Tutorial 3

LDA of 2×2 matrix using standard library

```
import numpy as np

# Define the 2x2 matrix
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])

# Define the class labels
y = np.array([0, 0, 1, 1])

# Calculate the mean of each class
mean_0 = np.mean(X[y == 0], axis=0)
mean_1 = np.mean(X[y == 1], axis=0)

# Calculate the overall mean
mean_overall = np.mean(X, axis=0)

# Calculate the within-class scatter matrix
Sw = np.dot((X[y == 0] - mean_0).T, (X[y == 0] - mean_0)) + np.dot((X[y == 1] - mean_1).T, (X[y
== 1] - mean_1))

# Calculate the between-class scatter matrix
Sb = np.dot((mean_0 - mean_overall).reshape(-1, 1), (mean_0 - mean_overall).reshape(1, -1))
+ np.dot((mean_1 - mean_overall).reshape(-1, 1), (mean_1 - mean_overall).reshape(1, -1))

# Calculate the eigenvalues and eigenvectors of the generalized eigenvalue problem
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw).dot(Sb))

# Sort the eigenvalues and eigenvectors in descending order
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

# Select the top k eigenvectors (k=1 in this case)
k = 1
lda_components = eigenvectors[:, :k]

# Project the original data onto the LDA components
X_lda = np.dot(X, lda_components)

print("LDA Components:")
print(lda_components)
```

print("\nProjected Data:")
print(X_lda)
```

This implementation performs the following steps:

1. Calculates the mean of each class and the overall mean.
2. Computes the within-class scatter matrix (Sw) and the between-class scatter matrix (Sb).
3. Solves the generalized eigenvalue problem to obtain the eigenvalues and eigenvectors.
4. Sorts the eigenvalues and eigenvectors in descending order.
5. Selects the top k eigenvectors (k=1 in this case).
6. Projects the original data onto the LDA components.

<u>LDA of 2×2 matrix using normal matrix method</u>

```
# Define the function to calculate the mean of each class
def calculate_mean(X, y):
    # Calculate the mean of class 0
    mean_0 = [(X[0][0] + X[1][0]) / 2, (X[0][1] + X[1][1]) / 2]
    # Calculate the mean of class 1
    mean_1 = [(X[2][0] + X[3][0]) / 2, (X[2][1] + X[3][1]) / 2]
    return mean_0, mean_1

# Define the function to calculate the scatter matrices
def calculate_scatter_matrices(X, y, mean_0, mean_1):
    # Initialize the within-class scatter matrix (Sw)
    Sw = [[0, 0], [0, 0]]
    # Initialize the between-class scatter matrix (Sb)
    Sb = [[0, 0], [0, 0]]

    # Calculate the scatter matrices
    for i in range(len(X)):
        if y[i] == 0:
            # Calculate the within-class scatter matrix for class 0
            Sw[0][0] += (X[i][0] - mean_0[0]) ** 2
            Sw[0][1] += (X[i][0] - mean_0[0]) * (X[i][1] - mean_0[1])
            Sw[1][0] += (X[i][0] - mean_0[0]) * (X[i][1] - mean_0[1])
            Sw[1][1] += (X[i][1] - mean_0[1]) ** 2
        else:
            # Calculate the between-class scatter matrix
            Sb[0][0] += (mean_0[0] - mean_1[0]) ** 2
```
```

```
        Sb[0][1] += (mean_0[0] - mean_1[0]) * (mean_0[1] - mean_1[1])
        Sb[1][0] += (mean_0[0] - mean_1[0]) * (mean_0[1] - mean_1[1])
        Sb[1][1] += (mean_0[1] - mean_1[1]) ** 2

    return Sw, Sb

# Define the function to calculate the eigenvalues
def calculate_eigenvalues(Sw, Sb):
    # Calculate the determinant of Sw
    det_Sw = Sw[0][0] * Sw[1][1] - Sw[0][1] * Sw[1][0]
    # Calculate the inverse of Sw
    inv_Sw = [[Sw[1][1] / det_Sw, -Sw[0][1] / det_Sw], [-Sw[1][0] / det_Sw, Sw[0][0] / det_Sw]]

    # Calculate the matrix A
    A = [[inv_Sw[0][0] * Sb[0][0] + inv_Sw[0][1] * Sb[1][0], inv_Sw[0][0] * Sb[0][1] + inv_Sw[0][1] *
Sb[1][1]],
         [inv_Sw[1][0] * Sb[0][0] + inv_Sw[1][1] * Sb[1][0], inv_Sw[1][0] * Sb[0][1] + inv_Sw[1][1] *
Sb[1][1]]]

    # Calculate the eigenvalues
    a = A[0][0] + A[1][1]
    b = A[0][1] - A[1][0]
    c = A[1][1] - A[0][0]

    D = (a ** 2 + b ** 2 + c ** 2) / 4

    eigenvalue1 = (a + D ** 0.5) / 2
    eigenvalue2 = (a - D ** 0.5) / 2

    return eigenvalue1, eigenvalue2

# Define the function to calculate the eigenvectors
def calculate_eigenvectors(Sw, Sb, eigenvalue1, eigenvalue2):
    # Calculate the determinant of Sw
    det_Sw = Sw[0][0] * Sw[1][1] - Sw[0][1] * Sw[1][0]
    # Calculate the inverse of Sw
    inv_Sw = [[Sw[1][1] / det_Sw -Sw[0][1] / det_Sw], [-Sw[1][0] / det_Sw, Sw[0][0] / det_Sw]]

    A = [[inv_Sw[0][0] * Sb[0][0] + inv_Sw[0][1] * Sb[1][0], inv_Sw[0][0] * Sb[0][1] + inv_Sw[0][1] *
Sb[1][1]],
         [inv_Sw[1][0] * Sb[0][0] + inv_Sw[1][1] * Sb[1][0], inv_Sw[1][0] * Sb
```