Notebook

October 11, 2024

1 HW1

Author: QIhua Dong

Github: https://github.com/dddraxxx/ee5644

2 Q1

2.0.1 Generate a Gaussian mixture dataset

[2]:
```python
import numpy as np
from scipy.stats import multivariate_normal

# Parameters for class L=0
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]])

# Parameters for class L=1
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]])

# Class priors
P_L0 = 0.35
P_L1 = 0.65

# Number of samples
num_samples = 10000

# Generate class labels L (0 or 1) based on the priors
labels = np.random.choice([0, 1], size=num_samples, p=[P_L0, P_L1])

# Initialize an empty array to store the generated samples
```

```
samples = np.zeros((num_samples, 4))

# Generate samples for each class based on the Gaussian distributions
for i in range(num_samples):
    if labels[i] == 0:
        samples[i] = multivariate_normal.rvs(mean=m0, cov=C0)
    else:
        samples[i] = multivariate_normal.rvs(mean=m1, cov=C1)

# Save the generated samples and labels to a file
np.savez('gaussian_mixture_samples.npz', samples=samples, labels=labels)

print(f"Generated {num_samples} samples and saved to 'gaussian_mixture_samples.
 ↪npz'")
```

```
Generated 10000 samples and saved to 'gaussian_mixture_samples.npz'
```

[4]:
```
samples, labels
```

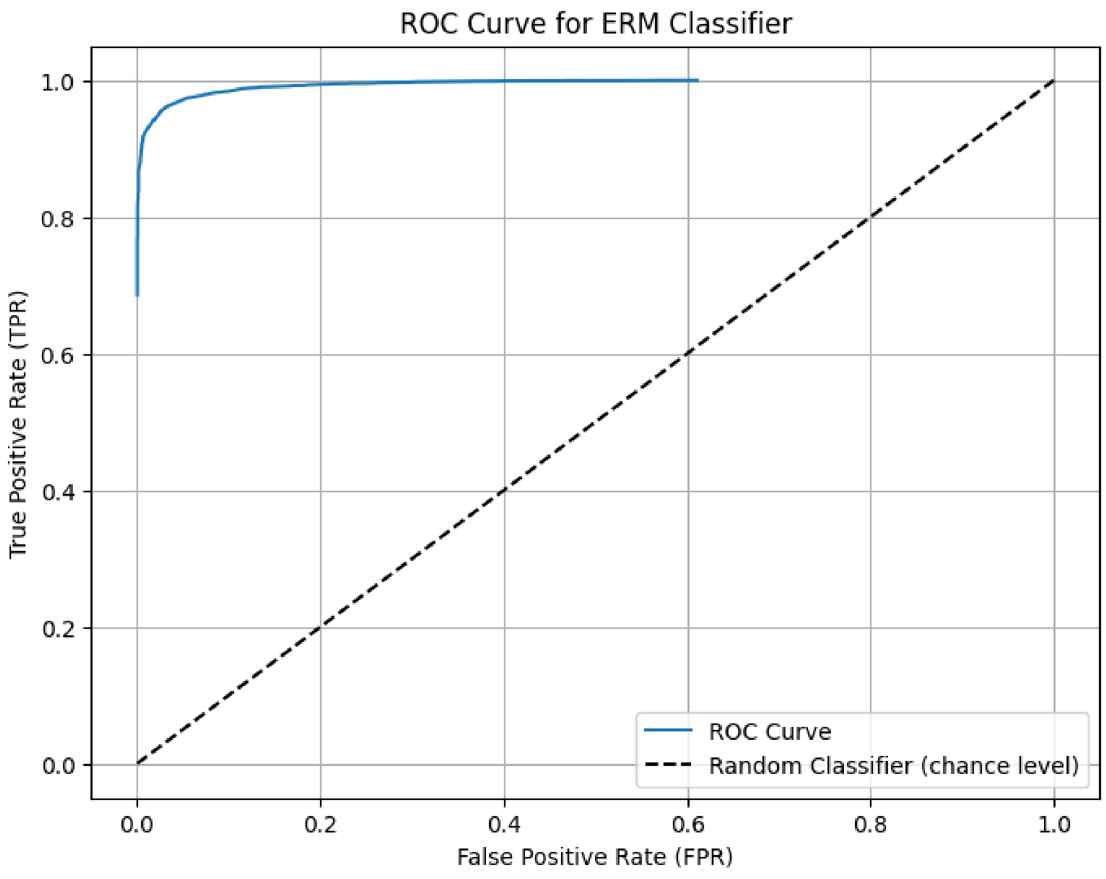
[4]:
```
(array([[ 1.21702054,  0.42134373,  0.10965336,  1.14882005],
        [ 3.86235031,  4.40286043,  2.04268341,  2.08843631],
        [-0.39410192, -0.41088299, -2.98079201, -1.47109445],
        …,
        [-0.69416581, -2.65615592,  1.06310199, -3.30495517],
        [ 0.69585596, -1.63946255,  0.50437631,  2.381229  ],
        [ 0.26764186,  4.44565221,  2.03894145,  0.79276422]]),
 array([1, 1, 0, …, 1, 1, 1]))
```

2.0.2 Part A

[12]:
```
# Import necessary libraries
import numpy as np
from scipy.stats import multivariate_normal

# Load the generated data from part 1
data = np.load('gaussian_mixture_samples.npz')
samples = data['samples']
labels = data['labels']

# Compute the likelihoods p(x|L=0) and p(x|L=1) for each sample
p_x_given_L0 = multivariate_normal.pdf(samples, mean=m0, cov=C0)
p_x_given_L1 = multivariate_normal.pdf(samples, mean=m1, cov=C1)

# Likelihood ratio for each sample
likelihood_ratio = p_x_given_L1 / p_x_given_L0

# Print the answer for step 1
print("Minimum Expected Risk Classification Rule")
print("Likelihood ratio computed as p(x|L=1) / p(x|L=0) for each sample.")
print(f"Sample likelihood ratios: {likelihood_ratio}")  # Print the first 5␣
 ↪likelihood ratios as a sample
```

```
Minimum Expected Risk Classification Rule
Likelihood ratio computed as p(x|L=1) / p(x|L=0) for each sample.
Sample likelihood ratios: [3.51761361e+02 1.89713759e+20 7.80182270e-05 …
2.50332324e-02
 7.99132323e+00 1.64115809e+16]
```

[16]:
```
import numpy as np
import matplotlib.pyplot as plt

# Set up a range of gamma (threshold) values to sweep through
gamma_values = np.logspace(-3, 3, num=500)  # 500 gamma values from 10^-3 to␣
 ↪10^3

# Lists to store true positive and false positive rates for the ROC curve
tpr_values = []  # True Positive Rate (P(D=1 | L=1))
fpr_values = []  # False Positive Rate (P(D=1 | L=0))

# Iterate through each gamma and compute TPR and FPR
for gamma in gamma_values:
    # Apply the likelihood ratio test: decide class based on threshold gamma
    decisions = (likelihood_ratio > gamma).astype(int)

    # True positives: D=1 and L=1
    tp = np.sum((decisions == 1) & (labels == 1))
    fn = np.sum((decisions == 0) & (labels == 1))
    tpr = tp / (tp + fn)  # True positive rate

    # False positives: D=1 and L=0
    fp = np.sum((decisions == 1) & (labels == 0))
    tn = np.sum((decisions == 0) & (labels == 0))
    fpr = fp / (fp + tn)  # False positive rate

    tpr_values.append(tpr)
    fpr_values.append(fpr)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr_values, tpr_values, label='ROC Curve')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.title('ROC Curve for ERM Classifier')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print the answer for step 2
print("Step 2: ROC Curve computed and plotted.")
# print(f"TPR values (sample): {np.array(tpr_values)}")
# print(f"FPR values (sample): {np.array(fpr_values)}")
```

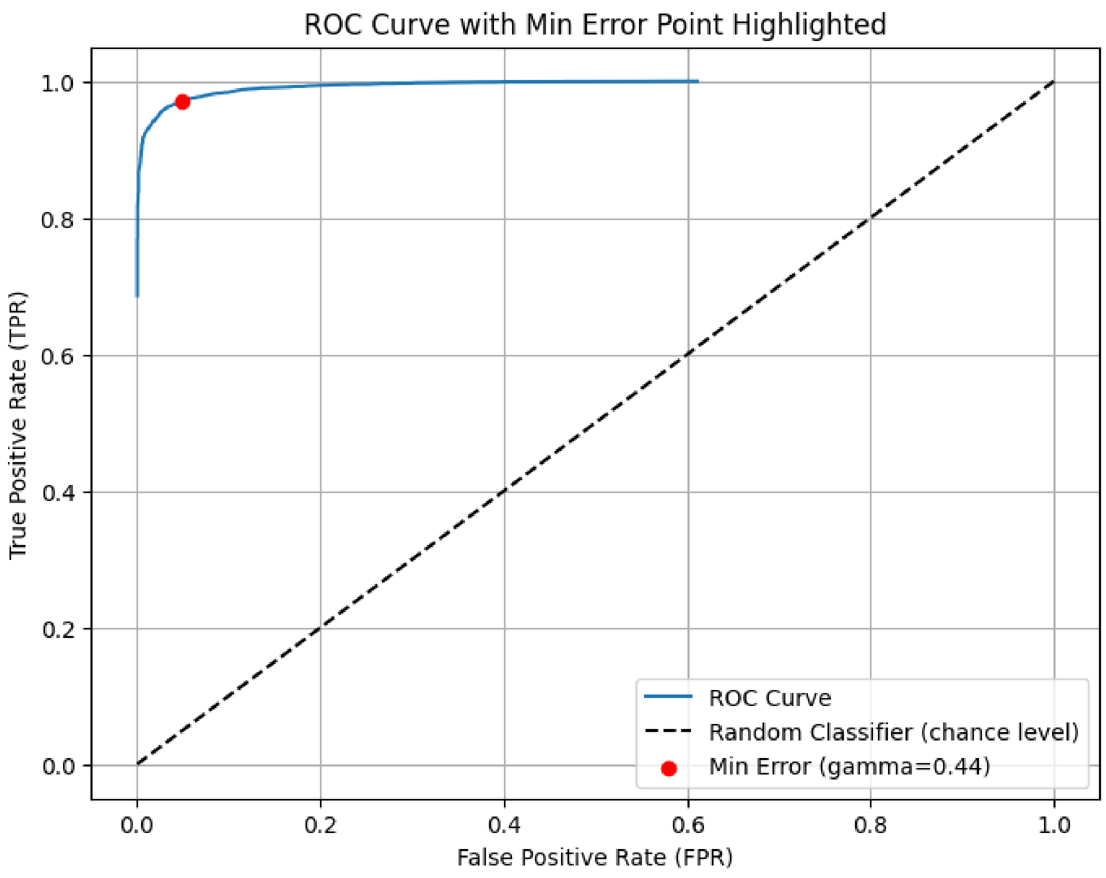


```
Step 2: ROC Curve computed and plotted.
```

[19]:
```
# Step 3: Find the gamma that minimizes the probability of error
# P(error; gamma) = P(D=1 | L=0) * P(L=0) + P(D=0 | L=1) * P(L=1)
P_L0 = 0.35
P_L1 = 0.65

# Initialize an empty list to store the probability of error for each gamma
errors = []

# Calculate the error for each gamma value
```

```
for i, gamma in enumerate(gamma_values):
    # False positive rate and false negative rate (1 - True positive rate)
    fpr = fpr_values[i]
    fnr = 1 - tpr_values[i]

    # Probability of error for this gamma
    p_error = fpr * P_L0 + fnr * P_L1
    errors.append(p_error)

# Find the index of the minimum error
min_error_idx = np.argmin(errors)
min_error_gamma = gamma_values[min_error_idx]
min_error_value = errors[min_error_idx]

# Plot the ROC curve again, but highlight the point of minimum error
plt.figure(figsize=(8, 6))
plt.plot(fpr_values, tpr_values, label='ROC Curve')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.scatter(fpr_values[min_error_idx], tpr_values[min_error_idx], color='red',␣
 ↪label=f'Min Error (gamma={min_error_gamma:.2f})', zorder=5)
plt.title('ROC Curve with Min Error Point Highlighted')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print the answer for step 3
print("Step 3: Minimizing the Probability of Error")
print(f"The minimum probability of error is {min_error_value:.4f} at gamma =␣
 ↪{min_error_gamma:.4f}")
print(f"Corresponding FPR: {fpr_values[min_error_idx]:.4f}, TPR:␣
 ↪{tpr_values[min_error_idx]:.4f}")

# Compute the theoretical optimal gamma
gamma_opt_theoretical = P_L0 / P_L1

# Print the comparison
print("\n")
print("Comparison of Empirically Selected Gamma and Theoretical Optimal Gamma")
print(f"Empirically selected gamma (from ROC curve) = {min_error_gamma:.4f}")
print(f"Theoretically optimal gamma (from priors) = {gamma_opt_theoretical:.
 ↪4f}")

# Compare the results
if np.isclose(min_error_gamma, gamma_opt_theoretical, atol=0.01):
    print("The empirical gamma is very close to the theoretical optimal gamma.")
```

```
else:
    print("The empirical gamma differs from the theoretical optimal gamma.")
```



```
Step 3: Minimizing the Probability of Error
The minimum probability of error is 0.0358 at gamma = 0.4419
Corresponding FPR: 0.0494, TPR: 0.9715


Comparison of Empirically Selected Gamma and Theoretical Optimal Gamma
Empirically selected gamma (from ROC curve) = 0.4419
Theoretically optimal gamma (from priors) = 0.5385
The empirical gamma differs from the theoretical optimal gamma.
```

2.0.3 Part B

[20]:
```
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt

# Load the generated data
```

```
data = np.load('gaussian_mixture_samples.npz')
samples = data['samples']
labels = data['labels']

# True mean vectors from Part A
m0 = np.array([-1, -1, -1, -1])
m1 = np.array([1, 1, 1, 1])

# True covariance matrices from Part A
C0_true = 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_true = np.array([[1, 0.3, -0.2, 0],
                    [0.3, 2, 0.3, 0],
                    [-0.2, 0.3, 1, 0],
                    [0, 0, 0, 3]])

# Naive Bayesian assumption: use diagonal covariance matrices
# Extract the diagonal entries (variances) for each class
C0_naive = np.diag(np.diag(C0_true))  # Diagonal matrix for class L=0
C1_naive = np.diag(np.diag(C1_true))  # Diagonal matrix for class L=1

# Print the diagonal covariance matrices to verify
print("Diagonal covariance matrix for class L=0 (Naive Bayes assumption):")
print(C0_naive)
print("Diagonal covariance matrix for class L=1 (Naive Bayes assumption):")
print(C1_naive)
```

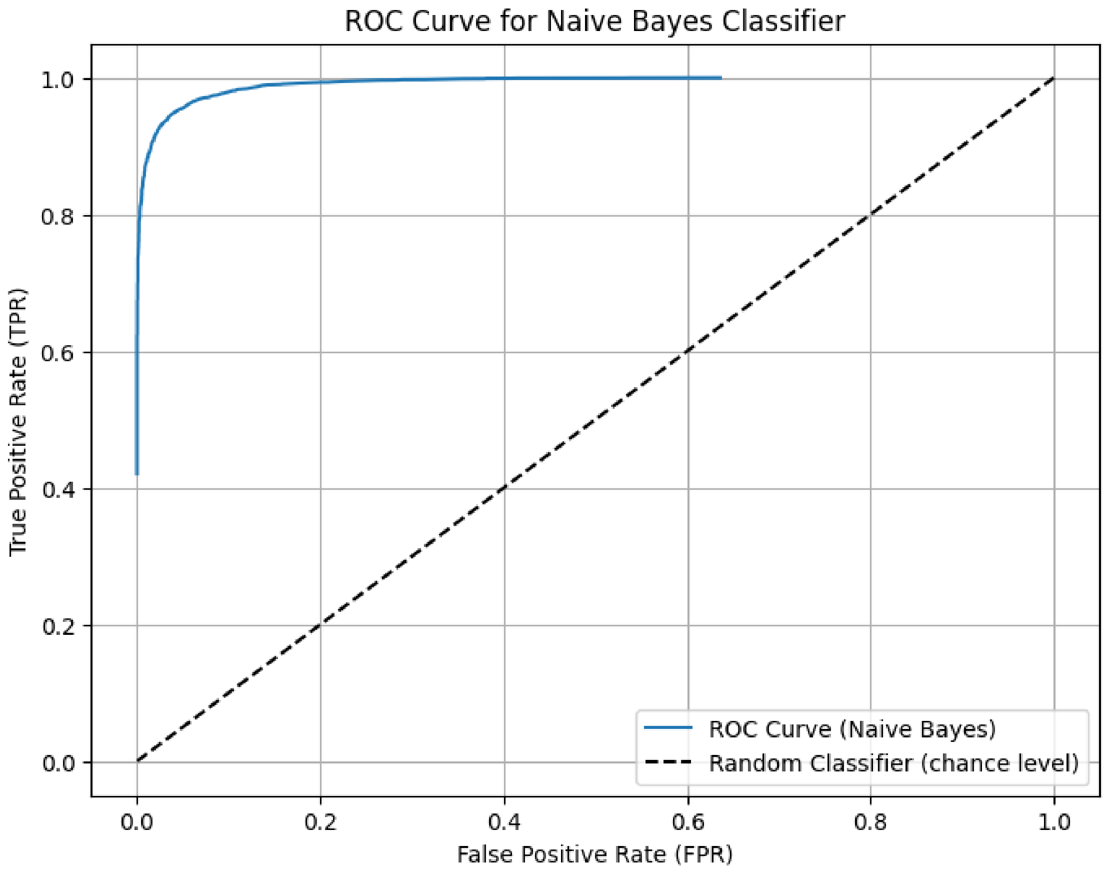
```
Diagonal covariance matrix for class L=0 (Naive Bayes assumption):
[[2. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 2.]]
Diagonal covariance matrix for class L=1 (Naive Bayes assumption):
[[1. 0. 0. 0.]
 [0. 2. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 3.]]
```

[22]:
```
# Class priors
P_L0 = 0.35
P_L1 = 0.65

# Compute the likelihoods p(x|L=0) and p(x|L=1) using the Naive Bayes assumption
p_x_given_L0_naive = multivariate_normal.pdf(samples, mean=m0, cov=C0_naive)
```

```
p_x_given_L1_naive = multivariate_normal.pdf(samples, mean=m1, cov=C1_naive)

# Likelihood ratio for each sample under Naive Bayes assumption
likelihood_ratio_naive = p_x_given_L1_naive / p_x_given_L0_naive

# Print some sample likelihood ratios
print(f"Sample likelihood ratios (Naive Bayes assumption):␣
 ↪{likelihood_ratio_naive}")
```

```
Sample likelihood ratios (Naive Bayes assumption): [2.70824011e+01
3.20203944e+08 2.41014784e-04 … 3.99848265e-02
 1.19619104e+01 1.72700163e+07]
```

[23]:
```
# Set up a range of gamma (threshold) values to sweep through
gamma_values = np.logspace(-3, 3, num=500)  # 500 gamma values from 10^-3 to␣
 ↪10^3

# Lists to store true positive and false positive rates for the ROC curve
tpr_values_naive = []  # True Positive Rate (P(D=1 | L=1))
fpr_values_naive = []  # False Positive Rate (P(D=1 | L=0))

# Iterate through each gamma and compute TPR and FPR using Naive Bayes␣
 ↪assumption
for gamma in gamma_values:
    decisions_naive = (likelihood_ratio_naive > gamma).astype(int)

    # True positives: D=1 and L=1
    tp = np.sum((decisions_naive == 1) & (labels == 1))
    fn = np.sum((decisions_naive == 0) & (labels == 1))
    tpr_naive = tp / (tp + fn)  # True positive rate

    # False positives: D=1 and L=0
    fp = np.sum((decisions_naive == 1) & (labels == 0))
    tn = np.sum((decisions_naive == 0) & (labels == 0))
    fpr_naive = fp / (fp + tn)  # False positive rate

    tpr_values_naive.append(tpr_naive)
    fpr_values_naive.append(fpr_naive)

# Plot the ROC curve for the Naive Bayes assumption
plt.figure(figsize=(8, 6))
plt.plot(fpr_values_naive, tpr_values_naive, label='ROC Curve (Naive Bayes)')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.title('ROC Curve for Naive Bayes Classifier')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print the ROC curve details
print("ROC Curve for Naive Bayes classifier plotted.")
```

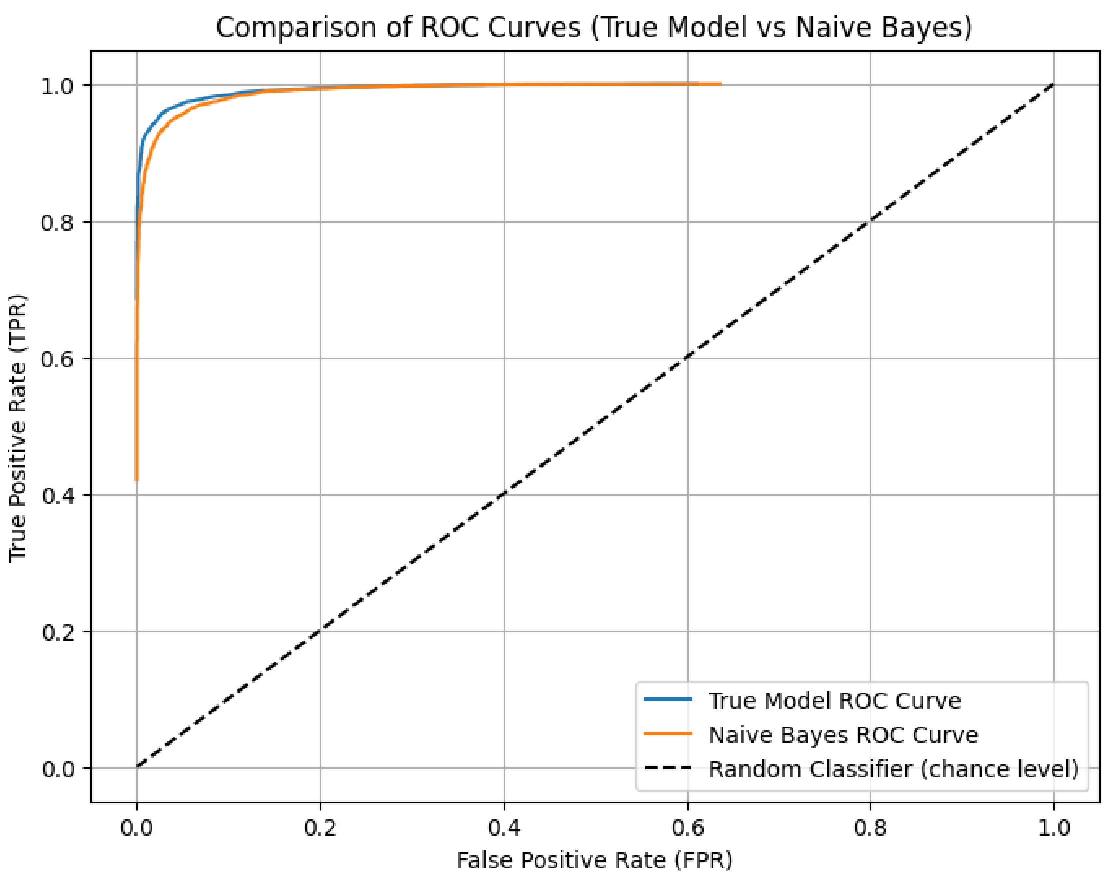


```
ROC Curve for Naive Bayes classifier plotted.
```

[24]:
```
# Step 4: Find the gamma that minimizes the probability of error using Naive␣
  ↪Bayes assumption
errors_naive = []

# Calculate the error for each gamma value using Naive Bayes assumption
for i, gamma in enumerate(gamma_values):
    # False positive rate and false negative rate (1 - True positive rate)
    fpr_naive = fpr_values_naive[i]
    fnr_naive = 1 - tpr_values_naive[i]

    # Probability of error for this gamma
    p_error_naive = fpr_naive * P_L0 + fnr_naive * P_L1
    errors_naive.append(p_error_naive)

# Find the index of the minimum error
min_error_idx_naive = np.argmin(errors_naive)
min_error_gamma_naive = gamma_values[min_error_idx_naive]
min_error_value_naive = errors_naive[min_error_idx_naive]

# Print the results
print("Step 4: Minimizing the Probability of Error with Naive Bayes Assumption")
print(f"The minimum probability of error (Naive Bayes) is␣
 ↪{min_error_value_naive:.4f} at gamma = {min_error_gamma_naive:.4f}")
```

```
Step 4: Minimizing the Probability of Error with Naive Bayes Assumption
The minimum probability of error (Naive Bayes) is 0.0438 at gamma = 0.4298
```

[25]:
```
# Plot ROC Curves for both models to compare
plt.figure(figsize=(8, 6))
plt.plot(fpr_values, tpr_values, label='True Model ROC Curve')
plt.plot(fpr_values_naive, tpr_values_naive, label='Naive Bayes ROC Curve')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.title('Comparison of ROC Curves (True Model vs Naive Bayes)')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print minimum error comparison
print("Comparison of Minimum Probability of Error")
print(f"True Model: Minimum probability of error = {min(errors):.4f} at gamma =␣
 ↪{gamma_values[np.argmin(errors)]:.4f}")
print(f"Naive Bayes Model: Minimum probability of error =␣
 ↪{min_error_value_naive:.4f} at gamma = {min_error_gamma_naive:.4f}")

# Analyze if the error rates differ significantly
error_diff = abs(min(errors) - min_error_value_naive)
print(f"Difference in minimum error between True Model and Naive Bayes:␣
 ↪{error_diff:.4f}")

# Conclusion
if error_diff > 0.01:
    print("The Naive Bayes assumption negatively impacted the performance,␣
 ↪leading to a higher probability of error.")
else:
    print("The Naive Bayes assumption had a minimal impact on the performance␣
 ↪in this case.")
```



```
Comparison of Minimum Probability of Error
True Model: Minimum probability of error = 0.0358 at gamma = 0.4419
Naive Bayes Model: Minimum probability of error = 0.0438 at gamma = 0.4298
Difference in minimum error between True Model and Naive Bayes: 0.0080
The Naive Bayes assumption had a minimal impact on the performance in this case.
```

2.1 Part C

[26]:
```python
import numpy as np

# Load the generated data
data = np.load('gaussian_mixture_samples.npz')
samples = data['samples']
labels = data['labels']

# Separate the samples by class
class_0_samples = samples[labels == 0]
class_1_samples = samples[labels == 1]

# Estimate the means (sample average for each class)
```

```
mean_0 = np.mean(class_0_samples, axis=0)
mean_1 = np.mean(class_1_samples, axis=0)

# Estimate the covariances (sample covariance for each class)
cov_0 = np.cov(class_0_samples, rowvar=False)
cov_1 = np.cov(class_1_samples, rowvar=False)

# Print the estimated means and covariances
print("Estimated mean for class 0:", mean_0)
print("Estimated mean for class 1:", mean_1)
print("Estimated covariance for class 0:\n", cov_0)
print("Estimated covariance for class 1:\n", cov_1)
```

```
Estimated mean for class 0: [-0.97885076 -1.0237032  -0.9936786  -0.98522708]
Estimated mean for class 1: [0.98877577 1.01837441 1.00133738 1.03811884]
Estimated covariance for class 0:
 [[ 2.05628253 -0.50954691  0.30881279  0.02713176]
 [-0.50954691  0.99377434 -0.50324397 -0.0270095 ]
 [ 0.30881279 -0.50324397  1.01671457  0.02585526]
 [ 0.02713176 -0.0270095   0.02585526  1.93776272]]
Estimated covariance for class 1:
 [[ 0.98596894  0.28232123 -0.17598996 -0.00444704]
 [ 0.28232123  2.00259907  0.33323045 -0.02692974]
 [-0.17598996  0.33323045  1.00285683  0.03258915]
 [-0.00444704 -0.02692974  0.03258915  2.96640964]]
```

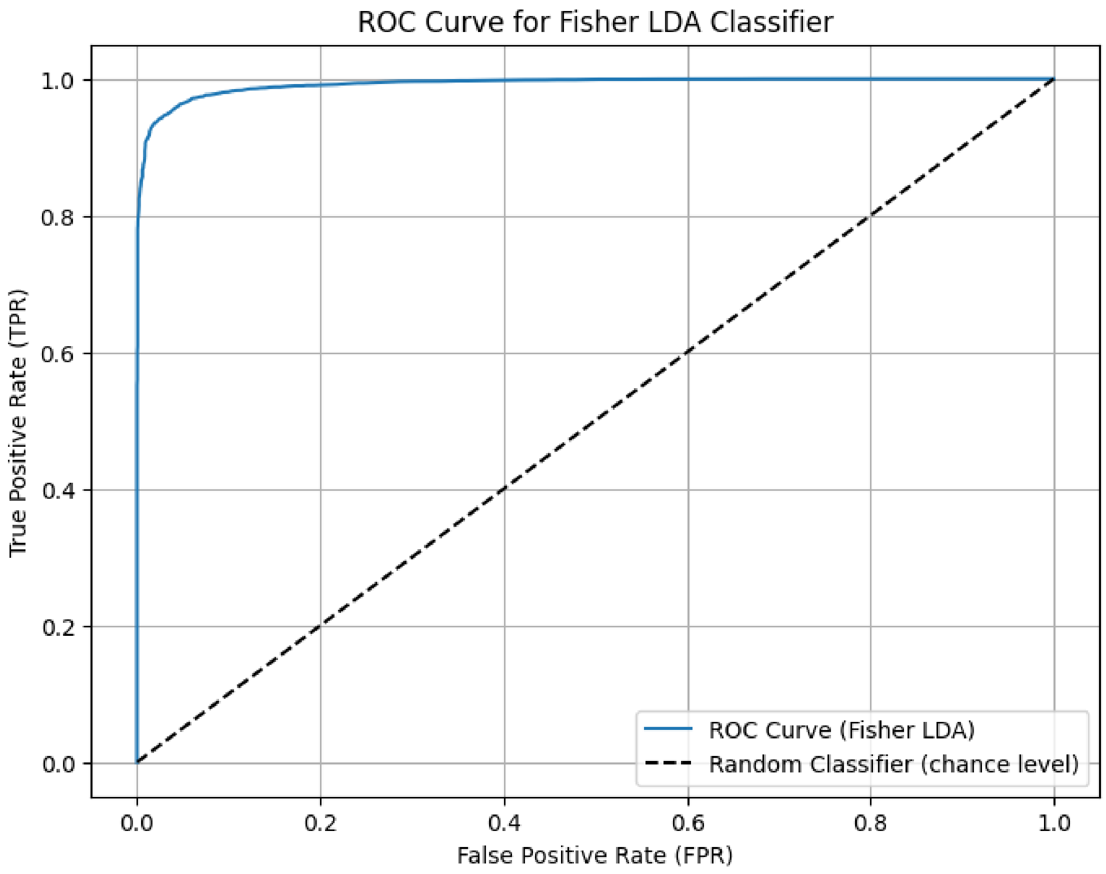
[27]:
```
# Compute the within-class scatter matrix (S_W)
S_W = cov_0 + cov_1

# Compute the between-class difference in means
mean_diff = mean_1 - mean_0

# Compute the Fisher LDA weight vector
w_LDA = np.linalg.inv(S_W).dot(mean_diff)

# Print the Fisher LDA weight vector
print("Fisher LDA projection vector (w_LDA):", w_LDA)
```

```
Fisher LDA projection vector (w_LDA): [0.65951191 0.79556628 0.99968001
0.40636247]
```

[29]:
```
# Project all the samples onto the LDA direction
projected_data = samples.dot(w_LDA)

# Print a few sample projections
print(f"First 5 projected data points: {projected_data}")
```

```
First 5 projected data points: [ 1.71430203  8.94072525 -4.16443533 …
-2.85120666  0.62648119
6.07376262]
```

[30]:
```
import matplotlib.pyplot as plt

# Set up a range of tau (threshold) values to sweep through
tau_values = np.linspace(np.min(projected_data), np.max(projected_data),␣
  ↪num=500)

# Lists to store true positive and false positive rates for the ROC curve
tpr_values_lda = []  # True Positive Rate (P(D=1 | L=1))
fpr_values_lda = []  # False Positive Rate (P(D=1 | L=0))

# Iterate through each tau and compute TPR and FPR
for tau in tau_values:
    decisions_lda = (projected_data > tau).astype(int)

    # True positives: D=1 and L=1
    tp = np.sum((decisions_lda == 1) & (labels == 1))
    fn = np.sum((decisions_lda == 0) & (labels == 1))
    tpr_lda = tp / (tp + fn)  # True positive rate

    # False positives: D=1 and L=0
    fp = np.sum((decisions_lda == 1) & (labels == 0))
    tn = np.sum((decisions_lda == 0) & (labels == 0))
    fpr_lda = fp / (fp + tn)  # False positive rate

    tpr_values_lda.append(tpr_lda)
    fpr_values_lda.append(fpr_lda)

# Plot the ROC curve for Fisher LDA
plt.figure(figsize=(8, 6))
plt.plot(fpr_values_lda, tpr_values_lda, label='ROC Curve (Fisher LDA)')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.title('ROC Curve for Fisher LDA Classifier')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print the ROC curve details
print("ROC Curve for Fisher LDA classifier plotted.")
```



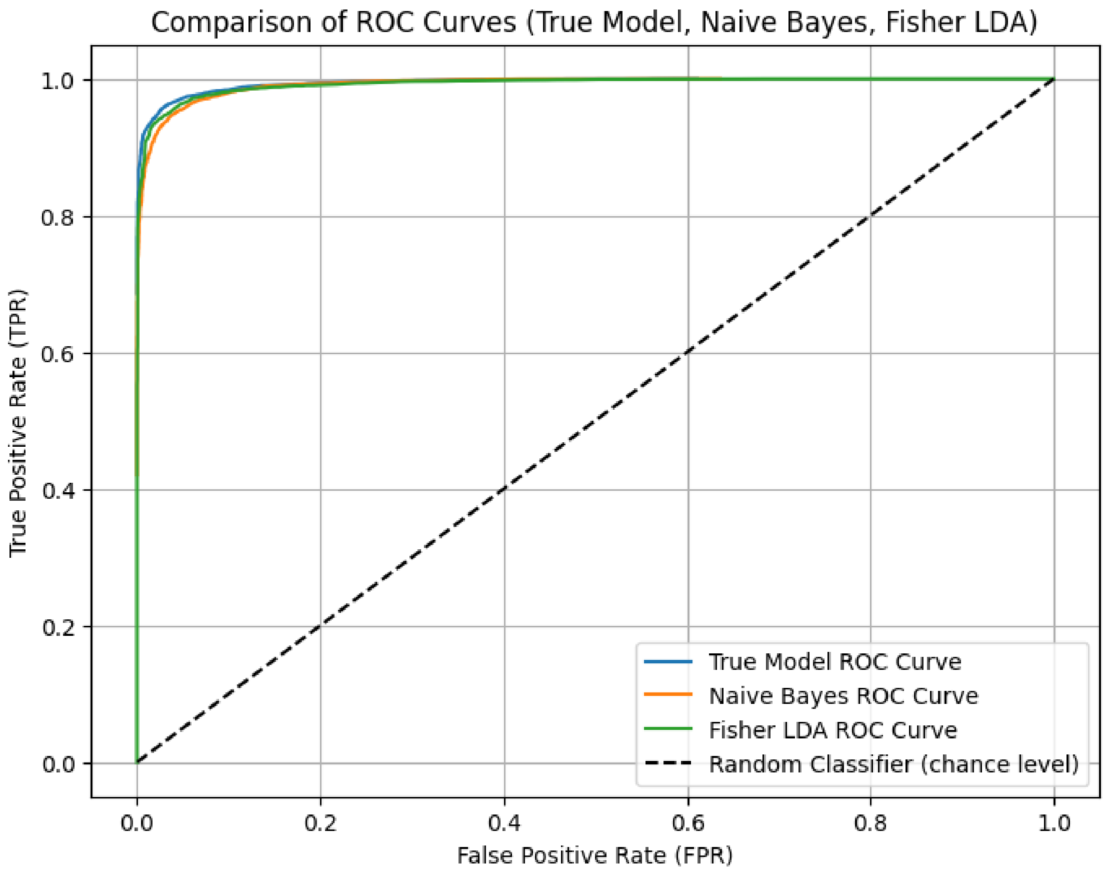
```
ROC Curve for Fisher LDA classifier plotted.
```

[31]:
```
# Class priors
P_L0 = 0.35
P_L1 = 0.65

# Initialize an empty list to store the probability of error for each tau
errors_lda = []

# Calculate the error for each tau value
for i, tau in enumerate(tau_values):
    # False positive rate and false negative rate (1 - True positive rate)
    fpr_lda = fpr_values_lda[i]
    fnr_lda = 1 - tpr_values_lda[i]

    # Probability of error for this tau
    p_error_lda = fpr_lda * P_L0 + fnr_lda * P_L1
    errors_lda.append(p_error_lda)

# Find the index of the minimum error
```

```
min_error_idx_lda = np.argmin(errors_lda)
min_error_tau_lda = tau_values[min_error_idx_lda]
min_error_value_lda = errors_lda[min_error_idx_lda]

# Print the results
print("Step 5: Minimizing the Probability of Error with Fisher LDA")
print(f"The minimum probability of error (LDA) is {min_error_value_lda:.4f} at␣
↪tau = {min_error_tau_lda:.4f}")
```

```
Step 5: Minimizing the Probability of Error with Fisher LDA
The minimum probability of error (LDA) is 0.0397 at tau = -0.7305
```

[33]:
```
# Plot ROC Curves for all three models (True Model, Naive Bayes, Fisher LDA)␣
↪for comparison
plt.figure(figsize=(8, 6))
plt.plot(fpr_values, tpr_values, label='True Model ROC Curve')
plt.plot(fpr_values_naive, tpr_values_naive, label='Naive Bayes ROC Curve')
plt.plot(fpr_values_lda, tpr_values_lda, label='Fisher LDA ROC Curve')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier (chance level)')
plt.title('Comparison of ROC Curves (True Model, Naive Bayes, Fisher LDA)')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Print minimum error comparison for all three models
print("Comparison of Minimum Probability of Error")
print(f"True Model: Minimum probability of error = {min(errors):.4f} at gamma =␣
↪{gamma_values[np.argmin(errors)]:.4f}")
print(f"Naive Bayes Model: Minimum probability of error =␣
↪{min_error_value_naive:.4f} at gamma = {min_error_gamma_naive:.4f}")
print(f"Fisher LDA Model: Minimum probability of error = {min_error_value_lda:.
↪4f} at tau = {min_error_tau_lda:.4f}")
```



```
Comparison of Minimum Probability of Error
True Model: Minimum probability of error = 0.0358 at gamma = 0.4419
Naive Bayes Model: Minimum probability of error = 0.0438 at gamma = 0.4298
Fisher LDA Model: Minimum probability of error = 0.0397 at tau = -0.7305
```

2.1.1 LDA Classifier Performance Compared to True Model and Naive Bayes

Summary:

- **True Model**: Optimal performance due to full knowledge of class distributions.
- **Naive Bayes**: Suffers from feature independence assumption.
- **Fisher LDA**: Provides a strong balance between simplicity and performance, offering a near-optimal solution while being computationally efficient.

Notebook

October 11, 2024

1 Q2

1.1 Part A: Minimum Probability of Error Classification (0-1 Loss)

[2]:
```python
## Step 1: Generate 10,000 Samples from the Data Distribution
import numpy as np

# Class priors
P_L1 = 0.3
P_L2 = 0.3
P_L3 = 0.4

# Means and covariances for each class
mean_1 = [0, 0, 0]  # Mean for class 1
mean_2 = [3, 3, 3]  # Mean for class 2
mean_3a = [6, 0, 0]  # First component of class 3
mean_3b = [0, 6, 6]  # Second component of class 3

cov_1 = np.eye(3)  # Covariance for class 1 (identity)
cov_2 = np.eye(3)  # Covariance for class 2 (identity)
cov_3 = np.eye(3)  # Same covariance for both components of class 3

# Number of samples to generate
n_samples = 10000

# Generate samples for each class based on the priors
n_L1 = int(P_L1 * n_samples)
n_L2 = int(P_L2 * n_samples)
n_L3 = n_samples - n_L1 - n_L2

# Generate class 1 samples (single Gaussian)
samples_L1 = np.random.multivariate_normal(mean_1, cov_1, n_L1)

# Generate class 2 samples (single Gaussian)
samples_L2 = np.random.multivariate_normal(mean_2, cov_2, n_L2)

# Generate class 3 samples (from mixture of two Gaussians)
samples_L3a = np.random.multivariate_normal(mean_3a, cov_3, n_L3 // 2)
```

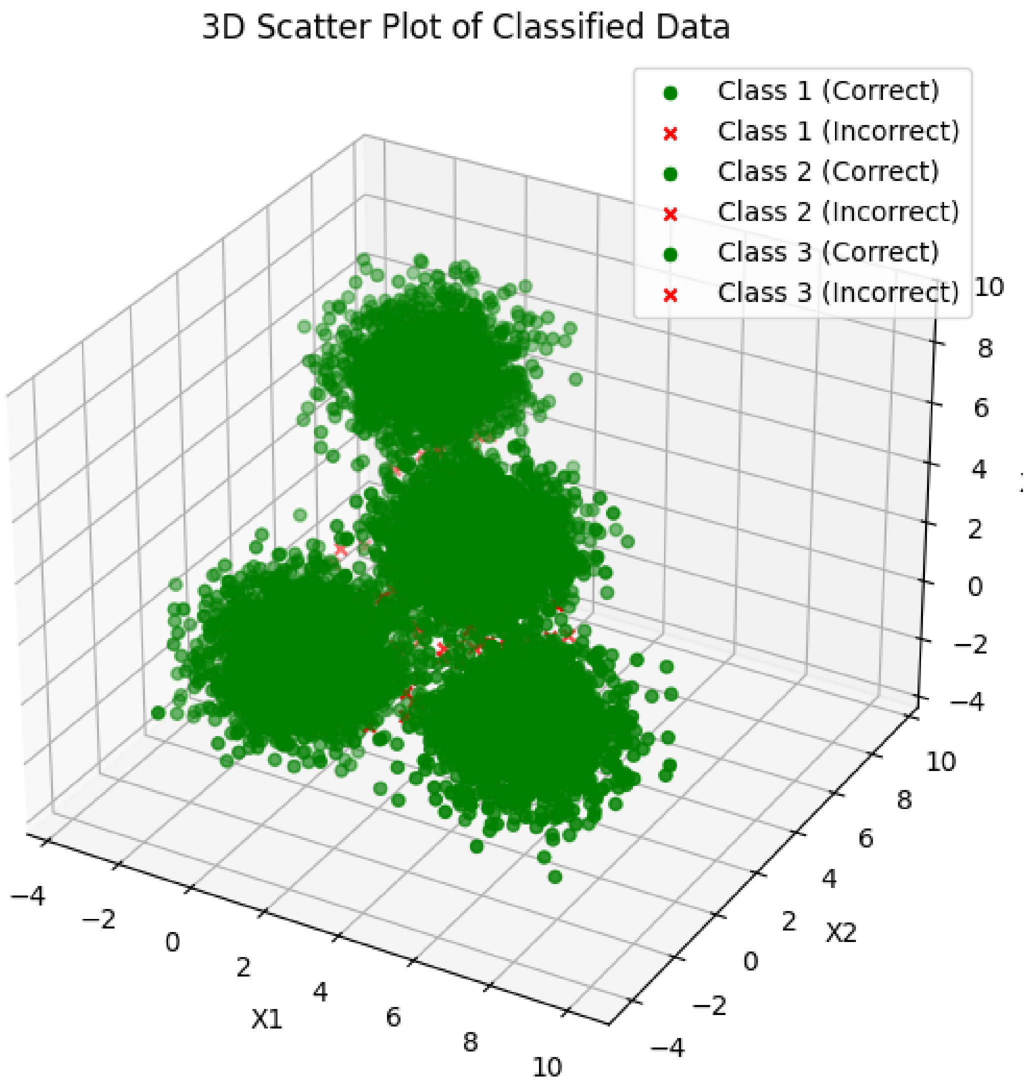
```
samples_L3b = np.random.multivariate_normal(mean_3b, cov_3, n_L3 // 2)
samples_L3 = np.vstack((samples_L3a, samples_L3b))

# Combine samples and create true labels
samples = np.vstack((samples_L1, samples_L2, samples_L3))
labels = np.array([1] * n_L1 + [2] * n_L2 + [3] * n_L3)

# Print shapes to verify
print(f"Generated samples shape: {samples.shape}")
print(f"Generated labels shape: {labels.shape}")
```

```
Generated samples shape: (10000, 3)
Generated labels shape: (10000,)
```

[3]:
```
## Step 2: Implement the Bayesian Decision Rule (Minimum Error)
from scipy.stats import multivariate_normal

# Compute the class-conditional likelihoods for each class
likelihood_L1 = multivariate_normal.pdf(samples, mean=mean_1, cov=cov_1)
likelihood_L2 = multivariate_normal.pdf(samples, mean=mean_2, cov=cov_2)
likelihood_L3a = multivariate_normal.pdf(samples, mean=mean_3a, cov=cov_3)
likelihood_L3b = multivariate_normal.pdf(samples, mean=mean_3b, cov=cov_3)

# Class 3 is a mixture, so we average the likelihoods from the two components
likelihood_L3 = 0.5 * (likelihood_L3a + likelihood_L3b)

# Compute the posterior probabilities using Bayes' Rule
posterior_L1 = likelihood_L1 * P_L1
posterior_L2 = likelihood_L2 * P_L2
posterior_L3 = likelihood_L3 * P_L3

# Combine posteriors into a matrix
posteriors = np.vstack((posterior_L1, posterior_L2, posterior_L3)).T

# Classify based on maximum posterior probability (Bayesian Decision Rule)
predicted_labels = np.argmax(posteriors, axis=1) + 1  # Add 1 to match label␣
 ↪indexing

# Calculate the confusion matrix
confusion_matrix = np.zeros((3, 3), dtype=int)

for true_label, predicted_label in zip(labels, predicted_labels):
    confusion_matrix[true_label - 1, predicted_label - 1] += 1

# Print confusion matrix
print("Confusion Matrix:")
print(confusion_matrix)
```

```
Confusion Matrix:
[[2983   13    4]
 [  17 2960   23]
 [   3   19 3978]]
```

[5]:
```
## Step 3: Visualize the Data in 3D and Indicate Correct/Incorrect␣
↪Classifications

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create a 3D scatter plot of the data
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

# Correctly classified points will be green, incorrect ones will be red
for i in range(1, 4):
    correct_idx = (labels == i) & (predicted_labels == i)
    incorrect_idx = (labels == i) & (predicted_labels != i)

    ax.scatter(samples[correct_idx, 0], samples[correct_idx, 1],␣
↪samples[correct_idx, 2], label=f'Class {i} (Correct)', marker='o',␣
↪color='green', s=20)
    ax.scatter(samples[incorrect_idx, 0], samples[incorrect_idx, 1],␣
↪samples[incorrect_idx, 2], label=f'Class {i} (Incorrect)', marker='x',␣
↪color='red', s=20)

# Labels and title
ax.set_title("3D Scatter Plot of Classified Data")
ax.set_xlabel("X1")
ax.set_ylabel("X2")
ax.set_zlabel("X3")
ax.legend(loc="best")

# Show the plot
plt.show()
```



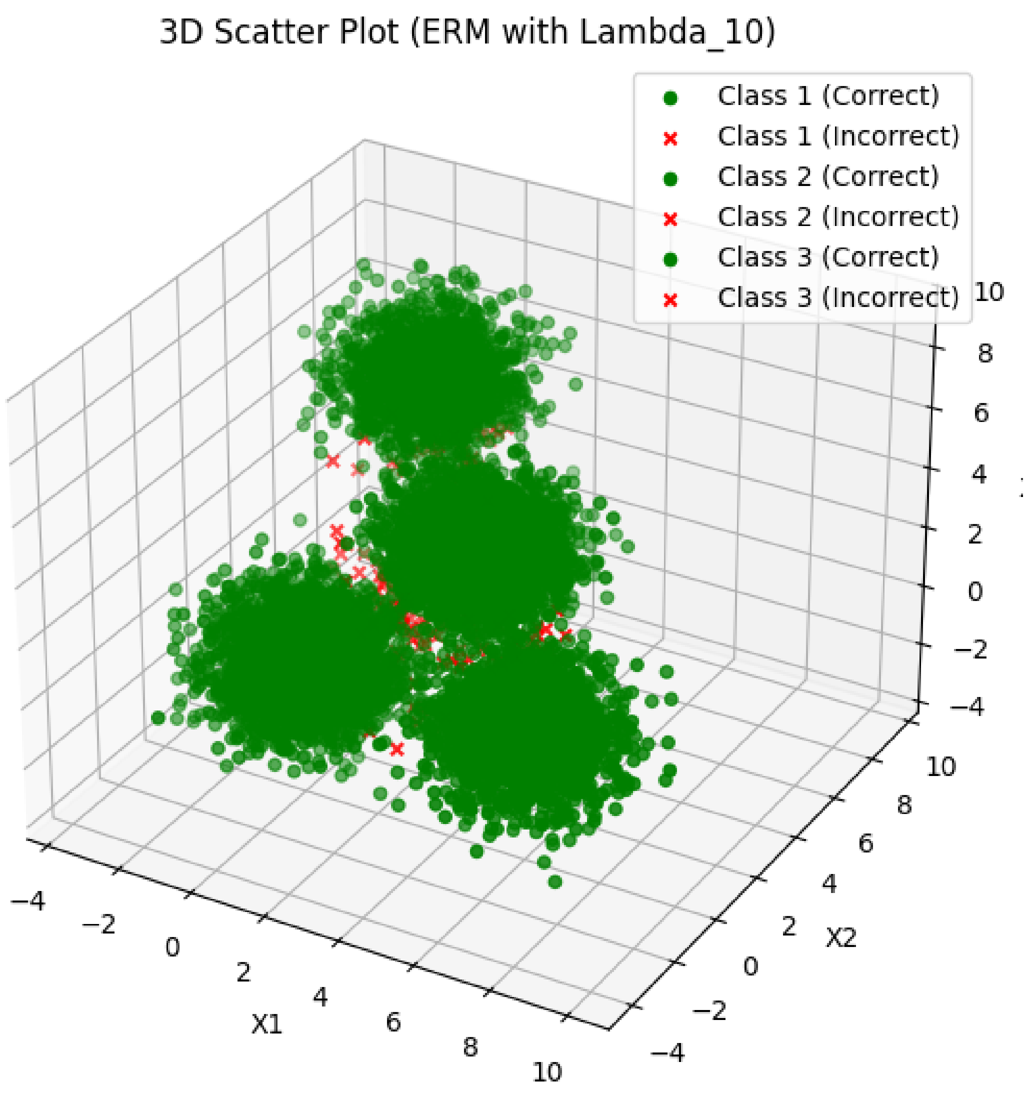
1.2 Part B: Expected Risk Minimization (ERM) with Different Loss Matrices

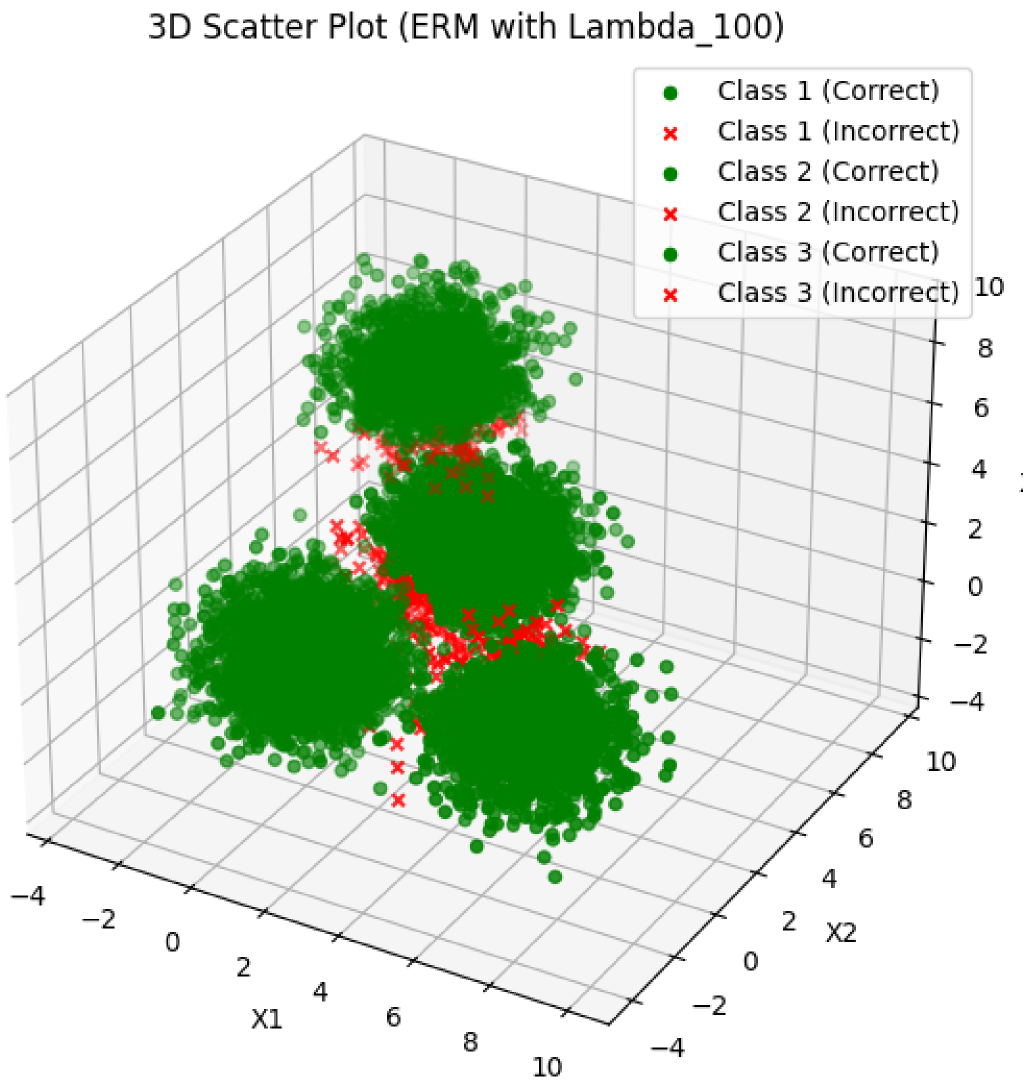
[6]:
```python
# Loss matrices
Lambda_10 = np.array([[0, 10, 10],
                      [1, 0, 10],
                      [1, 1, 0]])

Lambda_100 = np.array([[0, 100, 100],
                       [1, 0, 100],
                       [1, 1, 0]])

# Function to compute expected risk
def expected_risk(posterior_probs, loss_matrix):
    return posterior_probs.dot(loss_matrix)

# Perform classification with Lambda_10
risk_10 = expected_risk(posteriors, Lambda_10)
predicted_labels_10 = np.argmin(risk_10, axis=1) + 1

# Perform classification with Lambda_100
risk_100 = expected_risk(posteriors, Lambda_100)
predicted_labels_100 = np.argmin(risk_100, axis=1) + 1

# Calculate and print confusion matrices for both
confusion_matrix_10 = np.zeros((3, 3), dtype=int)
confusion_matrix_100 = np.zeros((3, 3), dtype=int)

for true_label, predicted_label in zip(labels, predicted_labels_10):
    confusion_matrix_10[true_label - 1, predicted_label - 1] += 1

for true_label, predicted_label in zip(labels, predicted_labels_100):
    confusion_matrix_100[true_label - 1, predicted_label - 1] += 1

print("Confusion Matrix (Lambda_10):")
print(confusion_matrix_10)

print("Confusion Matrix (Lambda_100):")
print(confusion_matrix_100)
```

```
Confusion Matrix (Lambda_10):
[[2996    4    0]
 [  58 2935    7]
 [   9   61 3930]]
Confusion Matrix (Lambda_100):
[[3000    0    0]
 [ 148 2850    2]
 [  43  167 3790]]
```

[7]:
```
# Function to create the 3D scatter plot
def plot_3d_classification(samples, labels, predicted_labels, title):
    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')

    for i in range(1, 4):
        correct_idx = (labels == i) & (predicted_labels == i)
        incorrect_idx = (labels == i) & (predicted_labels != i)

        ax.scatter(samples[correct_idx, 0], samples[correct_idx, 1],␣
 ↪samples[correct_idx, 2], label=f'Class {i} (Correct)', marker='o',␣
 ↪color='green', s=20)
```

```
        ax.scatter(samples[incorrect_idx, 0], samples[incorrect_idx, 1],␣
 ↪samples[incorrect_idx, 2], label=f'Class {i} (Incorrect)', marker='x',␣
 ↪color='red', s=20)

    ax.set_title(title)
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_zlabel("X3")
    ax.legend(loc="best")
    plt.show()

# Plot for Lambda_10
plot_3d_classification(samples, labels, predicted_labels_10, "3D Scatter Plot␣
 ↪(ERM with Lambda_10)")

# Plot for Lambda_100
plot_3d_classification(samples, labels, predicted_labels_100, "3D Scatter Plot␣
 ↪(ERM with Lambda_100)")
```


3D Scatter Plot (ERM with Lambda_10)
Class 1 (Correct)
Class 1 (Incorrect)
Class 2 (Correct)
Class 2 (Incorrect)
Class 3 (Correct)
Class 3 (Incorrect)
X1
X2
−4
−2
0
2
4
6
8
10



```
[8]: # Compare Confusion Matrices
     print("Comparison of Confusion Matrices:\n")

     # Confusion matrix for Bayesian classification (Part A)
     print("Confusion Matrix (Bayesian Classifier - Part A):")
     print(confusion_matrix)

     # Confusion matrix for ERM with Lambda_10
     print("\nConfusion Matrix (ERM with Lambda_10):")
     print(confusion_matrix_10)

     # Confusion matrix for ERM with Lambda_100
     print("\nConfusion Matrix (ERM with Lambda_100):")
print(confusion_matrix_100)
```

```
Comparison of Confusion Matrices:

Confusion Matrix (Bayesian Classifier - Part A):
[[2983   13    4]
 [  17 2960   23]
 [   3   19 3978]]

Confusion Matrix (ERM with Lambda_10):
[[2996    4    0]
 [  58 2935    7]
 [   9   61 3930]]

Confusion Matrix (ERM with Lambda_100):
[[3000    0    0]
 [ 148 2850    2]
 [  43  167 3790]]
```

1.2.1 Inisght

1.2.2 Summary of Insights:

- **As the penalty for Class 3 errors increases** (from (*{10}) to (* {100})), the model becomes highly accurate in classifying Class 3 but **at the cost of more errors** in Classes 1 and 2.
- **The trade-off** is evident: focusing on reducing errors for one class (Class 3) leads to increased misclassifications for other classes (especially Class 2).

Notebook

October 11, 2024

1 Q3

[1]:
```
! wget -q https://archive.ics.uci.edu/static/public/186/wine+quality.zip
! unzip wine+quality.zip -d wine_quality
! wget -q https://archive.ics.uci.edu/static/public/240/
  ↪human+activity+recognition+using+smartphones.zip
! unzip human+activity+recognition+using+smartphones.zip -d␣
  ↪human_activity_recognition_using_smartphones
```

```
Archive:  wine+quality.zip
  inflating: wine_quality/winequality-red.csv
  inflating: wine_quality/winequality-white.csv
  inflating: wine_quality/winequality.names
Archive:  human+activity+recognition+using+smartphones.zip
 extracting: human_activity_recognition_using_smartphones/UCI HAR Dataset.names
 extracting: human_activity_recognition_using_smartphones/UCI HAR Dataset.zip
```

2 Wine Quality Dataset

2.0.1 Step 1: Load the Wine Quality Dataset

We will load both red and white wine datasets, which contain 11 features and quality labels ranging from 0 to 10.

[12]:
```
import pandas as pd

# Load the Wine Quality data (both red and white wine)
wine_red = pd.read_csv('/mnt/localssd/ee5644/hw1/wine_quality/winequality-red.
  ↪csv', sep=';')
wine_white = pd.read_csv('/mnt/localssd/ee5644/hw1/wine_quality/
  ↪winequality-white.csv', sep=';')

# Display information about the datasets
print("Wine Red Dataset Shape:", wine_red.shape)
print("Wine White Dataset Shape:", wine_white.shape)

# # Display the first few rows of the red wine data
# print("Wine Red Data Sample:\n", wine_red.head())
# # Display the first few rows of the white wine data
# print("Wine White Data Sample:\n", wine_white.head())
```

```
Wine Red Dataset Shape: (1599, 12)
Wine White Dataset Shape: (4898, 12)
```

[15]:
```
import numpy as np
import pandas as pd
from scipy.stats import multivariate_normal
from sklearn.metrics import confusion_matrix, accuracy_score

# Helper function to compute class priors, means, and covariances
def estimate_statistics(X, y):
    classes = np.unique(y)
    class_priors = []
    class_means = []
    class_covariances = []

    for cls in classes:
        X_class = X[y == cls]
        prior = len(X_class) / len(X)
        mean = np.mean(X_class, axis=0)
        covariance = np.cov(X_class, rowvar=False)

        class_priors.append(prior)
        class_means.append(mean)
        class_covariances.append(covariance)

    return np.array(class_priors), np.array(class_means), np.
 ↪array(class_covariances)

# Load the Wine Quality data (red wine for this example)
wine_red = pd.read_csv('/mnt/localssd/ee5644/hw1/wine_quality/winequality-red.
 ↪csv', sep=';')

# Extract features and labels
wine_red_X = wine_red.drop(columns='quality').values
wine_red_y = wine_red['quality'].values

# Debug: Print the unique values and range of the true labels
print("True Labels (Wine Red):", np.unique(wine_red_y))
print(f"True Labels Range: {wine_red_y.min()} to {wine_red_y.max()}")

# Estimate priors, means, and covariances for Wine Red dataset
priors_red, means_red, covs_red = estimate_statistics(wine_red_X, wine_red_y)
```

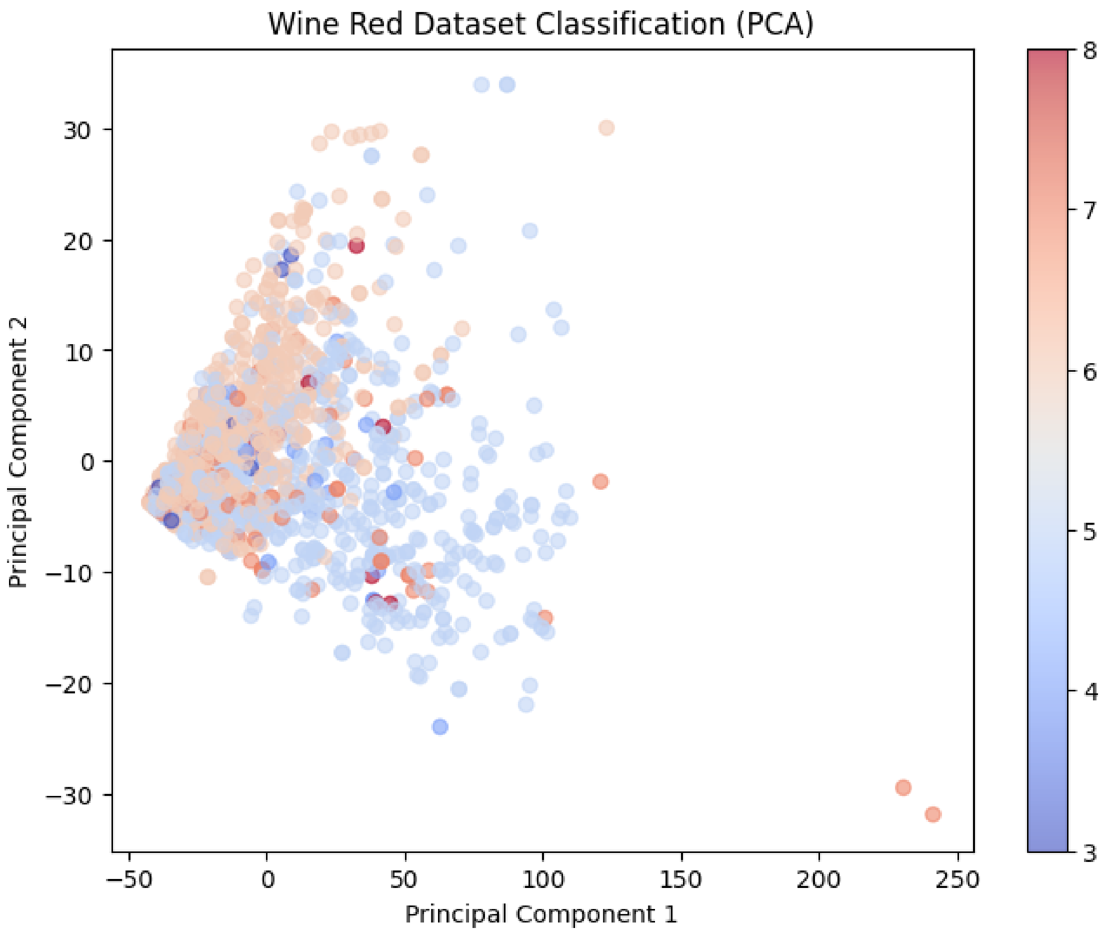
```python
# Regularization function for covariance matrices
def regularize_covariance(cov_matrix, lambda_value):
    I = np.eye(cov_matrix.shape[0])
    regularized_cov = cov_matrix + lambda_value * I
    return regularized_cov

# Apply regularization to the covariance matrices with lambda = 0.01
lambda_reg = 0.01
covs_red_reg = [regularize_covariance(cov, lambda_reg) for cov in covs_red]

# Function to apply minimum-probability-of-error classification
def classify_min_error(X, priors, means, covs):
    n_samples = X.shape[0]
    n_classes = len(priors)

    posteriors = np.zeros((n_samples, n_classes))

    for i in range(n_classes):
        likelihood = multivariate_normal.pdf(X, mean=means[i], cov=covs[i])
        posteriors[:, i] = likelihood * priors[i]

    predicted_labels = np.argmax(posteriors, axis=1)
    return predicted_labels + 3

# Apply minimum-probability-of-error classification
predicted_red = classify_min_error(wine_red_X, priors_red, means_red,␣
 ↪covs_red_reg)


# Debug: Print the unique values and range of the predicted labels after the fix
print("Predicted Labels (Wine Red) After Fix:", np.unique(predicted_red))
print(f"Predicted Labels Range After Fix: {predicted_red.min()} to␣
 ↪{predicted_red.max()}")

# Compute confusion matrix and error probability for Wine Red
conf_matrix_red = confusion_matrix(wine_red_y, predicted_red)
error_prob_red = 1 - accuracy_score(wine_red_y, predicted_red)

# Print confusion matrix and error probability
print("Confusion Matrix (Wine Red):\n", conf_matrix_red)
print("Error Probability (Wine Red):", error_prob_red)
```

```
True Labels (Wine Red): [3 4 5 6 7 8]
True Labels Range: 3 to 8
Predicted Labels (Wine Red) After Fix: [3 4 5 6 7 8]
Predicted Labels Range After Fix: 3 to 8
Confusion Matrix (Wine Red):
```

```
[[  5   0   3   2   0   0]
 [  2   6  28  15   2   0]
 [  5  13 486 165  12   0]
 [  1   6 188 394  45   4]
 [  0   0  11 119  66   3]
 [  0   0   1   7   5   5]]
Error Probability (Wine Red): 0.39837398373983735
```

[16]:

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

# Function to visualize the dataset using PCA
def visualize_pca(X, y, predicted_labels, title):
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=predicted_labels,␣
 ↪cmap='coolwarm', alpha=0.6)
    plt.title(title)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.colorbar(scatter)
    plt.show()

# Visualize the original wine data and the predicted labels (Wine Red)
visualize_pca(wine_red_X, wine_red_y, predicted_red, "Wine Red Dataset␣
 ↪Classification (PCA)")
```



2.0.2 Results and Discussion

1. **Confusion Matrix**:
    - The confusion matrix shows how well the classifier performed for each class.
    - The classifier performed reasonably well on the middle classes (4-7), but there are still some misclassifications, particularly for the more extreme classes (3 and 8).
2. **Error Probability**:
    - The overall error probability is **39.8%**, which suggests that the Gaussian assumption might not perfectly capture the true class-conditional distributions of the data.
3. **Visualization Using PCA**:
    - The PCA visualization shows the separation of different wine quality classes in a 2D projection.
    - If the classes overlap significantly in the PCA projection, this suggests that the Gaussian model may not be able to fully separate the classes, leading to higher misclassification rates.
4. **Suitability of the Gaussian Model**:
    - The assumption that the features follow a Gaussian distribution for each class might not fully hold for the Wine Quality dataset.

    - Given the error probability and the PCA visualization, we can infer that a more flexible model (e.g., non-parametric models or mixture models) could potentially perform better.
5. **Conclusion**:
    - While the Gaussian class-conditional model provides a good baseline, it may not be the best fit for this dataset due to the complex nature of the feature distributions.
    - Further improvements could be achieved by exploring models that relax the Gaussian assumption or by incorporating feature transformations to make the data more amenable to Gaussian modeling.

3 Human Activity Recognition Using Smartphones Dataset

3.0.1 Step 1: Load the Human Activity Recognition (HAR) Dataset

We will load the HAR dataset's training data (`X_train.txt`) and the corresponding labels (`y_train.txt`).

[17]:
```
import pandas as pd
import numpy as np

# Load the training data for HAR dataset
X_train = pd.read_csv('/mnt/localssd/ee5644/hw1/
↪human_activity_recognition_using_smartphones/UCI HAR Dataset/train/X_train.
↪txt', delim_whitespace=True, header=None)
y_train = pd.read_csv('/mnt/localssd/ee5644/hw1/
↪human_activity_recognition_using_smartphones/UCI HAR Dataset/train/y_train.
↪txt', delim_whitespace=True, header=None)

# Convert to numpy arrays
X_train = X_train.values
y_train = y_train.values.flatten()

# Display shape and basic information
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("Unique labels in y_train:", np.unique(y_train))
```

```
/tmp/ipykernel_3182837/2131676244.py:5: FutureWarning: The 'delim_whitespace'
keyword in pd.read_csv is deprecated and will be removed in a future version.
Use ``sep='\s+'`` instead
  X_train = pd.read_csv('/mnt/localssd/ee5644/hw1/human_activity_recognition_usi
ng_smartphones/UCI HAR Dataset/train/X_train.txt', delim_whitespace=True,
header=None)
```

```
X_train shape: (7352, 561)
y_train shape: (7352,)
Unique labels in y_train: [1 2 3 4 5 6]
```

```
/tmp/ipykernel_3182837/2131676244.py:6: FutureWarning: The 'delim_whitespace'
keyword in pd.read_csv is deprecated and will be removed in a future version.
```

```
Use ``sep='\s+'`` instead
  y_train = pd.read_csv('/mnt/localssd/ee5644/hw1/human_activity_recognition_usi
ng_smartphones/UCI HAR Dataset/train/y_train.txt', delim_whitespace=True,
header=None)
```

[19]:
```python
# Helper function to compute class priors, means, and covariances
def estimate_statistics(X, y):
    classes = np.unique(y)
    class_priors = []
    class_means = []
    class_covariances = []

    for cls in classes:
        X_class = X[y == cls]
        prior = len(X_class) / len(X)
        mean = np.mean(X_class, axis=0)
        covariance = np.cov(X_class, rowvar=False)

        class_priors.append(prior)
        class_means.append(mean)
        class_covariances.append(covariance)

    return np.array(class_priors), np.array(class_means), np.
 ↪array(class_covariances)

# Estimate priors, means, and covariances for HAR training dataset
priors_har, means_har, covs_har = estimate_statistics(X_train, y_train)

# Print estimated statistics for HAR
print("Class Priors (HAR):", priors_har)
# print("Mean Vectors (HAR) - Class 1:\n", means_har[0])
print("Covariance Matrix (HAR) - Class 1:\n", covs_har[0])
```

```
Class Priors (HAR): [0.16675734 0.14594668 0.13411317 0.17491839 0.18688792
0.1913765 ]
Covariance Matrix (HAR) - Class 1:
 [[ 2.53540827e-03  1.06151691e-05 -1.73625491e-04 … -4.95972723e-05
   1.83172712e-05 -1.52433104e-04]
 [ 1.06151691e-05  4.35990203e-04  9.60152451e-05 … -2.31419170e-05
   4.37971757e-05 -9.74079500e-05]
 [-1.73625491e-04  9.60152451e-05  1.05206402e-03 …  2.24479930e-06
   2.87508524e-05  7.62211531e-05]
 …
 [-4.95972723e-05 -2.31419170e-05  2.24479930e-06 …  9.72430979e-03
   3.20320771e-03  7.49682841e-03]
 [ 1.83172712e-05  4.37971757e-05  2.87508524e-05 …  3.20320771e-03
   3.19049432e-03  1.59055320e-03]
 [-1.52433104e-04 -9.74079500e-05  7.62211531e-05 …  7.49682841e-03
```

```
  1.59055320e-03  1.30640909e-02]]
```

[20]:
```
# Regularization function for covariance matrices
def regularize_covariance(cov_matrix, lambda_value):
    I = np.eye(cov_matrix.shape[0])
    regularized_cov = cov_matrix + lambda_value * I
    return regularized_cov

# Apply regularization to the covariance matrices with lambda = 0.01
lambda_reg = 0.01
covs_har_reg = [regularize_covariance(cov, lambda_reg) for cov in covs_har]

# Print the regularized covariance matrix for one class
print("Regularized Covariance Matrix for Class 1 (HAR):\n", covs_har_reg[0])
```

```
Regularized Covariance Matrix for Class 1 (HAR):
 [[ 1.25354083e-02  1.06151691e-05 -1.73625491e-04 … -4.95972723e-05
   1.83172712e-05 -1.52433104e-04]
 [ 1.06151691e-05  1.04359902e-02  9.60152451e-05 … -2.31419170e-05
   4.37971757e-05 -9.74079500e-05]
 [-1.73625491e-04  9.60152451e-05  1.10520640e-02 …  2.24479930e-06
   2.87508524e-05  7.62211531e-05]
 …
 [-4.95972723e-05 -2.31419170e-05  2.24479930e-06 …  1.97243098e-02
   3.20320771e-03  7.49682841e-03]
 [ 1.83172712e-05  4.37971757e-05  2.87508524e-05 …  3.20320771e-03
   1.31904943e-02  1.59055320e-03]
 [-1.52433104e-04 -9.74079500e-05  7.62211531e-05 …  7.49682841e-03
   1.59055320e-03  2.30640909e-02]]
```

[21]:
```
from scipy.stats import multivariate_normal

# Function to apply minimum-probability-of-error classification
def classify_min_error(X, priors, means, covs):
    n_samples = X.shape[0]
    n_classes = len(priors)

    posteriors = np.zeros((n_samples, n_classes))

    for i in range(n_classes):
        likelihood = multivariate_normal.pdf(X, mean=means[i], cov=covs[i])
        posteriors[:, i] = likelihood * priors[i]

    predicted_labels = np.argmax(posteriors, axis=1) + 1  # HAR class labels␣
 ↪start from 1
    return predicted_labels

# Apply classification to the HAR training dataset
```

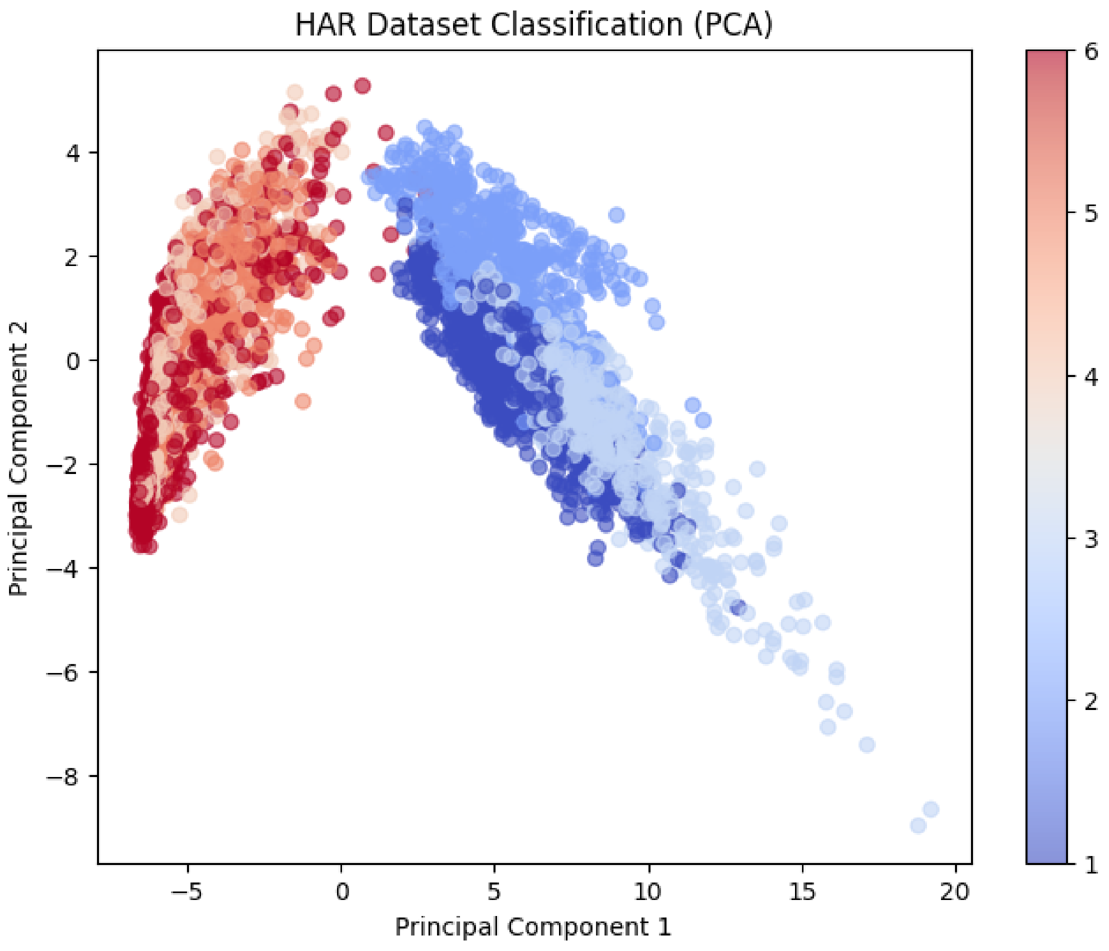
```
predicted_har = classify_min_error(X_train, priors_har, means_har, covs_har_reg)

# Print some predicted labels for HAR
print("Predicted Labels (HAR):", predicted_har[:10])
```

```
Predicted Labels (HAR): [5 5 5 5 5 5 5 5 5 5]
```

[22]:
```
from sklearn.metrics import confusion_matrix, accuracy_score

# Compute confusion matrix and error probability for HAR
conf_matrix_har = confusion_matrix(y_train, predicted_har)
error_prob_har = 1 - accuracy_score(y_train, predicted_har)

# Print confusion matrix and error probability
print("Confusion Matrix (HAR):\n", conf_matrix_har)
print("Error Probability (HAR):", error_prob_har)
```

```
Confusion Matrix (HAR):
 [[1226    0    0    0    0    0]
 [   0 1073    0    0    0    0]
 [   0    1  985    0    0    0]
 [   0    0    0 1197   89    0]
 [   0    0    0    1 1373    0]
 [   0    0    0    0    0 1407]]
Error Probability (HAR): 0.012377584330794389
```

[23]:
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Function to visualize the dataset using PCA
def visualize_pca(X, y, predicted_labels, title):
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=predicted_labels,␣
 ↪cmap='coolwarm', alpha=0.6)
    plt.title(title)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.colorbar(scatter)
    plt.show()

# Visualize the HAR training data using PCA
visualize_pca(X_train, y_train, predicted_har, "HAR Dataset Classification␣
 ↪(PCA)")
```



3.0.2 Discussion of Results for HAR Dataset

1. **Confusion Matrix**:
   - The classifier performs very well on most classes.
   - Most samples are correctly classified, with only slight errors between some activity classes (e.g., Class 3 and 4).
2. **Error Probability**:
   - The overall error probability is **1.23%**, indicating strong performance.
   - This suggests that the Gaussian assumption works well for the HAR dataset.
3. **PCA Visualization**:
   - The PCA plot shows two clear clusters of classes.
   - Some overlap exists between adjacent classes (like Class 3 and 4), which explains minor misclassifications.
4. **Suitability of Gaussian Model**:
   - The Gaussian model appears suitable for this dataset due to its structured features.
   - The low error rate and well-formed PCA projection support this assumption.
5. **Model Assumptions**:
   - The multivariate Gaussian assumption for each class seems valid here.

    - Regularization was necessary due to the high dimensionality (561 features).

6. **Conclusion**:
    - The Gaussian class-conditional model works very well for the HAR dataset.
    - While the performance is strong, further improvements could be explored using non-parametric or more complex models.