]
        non_faces_test_labels = non_faces_labels[1:non_faces_count:2]
    else:
        n = len(faces) #400 faces
        n_train = int(n*alpha) #no. faces in train
        idx = np.random.permutation(n) #random index (400)
        train_idx = idx[:n_train] #select first (n_train) index
        test_idx = idx[n_train:] #select the remaining elements
        faces_train = faces[train_idx]
        faces_train_labels = faces_labels[train_idx]
        faces_test = faces[test_idx]
        faces_test_labels = faces_labels[test_idx]

        n = non_faces_count
        n_train = int(n*alpha) #no. nonfaces in train
        idx = np.random.permutation(n)
        train_idx = idx[:n_train]
        test_idx = idx[n_train:]
        non_faces_train = non_faces[train_idx]
        non_faces_train_labels = non_faces_labels[train_idx]
        non_faces_test = non_faces[test_idx]
        non_faces_test_labels = non_faces_labels[test_idx]
```

PCA & Eigenfaces

```
mean, space, projected_data= PCA(train_data,0.85)
train_projected = (train_data - mean) @ space
test_projected = (test_data - mean) @ space

#plot eigen faces
def plot_eigenfaces(eigenvectors, n=10):
    num_rows = n // 5
    _, axs = plt.subplots(num_rows, 5, figsize=(15, 3 * num_rows), gridspec_kw={'hspace': 0.3})
    axs = axs.ravel()
    for i in range(n):
        axs[i].imshow(eigenvectors[:, i].reshape((112, 92)), cmap="gray")
        axs[i].set_title(f"EigenImages {i+1}")
        axs[i].axis("off")
    plt.show()
plot_eigenfaces(space, 10)
```

1. PCA(train_data, 0.85):
   - Computes PCA retaining 85% variance.
   - Returns mean, eigenvectors (space), and projected data.
2. Projection:
   - Centers data (train_data - mean) and projects onto PCA space.
   - Displays the first **n** eigenvectors (principal components) as images.
   -

```python
def knn_classifier(train_data, train_labels, test_data, test_labels, k=1):
    knn = KNeighborsClassifier( n_neighbors=1, weights='distance')
    knn.fit( train_data, train_labels.ravel() )
    return accuracy_score(test_labels, knn.predict(test_data).ravel()), knn.predict(test_data).ravel()

print("Accuracy of KNN classifier with k=1:", knn_classifier(train_projected, train_labels, test_projected, test_labels, 1)[0])


# Compute the number of unique classes
```

- NeighborsClassifier:
  - n_neighbors=1: Uses 1-NN (nearest neighbor).
  - weights='distance': Weights points by inverse distance.
- fit(): Trains the model on train_data and flattened train_labels.
- accuracy_score: Computes accuracy between true test_labels and predictions.
- Returns: Accuracy and predictions.
- Prints accuracy of 1-NN on PCA-projected data.
-

```python
num_classes = len(np.unique(train_labels))

# Determine the number of dominant eigenvectors for LDA
'''
the number of dominant eigenvectors used in LDA corresponds to the number of classes minus one,
 or the minimum between the number of classes and the dimensionality of the feature space.
'''
num_eigenvectors_lda = min(num_classes - 1, train_data.shape[1])  # Number of features can also be used instead of train_data.shape[1]

print("Number of dominant eigenvectors used for LDA:", num_eigenvectors_lda)

lda_space = LDA2(train_data, train_labels)
train_lda_projected = np.dot(train_data, lda_space) # project train
test_lda_projected = np.dot(test_data, lda_space) # project test
print("Accuracy of KNN classifier with k=1 after LDA:", knn_classifier(train_lda_projected, train_labels, test_lda_projected, test_labels)[0])
```

- num_classes: Counts unique labels (binary: faces/non-faces → 2).
- num_eigenvectors_lda:
  - LDA reduces to C-1 dimensions (where C = number of classes).
  - Here, C=2 → 1 eigenvector.
  - **LDA2**: Custom LDA function (likely computes optimal projection for class separation).
  - Projects train/test data onto LDA space using **np.dot**.
  - Evaluates 1-NN accuracy on LDA-projected data.
  -
  -

```python
def plot_failure_and_success(data, labels, predictions, n=10):
    failure_idx = np.where(predictions != labels)[1]
    success_idx = np.where(predictions == labels)[1]
    num_rows = n // 5
    fig, axs = plt.subplots(num_rows, 5, figsize=(15, 3 * num_rows), gridspec_kw={'hspace': 0.3})
    axs = axs.ravel()

    for i in range(n):
        if i < n/2:
            axs[i].imshow(data[failure_idx[i]].reshape((112, 92)), cmap="gray")
            axs[i].set_title(f"Predicted: {predictions[failure_idx[i]]}, Actual: {labels[0,failure_idx[i]]} \n failure")
        else:
            axs[i].imshow(data[success_idx[i-len(failure_idx)]].reshape((112, 92)), cmap="gray")
            axs[i].set_title(f"Predicted: {predictions[success_idx[i-len(failure_idx)]]}, Actual: {labels[0,success_idx[i-len(failure_idx)]]}\nsuccess")
        axs[i].axis("off")
    plt.show()

plot_failure_and_success(test_data, test_labels.reshape(1,-1), knn_classifier(train_lda_projected, train_labels, test_lda_projected, test_labels, 1)[1], 10)
```

- failure_idx: Indices where predictions ≠ true labels.
- success_idx: Indices where predictions match labels.
- Plots n images (half failures, half successes) with titles showing predicted/actual labels.

Visualizes misclassified and correctly classified examples after LDA.

- aries the number of non-faces (from **steps** to total, incrementing by **steps**).
- For each step:
  - Splits data (50% train/test).
  - Applies PCA/LDA.
  - Computes 1-NN accuracy.
- Plots accuracy vs. non-faces count.

```python
def plot_acc_vs_non_faces(algorithm,faces,faces_labels, non_faces,non_faces_labels,steps=50):
    acc = []
    n=len(non_faces)
    for i in range(steps,n+steps,steps):
        train_data, train_labels, test_data, test_labels = split_data(faces, faces_labels, non_faces, non_faces_labels,i,0.5,1)
        train_data, train_labels = shuffle_data( train_data,train_labels)
        test_data, test_labels = shuffle_data( test_data,  test_labels)
        if algorithm==0:
            mean, space, projected_data= PCA(train_data70,0.85)
            train_projected = (train_data70 - mean) @ space
            test_projected = (test_data70 - mean) @ space
            acc.append(knn_classifier(train_projected, train_labels, test_projected, test_labels, 1)[0]*100)
        else:
            lda_space = LDA2(train_data, train_labels)
            train_lda_projected = np.dot(train_data, lda_space)
            test_lda_projected = np.dot(test_data, lda_space)
            acc.append(knn_classifier(train_lda_projected, train_labels, test_lda_projected, test_labels, 1)[0]*100)

    plt.plot(range(steps,n+steps,steps),acc)
    plt.xlabel("Number of non faces")
    plt.ylabel("Accuracy")
    plt.show()
```

```
Faces=  (400, 10304) (400, 1)
NonFaces=  (550, 10304) (550, 1)
50% Train:  (400, 10304) (400, 1)
50% Test:  (400, 10304) (400, 1)
Accuracy of KNN classifier with k=1: 0.9375
Number of dominant eigenvectors used for LDA: 1
```

- Tests bias when non-faces dominate training data:
  - Varies the percentage of non-faces in training (from low to 100%).
  - Uses np.linspace for smooth interpolation.
  - Plots accuracy curve with points, revealing potential imbalance issues.

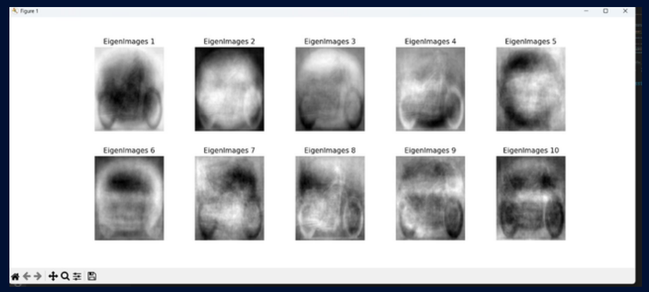
Compares different splits:

- 70% train may show higher accuracy due to more training data.
- 50% split is more balanced but may underfit if data is scarce

```python
train_data70, train_labels70, test_data70, test_labels70 = split_data(faces, faces_labels, non_faces, non_faces_labels, 400, 0.7, 1)
train_data50, train_labels50, test_data50, test_labels50 = split_data(faces, faces_labels, non_faces, non_faces_labels, 400, 0.5, 1)

train_data70, train_labels70 = shuffle_data(train_data70, train_labels70)
test_data70, test_labels70 = shuffle_data(test_data70, test_labels70)

train_data50, train_labels50 = shuffle_data(train_data50, train_labels50)
test_data50, test_labels50 = shuffle_data(test_data50, test_labels50)
```

1. PCA vs. LDA:
   - PCA maximizes variance (unsupervised).
   - LDA maximizes class separation (supervised).
2. Data Balance:
   - Accuracy drops if non-faces dominate training (class imbalance).
3. Train/Test Split:
   - Larger training sets generally improve accuracy but risk overfitting if validation is weak.

the output:



QUESTIONS:

PCA

1. What is Principal Component Analysis (PCA)?

PCAis a dimensionality reduction technique used in statistics and machine learning to transform high-dimensional data into a lower-dimensional representation, preserving the most important information.

2. What is Principal Component?

Principal components represent the directions of the data that explain a maximal amount of variance. The lines that capture most information of the data. The relationship between variance and information here, is that, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more information it has.

QUESTIONS:

PCA

3. What is Covariance Matrix?

The covariance matrix is a p × p symmetric matrix (where p is the number of dimensions).It’s actually the sign of the covariance that matters: If positive then: the two variables increase or decrease together (correlated) If negative then: one increases when the other decreases (Inversely correlated)

4. What are Eigenvectors and Eigenvalues?

The eigenvectors of the covariance matrix are actually the directions of the axes where there is the most variance (most information) and that we call Principal Components. And eigenvalues are the coefficients attached to eigenvectors, which give the amount of variance carried in each Principal Component. Their number is equal to the number of dimensions of the data. For example, for a 3-dimensional data set, there are 3 variables, therefore there are 3 eigenvectors with 3 corresponding eigenvalues.

QUESTIONS:

PCA

5. Disadvantages of Principal Component Analysis

1. Interpretation of Principal Components: The principal components created by Principal Component Analysis are linear combinations of the original variables, and it is often difficult to interpret them in terms of the original variables. This can make it difficult to explain the results of PCA to others.
2. Data Scaling: Principal Component Analysis is sensitive to the scale of the data. If the data is not properly scaled, then PCA may not work well. Therefore, it is important to scale the data before applying Principal Component Analysis.
3. Information Loss: Principal Component Analysis can result in information loss. While Principal Component Analysis reduces the number of variables, it can also lead to loss of information. The degree of information loss depends on the number of principal components selected. Therefore, it is important to carefully select the number of principal components to retain.

QUESTIONS:

PCA

4. Non-linear Relationships: Principal Component Analysis assumes that the relationships between variables are linear. 19 However, if there are non-linear relationships between variables, Principal Component Analysis may not work well.

5. Overfitting: Principal Component Analysis can sometimes result in overfitting , which is when the model fits the training data too well and performs poorly on new data. This can happen if too many principal components are used or if the model is trained on a small dataset.

QUESTIONS:

LDA

Whyis the number of dominant eigenvectors used for LDA = 1?

Number of Classes: In a binary classification problem (faces vs. non-faces), there are two classes: faces and non-faces.Number of Dominant Eigenvectors for LDA: In LDA, the number of dominant eigenvectors used is often limited by the number of unique classes minus one. This is because LDA seeks to find discriminant directions in the data that maximize class separability. In a binary classification problem, using one dominant eigenvector is common since it's the maximum number allowed by the formula.
(unique classes- 1)