Introduction to Machine Learning
HOMEWORK 1

AKSHATA KUMBLE

Q1.

PART A:

The pdf for X is given as a mixture:

$$p(x) = p(x|L=0)P(L=0) + p(x|L=1)P(L=1)$$

where L is the true class label (0 or 1) and the class priors are $P(L=0) = 0.35$ and $P(L=1) = 0.65$

The minimum expected risk classification rule is based on a liklihood ratio test for two classes using gaussian distributions

$$\frac{p(x|L=1)}{p(x|L=0)} \gtrless \gamma$$

where the threshold γ is determined by:

$$\gamma = \frac{P(L=0)}{P(L=1)} \cdot \frac{\lambda_{10}}{\lambda_{01}}$$

$p(x|L=1)$ is the gaussian pdf for class 1 (with mean m_1 & covariance C_1)

$p(x|L=0)$ is the gaussian pdf for class 0 (with mean m_0 & covariance C_0)

γ depends on class priors & loss values

Gaussian pdf is given by:

$$p(x|m, C) = \frac{1}{\sqrt{(2\pi)^d |C|}} e^{(-\frac{1}{2}(x-m)^T C^{-1}(x-m))}$$

Given:

Class priors: $P(L=0) = 0.35$, $P(L=1) = 0.65$

Loss values: Assume symmetric loss $\rightarrow \lambda_{10} = \lambda_{01} = 1$

$\therefore$ Threshold γ becomes:

$$\gamma = \frac{0.35}{0.65} \approx 0.538$$

Implement the classifier

I generated 10,000 samples from the two gaussian distributions for each class:

Class 0 has mean vector $m_0 = [-1, -1, -1, -1]$ and covariance matrix C_0

Class 1 has mean vector $m_1 = [1, 1, 1, 1]$ and covariance matrix C_1.

Sample generation using multivariate gaussian distribution to generate samples for each class.

In [1]:

```python
import numpy as np

# Parameters
m0 = np.array([-1, -1, -1, -1])
C0 = np.array([[2, -0.5, 0.3, 0],
               [-0.5, 1, -0.5, 0],
               [0.3, -0.5, 1, 0],
               [0, 0, 0, 2]])

m1 = np.array([1, 1, 1, 1])
C1 = np.array([[1, 0.3, -0.2, 0],
               [0.3, 2, 0.3, 0],
               [-0.2, 0.3, 1, 0],
               [0, 0, 0, 3]])

N = 10000  # Number of samples

# Class priors
P0 = 0.35
P1 = 0.65

# Generate samples
samples_class0 = np.random.multivariate_normal(m0, C0, int(N * P0))
samples_class1 = np.random.multivariate_normal(m1, C1, int(N * P1))

# Combine samples and labels
samples = np.vstack((samples_class0, samples_class1))
labels = np.hstack((np.zeros(len(samples_class0)), np.ones(len(samples_class1))))

```

Computing likelihood ratio for each sample

In [2]:

```python
from scipy.stats import multivariate_normal

def likelihood_ratio(x, m0, C0, m1, C1):
    p_x_given_L0 = multivariate_normal.pdf(x, mean=m0, cov=C0)
    p_x_given_L1 = multivariate_normal.pdf(x, mean=m1, cov=C1)
    return p_x_given_L1 / p_x_given_L0

```

Varying the threshold γ from 0 to ∞

True positive rate: $P(D = 1 \mid L = 1)$

False positive rate: $P(D = 1 \mid L = 0)$

In [3]:

```python
gammas = np.logspace(-3, 3, 100)  # Vary threshold

tpr = []  # True Positive Rate
fpr = []  # False Positive Rate

for gamma in gammas:
    decisions = np.array([1 if likelihood_ratio(x, m0, C0, m1, C1) > gamma else 0 for x in samples])
    tp = np.sum((decisions == 1) & (labels == 1)) / np.sum(labels == 1)
    fp = np.sum((decisions == 1) & (labels == 0)) / np.sum(labels == 0)
    tpr.append(tp)
    fpr.append(fp)

```

ROC and minimum probability of error

In [4]:

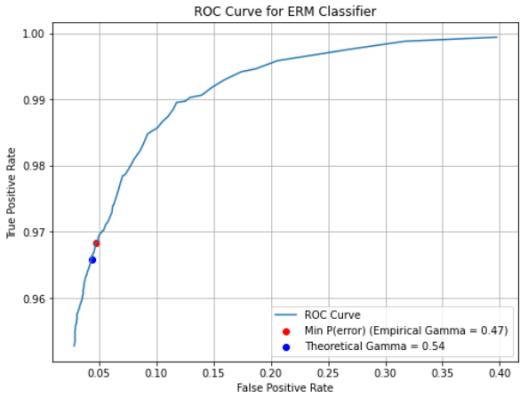
```python
import matplotlib.pyplot as plt

plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for ERM Classifier")
plt.show()


errors = [fp * P0 + (1 - tp) * P1 for tp, fp in zip(tpr, fpr)]
min_error_gamma = gammas[np.argmin(errors)]
min_error = min(errors)

print(f"Minimum Probability of Error: {min_error} at gamma = {min_error_gamma}")
```

The min probability of error occurs when

$$P(\text{error}; \gamma) = P(D=1 \mid L=0; \gamma)P(L=0) + P(D=0 \mid L=1; \gamma)P(L=1)$$

is minimized

```
Theoretical gamma: 0.5384615384615384
```




```
Empirical Minimum Probability of Error: 0.037099999999999994 at gamma = 0.47000000000000003
Error at Theoretical Gamma = 0.5384615384615384: 0.03740000000000003
```

(0.47)

The empirical γ is slightly lower than (0.53) the theoretical γ likely due to the finite sample size and stochastic nature of the generated data. The difference in the empirical min. error & error at theoretical gamma is very small, which suggests that the classifier performs well near the theoretical optimal point.

PART B.

Naive Bayesian Classifier

Assume that features are independent given each class label. That means all the off-diagonal elements in the covariance matrix are set to 0

Diagonal Covariance Matrices:

- For class 0, the covariance matrix C_0 is modified to have only the diagonal elements from the original C_0:

$$C_0^{NB} = \text{diag}(2, 1, 1, 2)$$

- For class 1, covariance matrix C_1 is:

$$C_1^{NB} = \text{diag}(1, 2, 1, 3)$$

Likelihood Computation:

In Naive Bayes setting, the gaussian pdf simplifies since we only consider the diagonal elements of the covariance matrix. The pdfs for both classes $L=0$, $L=1$ are computed as:

$$p(x|L=j) = \prod_{i=1}^{4} \frac{1}{\sqrt{2\pi\sigma_{ji}^2}} \exp\left(-\frac{(x_i - m_{ji})^2}{2\sigma_{ji}^2}\right)$$

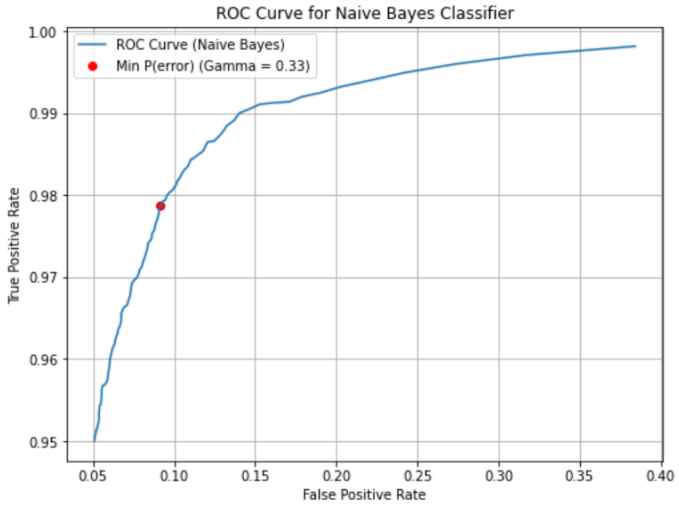
σ_{ji}^2 is variance for i^{th} feature in class $L=j$

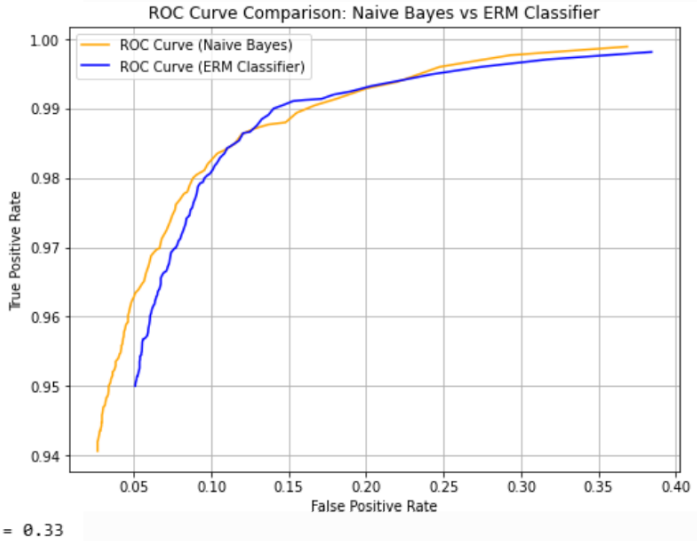
m_{ji} is mean for i^{th} feature in class $L=j$

Naive Bayes produces lower performing ROC curve. It is a less accurate model because it ignores feature correlation. This model mismatch increases the error rate.

In [16]:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

# Parameters (Same means)
m0 = np.array([-1, -1, -1, -1])
m1 = np.array([1, 1, 1, 1])

# Naive Bayes Assumption: Diagonal Covariance Matrices
C0_naive = np.diag(np.diag([[2, -0.5, 0.3, 0],
                            [-0.5, 1, -0.5, 0],
                            [0.3, -0.5, 1, 0],
                            [0, 0, 0, 2]]))

C1_naive = np.diag(np.diag([[1, 0.3, -0.2, 0],
                            [0.3, 2, 0.3, 0],
                            [-0.2, 0.3, 1, 0],
                            [0, 0, 0, 3]]))

# Class priors
P0 = 0.35
P1 = 0.65

# Generate samples (Same as before)
samples_class0 = np.random.multivariate_normal(m0, C0_naive, int(N * P0))
samples_class1 = np.random.multivariate_normal(m1, C1_naive, int(N * P1))

# Combine samples and labels
samples = np.vstack((samples_class0, samples_class1))
labels = np.hstack((np.zeros(len(samples_class0)), np.ones(len(samples_class1))))

def likelihood_ratio_naive(x, m0, C0, m1, C1):
    p_x_given_L0 = multivariate_normal.pdf(x, mean=m0, cov=C0)
    p_x_given_L1 = multivariate_normal.pdf(x, mean=m1, cov=C1)
    return p_x_given_L1 / p_x_given_L0

# Generate finer gamma range around expected values
gammas = np.linspace(0.01, 1, 100)  # Vary gamma finely around expected range

tpr = []  # True Positive Rate
fpr = []  # False Positive Rate

for gamma in gammas:
    decisions = np.array([1 if likelihood_ratio_naive(x, m0, C0_naive, m1, C1_naive) > gamma else 0 for x in samples])
    tp = np.sum((decisions == 1) & (labels == 1)) / np.sum(labels == 1)
    fp = np.sum((decisions == 1) & (labels == 0)) / np.sum(labels == 0)
    tpr.append(tp)
    fpr.append(fp)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label="ROC Curve (Naive Bayes)")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for Naive Bayes Classifier")
plt.grid(True)

# Compute error for each threshold gamma
errors = [fp * P0 + (1 - tp) * P1 for tp, fp in zip(tpr, fpr)]
min_error_idx = np.argmin(errors)
min_error_gamma = gammas[min_error_idx]
min_error = min(errors)

# Mark the minimum probability of error on the ROC curve
plt.scatter(fpr[min_error_idx], tpr[min_error_idx], color='red', label=f'Min P(error) (Gamma = {min_error_gamma:.2f})')
plt.legend()
plt.show()

# Report the minimum probability of error for Naive Bayes
print(f"Naive Bayes Minimum Probability of Error: {min_error} at gamma = {min_error_gamma}")
```






```
Naive Bayes Minimum Probability of Error: 0.04580000000000003 at gamma = 0.33
```

The ROC curve for naive bayes classifier is slightly worse than the true model's ROC curve. It is shifted slightly down and to the right compared to the true model. The model mismatch does negatively impact the ROC curve & the min. prob. error.

The Naive Bayes ROC is slightly inferior to the true models. At any given false +ve rate, the true +ve rate for Naive Bayes is slightly lower than for the true model.

The impact is noticeable but not severe suggesting that while there are correlations between features in the true distribution, they are not extremely strong.

The min. prob. error increased from 3.71% (true model) to 4.58% (Naive Bayes). This is an increase in error of about 23.45% relative to the true model's performance.

The shift in optimal gamma suggests that using the theoretical threshold based on priors might not be optimal for Naive Bayes classifier.

PART C

Fisher Linear Discriminant Analysis

LDA seeks a projection of the data onto a lower dimension space that maximizes the separation between the 2 classes

Using 10,000 samples the mean vectors and covariance matrices are estimated for each class

Estimated mean for class 0:

$$\hat{m}_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} x_i$$

Estimated covariance for class 0:

$$\hat{C}_0 = \frac{1}{N_0 - 1} \sum_{i=1}^{N_0} (x_i - \hat{m}_0)(x_i - \hat{m}_0)^T$$

Similarly for Class 1.

Fisher LDA projection vector w_{LDA} is:

$$w_{LDA} = S_W^{-1}(m_1 - m_0)$$

where $S_W = C_0 + C_1$

Then I project the samples onto the LDA vector: $z = w_{LDA}^T x$

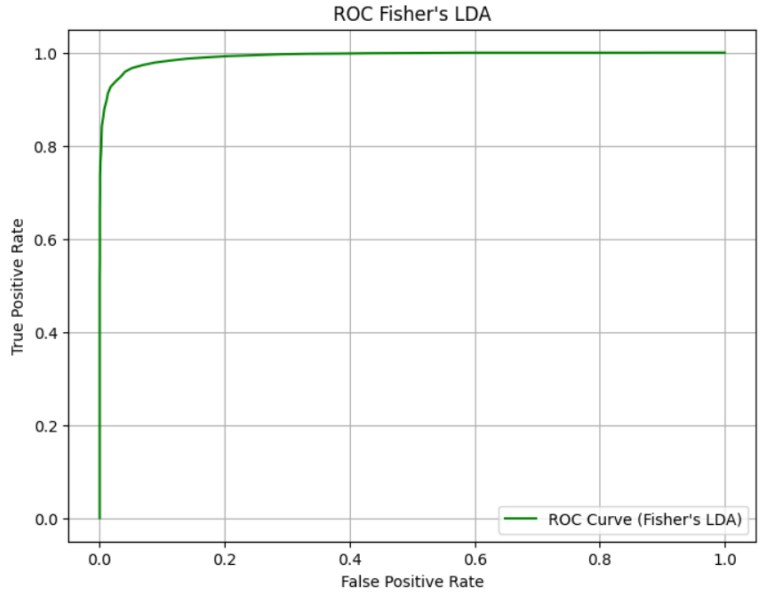
Classify the samples based on whether the projection z is greater than or less than some threshold τ

Then the threshold τ is varied from $-\infty$ to ∞ & the ROC curve is plotted

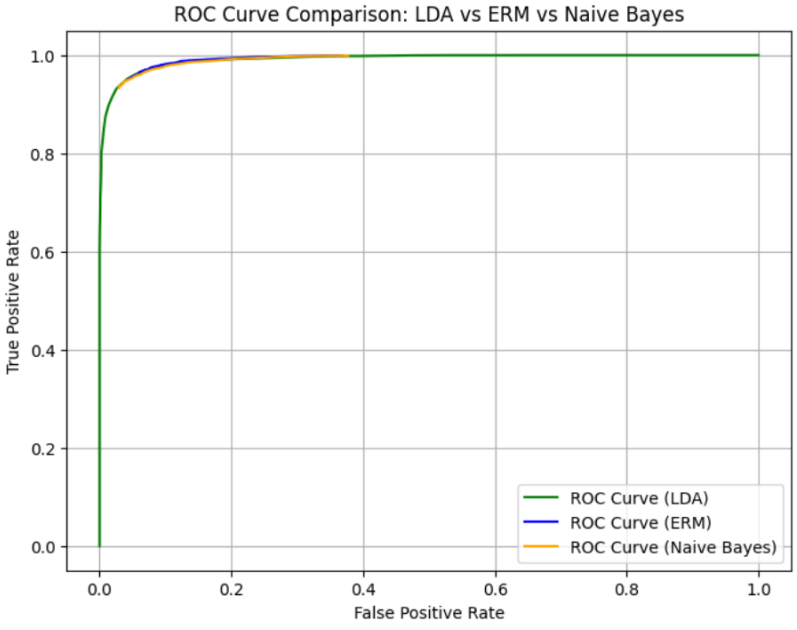
Fisher LDA classifier often performs well when the data is linearly separable & produces better results than Naive Bayes.

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters for the class means and covariance matrices
m0 = np.array([-1, -1, -1, -1])
m1 = np.array([1, 1, 1, 1])

C0 = np.array([[2, -0.5, 0.3, 0],
               [-0.5, 1, -0.5, 0],
               [0.3, -0.5, 1, 0],
               [0, 0, 0, 2]])

C1 = np.array([[1, 0.3, -0.2, 0],
               [0.3, 2, 0.3, 0],
               [-0.2, 0.3, 1, 0],
               [0, 0, 0, 3]])

# Diagonal covariance matrices for Naive Bayes assumption
C0_diag = np.diag(C0)
C1_diag = np.diag(C1)

# Class priors
P0 = 0.35
P1 = 0.65

# Generate samples (from the original Gaussian distributions)
samples_class0 = np.random.multivariate_normal(m0, C0, int(10000 * P0))
samples_class1 = np.random.multivariate_normal(m1, C1, int(10000 * P1))

# Combine samples and labels
samples = np.vstack((samples_class0, samples_class1))
labels = np.hstack((np.zeros(len(samples_class0)), np.ones(len(samples_class1))))

# Define Fisher's LDA projection function
def fisher_lda(m0, C0, m1, C1):
    Sw = C0 + C1  # Within-class scatter matrix
    w_lda = np.linalg.inv(Sw).dot(m1 - m0)  # Fisher LDA vector
    return w_lda

# Get Fisher's LDA projection vector
w_lda = fisher_lda(m0, C0, m1, C1)

# Project the samples onto the LDA axis
projected_data = samples.dot(w_lda)

# Compute ROC curve for Fisher's LDA
gammas_lda = np.linspace(np.min(projected_data), np.max(projected_data), 100)
tpr_lda = []  # True Positive Rate for LDA
fpr_lda = []  # False Positive Rate for LDA

for gamma in gammas_lda:
    decisions_lda = np.where(projected_data > gamma, 1, 0)
    tp = np.sum((decisions_lda == 1) & (labels == 1)) / np.sum(labels == 1)
    fp = np.sum((decisions_lda == 1) & (labels == 0)) / np.sum(labels == 0)
    tpr_lda.append(tp)
    fpr_lda.append(fp)

plt.figure(figsize=(8, 6))
plt.plot(fpr_lda, tpr_lda, label="ROC Curve (Fisher's LDA)", color='green')

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Fisher's LDA")
plt.legend()
plt.grid(True)
plt.show()

print(f"Naive Bayes: {np.interp(fpr_naive, tpr_naive)}")
print(f"Fisher LDA: {np.interp(fpr_lda, tpr_lda)}")
```




```
Fisher's LDA Minimum Probability of Error: 0.03960000000000004 at gamma = 0.43
```




```
Fisher's LDA Minimum Probability of Error: 0.04429999999999997 at gamma = 0.45
Naive Bayes Minimum Probability of Error: 0.04629999999999999 at gamma = 0.34
ERM Classifier Minimum Probability of Error: 0.043600000000000014 at gamma = 0.54
```

All 3 ROC curves are very close to each other, indicating similar performance. However on closer inspection we observe that ERM marginally outperforms LDA & Naive Bayes. The difference is subtle but reflects better class separation.

LDA is known for finding the linear combination of features that best separates the classes.

Naive Bayes assumes that all features are conditionally independent thereby resulting in a slightly higher error rate.

ERM directly minimizes the empirical error rate, fits the data more precisely as it has the lowest minimum prob. error.

The slightly different thresholds indicate how each model adjusts its decision boundary ERM's optimal gamma is higher suggesting a more conservative threshold.

Q2.

Use a mixture of 4 gaussian distributions to generate 10,000 samples

Class priors are set as $P(L=1) = 0.3$, $P(L=2) = 0.3$, $P(L=3) = 0.4$ for classes 1, 2, 3 respectively. Class 1 and Class 2 have their own gaussian distribution with mean μ_1, μ_2 and covariance Σ_1, Σ_2. Class 3 is a mixture of two gaussians with equal weight & means μ_{3a}, μ_{3b} & covariances Σ_{3a}, Σ_{3b}

We need to define the Gaussian parameters such that the means of the gaussians are separated by 2 to 3 times the average standard deviation of the components in order to ensure a moderate amount of overlap between the class conditional distribution which is important for testing the classifier.

eg. of parameters for the three classes:

class 1: $\mu_1 = [0,0,0]$, $\Sigma_1 = \text{diag}(1,1,1)$; class 2: $\mu_2 = [3,3,3]$, $\Sigma_2 = \text{diag}(1,1,1)$

class 3: Mixture of $\mu_{3a} = [0,5,0]$, $\Sigma_{3a} = \text{diag}(1,1,1)$; $\mu_{3b} = [5,0,5]$, $\Sigma_{3b} = \text{diag}(1,1,1)$

We need to:

1. Generate 10,000 samples
2. Specify a decision rule that achieves minimum probability error using 0-1 loss function

The classifier minimizes the probability of error, which classifies a sample a sample by maximizing the posterior probability.

$$\hat{L} = \arg\max_L P(L|x) = \arg\max_L p(x|L) P(L)$$

where: $p(x|L)$ is the class conditional pdf and $P(L)$ are class priors

Computing the posterior probabilities for each class as:

$P(L='|x) = p(x|L-1) P(L-1)$, $P(L=2|x) = p(x|L=2) P(L=2)$, $P(L=3|x) = p(x|L=3) P(L=3)$

Then we classify the sample to the class with the highest posterior probability given the sample

The decision rule for MAP classifier chooses the class with the highest posterior probability.

3. Implement the classifier and classify the samples by Compting the confusion matrix (which is a matrix that shows how many samples were correctly or incorrectly classified for each class).

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal

# Set random seed for reproducibility
np.random.seed(42)

def generate_data(n_samples=10000):
    # Define parameters for the Gaussian distributions
    means = [
        np.array([0, 0, 0]),      # Class 1
        np.array([2.5, 2.5, 2.5]), # Class 2 - first component
        np.array([-2.5, -2.5, -2.5]), # Class 2 - second component
        np.array([2.5, -2.5, 2.5])  # Class 3
    ]

    cov = np.eye(3)  # Standard deviation of 1 in each dimension

    # Generating samples
    X = np.zeros((n_samples, 3))
    y = np.zeros(n_samples)

    class_priors = [0.3, 0.3, 0.4]  # Given in the question

    current_idx = 0
    for class_idx, prior in enumerate(class_priors):
        n_class_samples = int(n_samples * prior)

        if class_idx == 1:  # Class 2 has two components
            n_per_component = n_class_samples // 2
            X[current_idx:current_idx + n_per_component] = np.random.multivariate_normal(
                means[1], cov, n_per_component)
            X[current_idx + n_per_component:current_idx + n_class_samples] = np.random.multivariate_normal(
                means[2], cov, n_class_samples - n_per_component)
        else:
            X[current_idx:current_idx + n_class_samples] = np.random.multivariate_normal(
                means[class_idx if class_idx < 1 else 3], cov, n_class_samples)

        y[current_idx:current_idx + n_class_samples] = class_idx + 1
        current_idx += n_class_samples

    return X, y
```

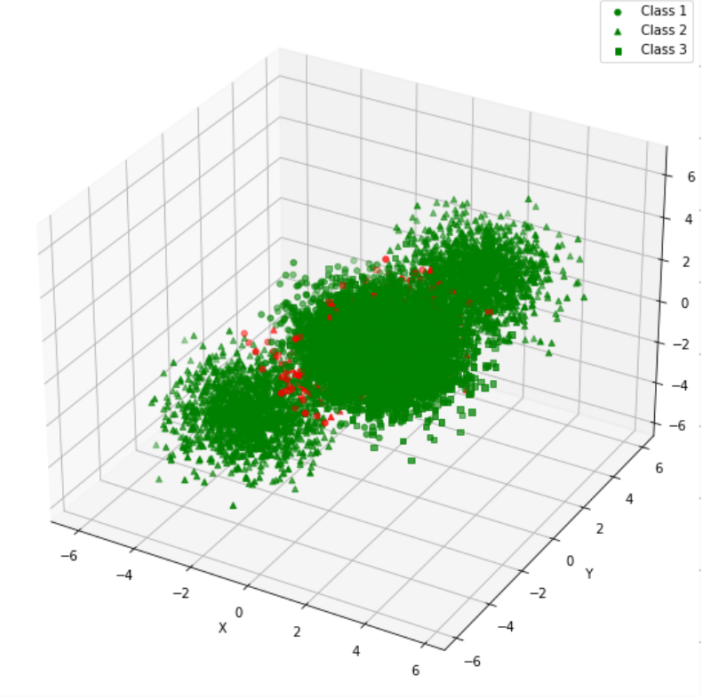
```python
def calculate_class_probabilities(X, means, cov, class_priors):
    probabilities = np.zeros((X.shape[0], 3))

    # Class 1 and 3 have single Gaussian
    probabilities[:, 0] = multivariate_normal.pdf(X, means[0], cov) * class_priors[0]
    probabilities[:, 2] = multivariate_normal.pdf(X, means[3], cov) * class_priors[2]

    # Class 2 has mixture of two Gaussians
    class2_prob = (multivariate_normal.pdf(X, means[1], cov) +
                   multivariate_normal.pdf(X, means[2], cov)) / 2
    probabilities[:, 1] = class2_prob * class_priors[1]

    return probabilities

def bayes_classifier(X, means, cov, class_priors):
    probabilities = calculate_class_probabilities(X, means, cov, class_priors)
    return np.argmax(probabilities, axis=1) + 1

def erm_classifier(X, means, cov, class_priors, loss_matrix):
    probabilities = calculate_class_probabilities(X, means, cov, class_priors)

    # Calculate expected risk for each decision
    risks = np.zeros((X.shape[0], 3))
    for i in range(3):  # For each possible decision
        for j in range(3):  # For each true class
            risks[:, i] += loss_matrix[i, j] * probabilities[:, j]

    return np.argmin(risks, axis=1) + 1

def evaluate_classifier(y_true, y_pred):
    confusion_matrix = np.zeros((3, 3))
    for i in range(len(y_true)):
        confusion_matrix[int(y_pred[i]-1), int(y_true[i]-1)] += 1

    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return confusion_matrix, accuracy
```

```python
def plot_results(X, y_true, y_pred):
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')

    colors = ['g' if pred == true else 'r' for pred, true in zip(y_pred, y_true)]
    markers = {1: 'o', 2: '^', 3: 's'}

    for class_label in [1, 2, 3]:
        mask = y_true == class_label
        ax.scatter(X[mask, 0], X[mask, 1], X[mask, 2],
                   c=[colors[i] for i in range(len(colors)) if y_true[i] == class_label],
                   marker=markers[class_label], label=f'Class {class_label}')

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.legend()
    plt.show()
X, y = generate_data()
# Defining parameters
means = [
    np.array([0, 0, 0]),
    np.array([2.5, 2.5, 2.5]),
    np.array([-2.5, -2.5, -2.5]),
    np.array([2.5, -2.5, 2.5])
]
cov = np.eye(3)
class_priors = [0.3, 0.3, 0.4]

```

```python
# Part A: Bayes Classifier
y_pred_bayes = bayes_classifier(X, means, cov, class_priors)
conf_matrix_bayes, accuracy_bayes = evaluate_classifier(y, y_pred_bayes)

print("Part A Results:")
print(f"Bayes Classifier Accuracy: {accuracy_bayes:.4f}")
print("Confusion Matrix:")
print(conf_matrix_bayes)

# Visualization
plot_results(X, y, y_pred_bayes)

# Part B: ERM Classifier with different loss matrices
loss_matrix_10 = np.array([[0, 10, 10],
                           [1, 0, 10],
                           [1, 1, 0]])

loss_matrix_100 = np.array([[0, 100, 100],
                            [1, 0, 100],
                            [1, 1, 0]])

y_pred_erm_10 = erm_classifier(X, means, cov, class_priors, loss_matrix_10)
y_pred_erm_100 = erm_classifier(X, means, cov, class_priors, loss_matrix_100)

_, accuracy_erm_10 = evaluate_classifier(y, y_pred_erm_10)
_, accuracy_erm_100 = evaluate_classifier(y, y_pred_erm_100)

print("\nPart B Results:")
print(f"ERM Classifier (Loss Matrix 10) Accuracy: {accuracy_erm_10:.4f}")
print(f"ERM Classifier (Loss Matrix 100) Accuracy: {accuracy_erm_100:.4f}")

# Plot results for ERM classifiers
plot_results(X, y, y_pred_erm_10)
plot_results(X, y, y_pred_erm_100)
```

Visualization

```
Part A Results:
Bayes Classifier Accuracy: 0.9748
Confusion Matrix:
[[2887.   58.   63.]
 [  55. 2932.    8.]
 [  58.   10. 3929.]]
```



The decision boundaries appear to be smooth and curved surfaces

PART B.

ERM Classification with Loss Matrices

2 loss matrices Λ_{10} and Λ_{100} are defined to penalize misclassifications more heavily when $L = 3$.

$$\Lambda_{10} = \begin{bmatrix} 0 & 10 & 10 \\ 1 & 0 & 10 \\ 1 & 1 & 0 \end{bmatrix} \; ; \quad \Lambda_{100} = \begin{bmatrix} 0 & 100 & 100 \\ 1 & 0 & 100 \\ 1 & 1 & 0 \end{bmatrix}$$

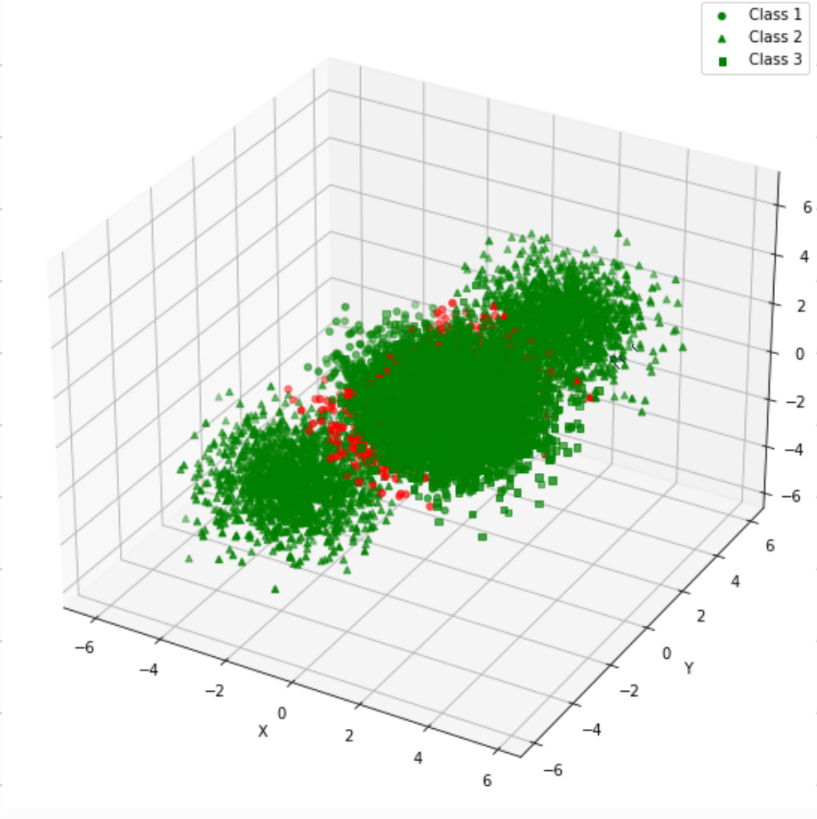
Expected Risk minimization

The classification rule that minimizes the expected risk for each sample:

$$\hat{L}(x) = \arg\min_i \sum_{j=1}^{3} \Lambda(i,j)\, P(L=j \mid x)$$

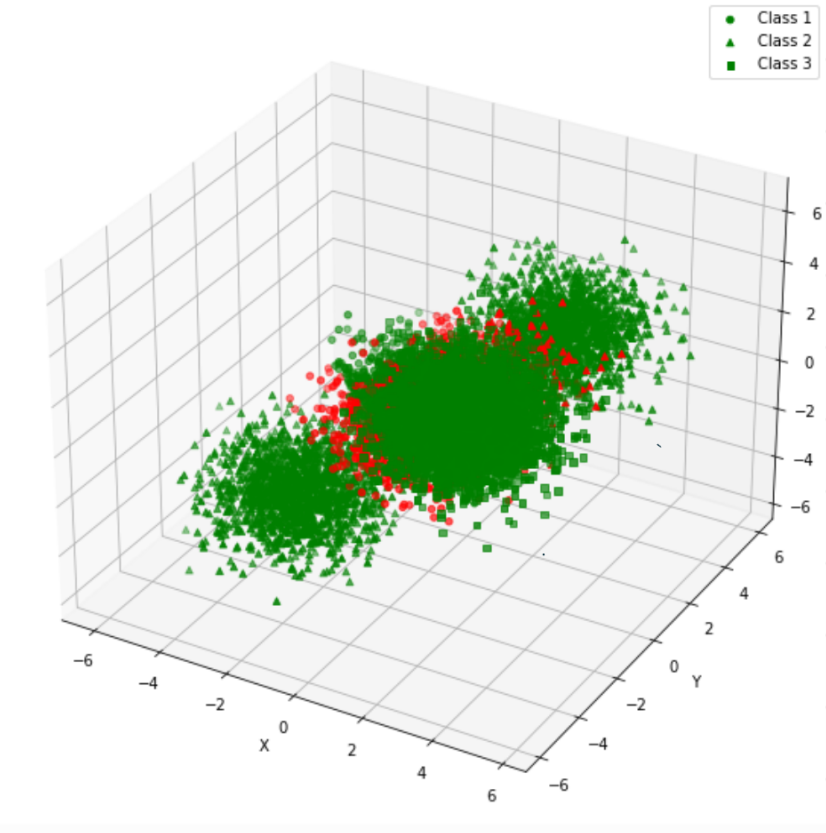
This means that for each class i, we compute the expected risk by summing the posterior probabilities $P(L=j|x)$ weighted by the corresponding loss $\Lambda(i,j)$ and choose the class i that minimizes the total risk.

```
Part B Results:
ERM Classifier (Loss Matrix 10) Accuracy: 0.9565
ERM Classifier (Loss Matrix 100) Accuracy: 0.8883
```



Introduces moderate bias, slightly favoring class 3.

Higher cost for misclassifying class 3.



Dramatic shift in the decision boundary.
Class 1 has reduced significantly
Class 3 dominates a larger portion of the features space.
High penalties for misclassifying class 1. Misclassification into class 3 are less costly.

The Bayes classifier achieves a better overall performance with balanced error rates across classes. The ERM classifier shows a trade off between accuracy and cost sensitivity. With loss matrix 10 there's a minor (1.83%) accuracy sacrifice for moderate cost optimization. With Loss matrix there's a significant accuracy drop suggesting strong avoidance of costly errors.

The progression from Bayes to ERM with increasing loss values demonstrates how the decision boundaries can be manipulated to address specific cost structures in classification problems.

Q3.

To implement minimum probability of error classifiers for the given Wine Quality and Human Activity Recognition datasets we follow these steps:

1. The data is preprocessed → load and clean the data the split the data into features and labels.
2. Estimate mean vectors and covariance matrices for each class. Apply regularization to covariance matrices by adding a small value λ to the diagonal if the covariance matrix is ill-conditioned (i.e. has very small or zero eigenvalues)

$$\Sigma_{regularized} = \Sigma_{sample} + \lambda I$$

3. Estimate Class priors by counting the number of samples per class
4. For each class, assume that the features follow a multivariate gaussian distribution

$$P(x|y) = \frac{1}{\sqrt{(2\pi)^d |\Sigma_y|}} \exp\left(-\frac{1}{2}(x-\mu_y)^T \Sigma_y^{-1} (x-\mu_y)\right)$$

5. Using the gaussian model for each class and the class priors classify the samples based on maximum posterior probability

$$P(y|x) = P(x|y)\,P(y)$$

6. Evaluate the error probability and confusion matrix. and visualize using principal components (to reduce the dimensionality of data)

$$P(\text{error}) = \frac{\text{number of misclassifications}}{\text{total number of samples}}$$

↓

To evaluate the performance of the classifier.

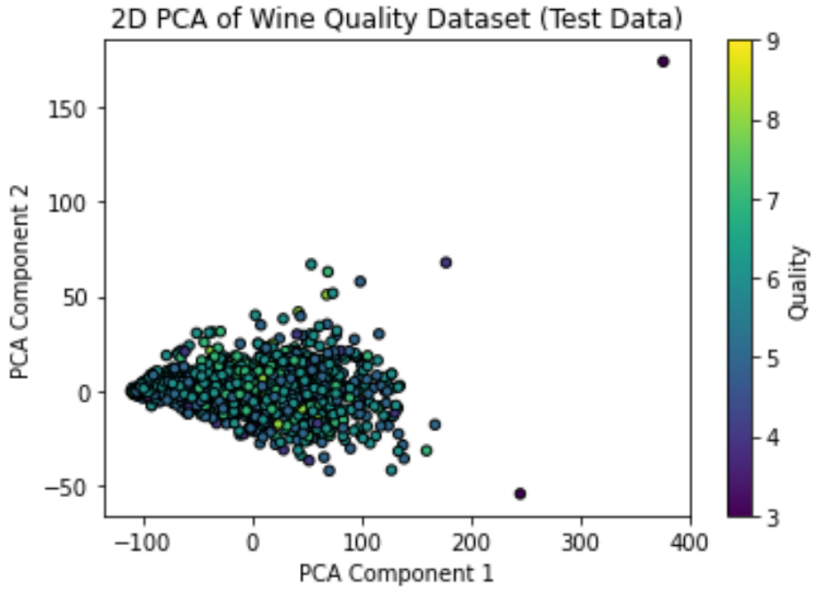
We assume that different classes are independent of eachother, allowing us to estimate parameters separately for each class. The model doesn't assume independence between features with a class as it uses full covariance matrix.

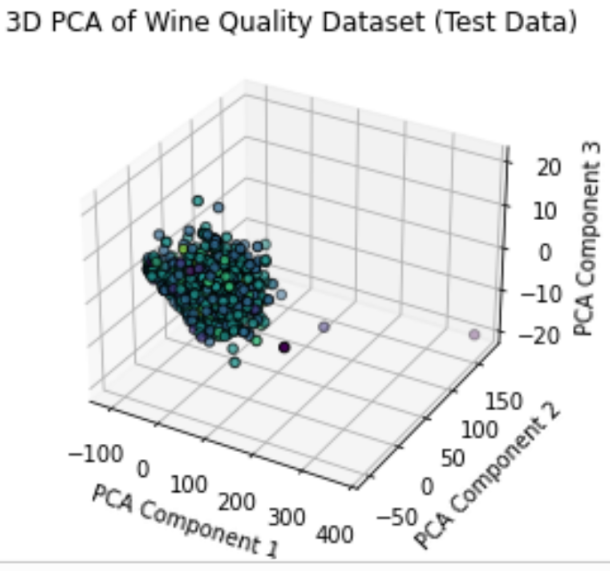
WINE

In [10]:

```python
import numpy as np
import pandas as pd
from scipy.stats import multivariate_normal
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Load the dataset (white wine in this case, but it can be easily changed to red wine)
white_wine_data = pd.read_csv("C:/Users/aksha/Downloads/wine+quality/winequality-red.csv", sep=";")
red_wine_data = pd.read_csv("C:/Users/aksha/Downloads/wine+quality/winequality-white.csv", sep=";")

# Combine the two datasets into one (if needed)
wine_data = pd.concat([white_wine_data, red_wine_data])

# Separate features (X) and labels (y)
X = wine_data.iloc[:, :-1].values  # All columns except the last one (quality)
y = wine_data.iloc[:, -1].values   # The last column (quality)

# Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Estimate the mean vectors and covariance matrices for each class
class_labels = np.unique(y_train)  # Unique class labels (wine quality scores)
means = {}
covariances = {}
priors = {}

# Regularization parameter (based on sample covariance matrix eigenvalues)
def regularize_covariance(cov_matrix, alpha=0.01):
    trace = np.trace(cov_matrix)
    rank = np.linalg.matrix_rank(cov_matrix)
    lambda_reg = alpha * trace / rank
    return cov_matrix + lambda_reg * np.eye(cov_matrix.shape[0])

for label in class_labels:
    X_class = X_train[y_train == label]
    means[label] = np.mean(X_class, axis=0)
    covariances[label] = np.cov(X_class, rowvar=False)

    # Regularize the covariance matrix if necessary
    covariances[label] = regularize_covariance(covariances[label])

    # Estimate class prior probabilities
    priors[label] = len(X_class) / len(X_train)

# Minimum-P(error) classification rule
def classify(sample, means, covariances, priors):
    posteriors = []
    for label in class_labels:
        likelihood = multivariate_normal.pdf(sample, mean=means[label], cov=covariances[label])
        posterior = likelihood * priors[label]
        posteriors.append(posterior)

    return class_labels[np.argmax(posteriors)]

# Apply the classifier to the test data
y_pred = [classify(sample, means, covariances, priors) for sample in X_test]

# Compute the confusion matrix and error probability
conf_matrix = confusion_matrix(y_test, y_pred)
error_probability = 1 - accuracy_score(y_test, y_pred)

print("Confusion Matrix:")
print(conf_matrix)
print(f"Error Probability: {error_probability:.4f}")

# Visualize the dataset using PCA (2D and 3D projections)
def plot_pca(X, y, title, n_components=2):
    pca = PCA(n_components=n_components)
    X_pca = pca.fit_transform(X)

    if n_components == 2:
        plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', edgecolor='k', s=20)
        plt.title(title)
        plt.xlabel('PCA Component 1')
        plt.ylabel('PCA Component 2')
        plt.colorbar(label='Quality')
        plt.show()
    elif n_components == 3:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=y, cmap='viridis', edgecolor='k', s=20)
        ax.set_title(title)
        ax.set_xlabel('PCA Component 1')
        ax.set_ylabel('PCA Component 2')
        ax.set_zlabel('PCA Component 3')
        plt.show()

# Plot the dataset using 2D and 3D PCA projections
plot_pca(X_test, y_test, '2D PCA of Wine Quality Dataset (Test Data)', n_components=2)
plot_pca(X_test, y_test, '3D PCA of Wine Quality Dataset (Test Data)', n_components=3)
```

```
Confusion Matrix:
[[  1   1   0   3   1   0   0]
 [  1   0   2  29  11   0   0]
 [  1   1   6 300  93   1   0]
 [  1   1   3 316 275   1   0]
 [  1   0   1  58 155   0   0]
 [  0   0   0   9  25   2   0]
 [  0   0   0   0   1   0   0]]
Error Probability: 0.6308
```





Interpretation:

- The error probability of 63.08% is quite high indicating that the gaussian classifier is not performing well on this dataset. This suggests that the assumption of gaussian distribution for wine quality classes may not be appropriate.
- From the confusion matrix we see a strong diagonal pattern in some rows indicating better prediction for these classes. However there's significant off-diagonal confusion. For instance row 3 (class 4) has 300 correct predictions but confuses 93 samples with the next class.

Similarly class 5 has 316 correct predictions but confuses 275 samples with the next class. Some classes like the last row have very few samples leading to poor prediction.

From the 2D PCA plot we see significant overlap between different wine quality classes. The data doesn't form clear, separate clusters. There's a gradual color transition, indicating that the wine quality is more of a continuum classes.

The 3D PCA plot shows that even with an additional dimension, classes remain heavily overlapped.

Conclusion:

The gaussian assumption is likely not ideal for this dataset. Wine quality appears to be more of a continuous spectrum rather than distinct classes. The high error rate suggests that using discrete classification might not be the best approach.

We could treat this as a regression problem instead of classification by using ordinal classification techniques that account for the ordered nature of wine quality scores to get better results. The regularization term (λI) could be tuned further for better results.

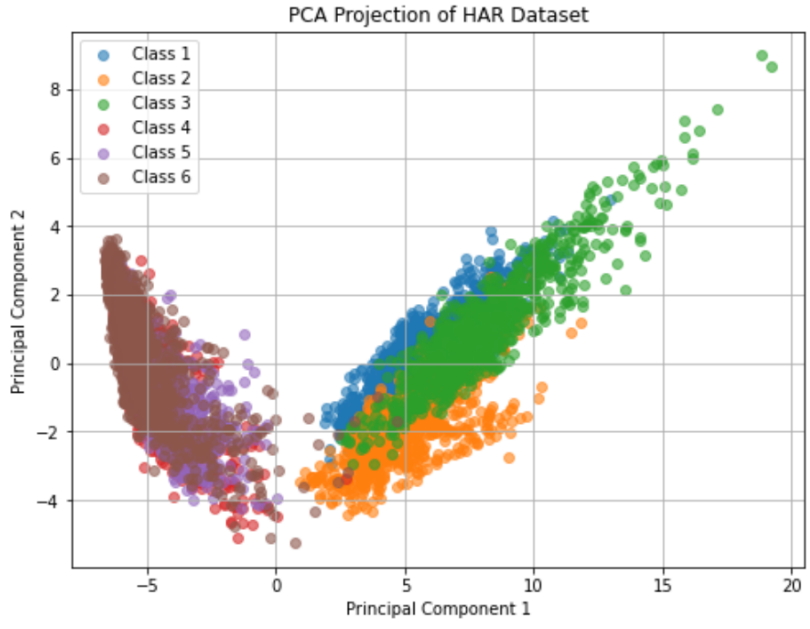
Human Activity Recognition

In [17]:

```python
import pandas as pd
import numpy as np

# Load features and labels
X_train = np.loadtxt('C:/Users/aksha/Downloads/UCI HAR Dataset/train/X_train.txt')
y_train = np.loadtxt('C:/Users/aksha/Downloads/UCI HAR Dataset/train/y_train.txt').astype(int)
X_test = np.loadtxt('C:/Users/aksha/Downloads/UCI HAR Dataset/test/X_test.txt')
y_test = np.loadtxt('C:/Users/aksha/Downloads/UCI HAR Dataset/test/y_test.txt').astype(int)

# Check the shape of the data
print("Training data shape:", X_train.shape)
print("Training labels shape:", y_train.shape)
print("Test data shape:", X_test.shape)
print("Test labels shape:", y_test.shape)


def estimate_mean_cov(X, y, num_classes, regularization_lambda=0.01):
    means = []
    covariances = []
    priors = []

    for c in range(1, num_classes + 1):
        X_c = X[y == c]  # Data belonging to class c
        mean_c = np.mean(X_c, axis=0)
        cov_c = np.cov(X_c, rowvar=False)

        # Apply regularization to the covariance matrix
        cov_c_reg = cov_c + regularization_lambda * np.eye(cov_c.shape[0])

        means.append(mean_c)
        covariances.append(cov_c_reg)
        priors.append(len(X_c) / len(y))  # Prior probability of class c

    return np.array(means), np.array(covariances), np.array(priors)

# Estimate mean vectors, covariance matrices, and priors
num_classes = 6  # There are 6 activity labels
means, covs, priors = estimate_mean_cov(X_train, y_train, num_classes)

# Check shapes
print("Means shape:", means.shape)
print("Covariances shape:", covs.shape)
print("Priors:", priors)

```

```python
from scipy.stats import multivariate_normal

def classify_gaussian(X, means, covs, priors):
    num_samples = X.shape[0]
    num_classes = len(means)
    log_probs = np.zeros((num_samples, num_classes))

    for c in range(num_classes):
        mvn = multivariate_normal(mean=means[c], cov=covs[c])
        log_probs[:, c] = mvn.logpdf(X) + np.log(priors[c])

    return np.argmax(log_probs, axis=1) + 1  # Class with highest posterior probability

# Classify the test data
y_pred = classify_gaussian(X_test, means, covs, priors)


from sklearn.metrics import confusion_matrix, accuracy_score

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Accuracy score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Reduce dimensionality to 2 for visualization
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)

# Plot the projected data
plt.figure(figsize=(8, 6))
for label in np.unique(y_train):
    plt.scatter(X_train_pca[y_train == label, 0], X_train_pca[y_train == label, 1], label=f'Class {label}', alpha=0.6)

plt.title('PCA Projection of HAR Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training data shape: (7352, 561)
Training labels shape: (7352,)
Test data shape: (2947, 561)
Test labels shape: (2947,)
Means shape: (6, 561)
Covariances shape: (6, 561, 561)
Priors: [0.16675734 0.14594668 0.13411317 0.17491839 0.18688792 0.1913765 ]
Confusion Matrix:
 [[482   1  13   0   0   0]
 [  0 470   1   0   0   0]
 [  4  41 375   0   0   0]
 [  0   0   0 387 103   1]
 [  0   0   0   7 525   0]
 [  0   0   0   0   0 537]]
Accuracy: 0.9419748897183576
```



Strong diagonal elements indicate good classification for most classes.

41 samples from Class 3 misclassified as class 2.

103 samples from class 4 misclassified as class 5.

The PCA projection plot shows good separation between some classes. The model achieved an accuracy of approximately 94.2%. This suggests that the gaussian model works well for most classes. The separation in PCA space for some classes aligns with gaussian assumption.

High dimensionality (561 features) may impact covariance estimation. Dimension reduction techniques might be beneficial for computational efficiency & improved classification by focusing on the most deterministic features

The use of λI regularization was crucial for numerical stability. Priors seem relatively balanced [0.167, 0.146, 0.134, 0.175, 0.187, 0.191]

Codes and Results :

https://colab.research.google.com/drive/1u88HJsaOSaJFS3n06uqukpz8fLsYBj5M?usp=sharing