NORTHEASTERN UNIVERSITY

Department of Electrical and Computer Engineering

EECE 5644 Machine Learning and Pattern Recognition

Homework Assignment – 1

Mansi Rao Mudrakola
NUID: 002702946

Imports:

```python
# Widget to manipulate plots in Jupyter notebooks
%matplotlib widget
# Import necessary libraries
from sys import float_info  # Threshold smallest positive floating value
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.stats import norm, multivariate_normal
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.decomposition import PCA

# Adjust display settings for readability
np.set_printoptions(suppress=True)

# Set a specific seed for reproducible results
np.random.seed(7)

# Customize font sizes for better visualization
plt.rc('font', size=20)          # controls default text sizes
plt.rc('axes', titlesize=16)     # fontsize of the axes title
plt.rc('axes', labelsize=16)     # fontsize of the x and y labels
plt.rc('xtick', labelsize=14)    # fontsize of the tick labels
plt.rc('ytick', labelsize=14)    # fontsize of the tick labels
plt.rc('legend', fontsize=18)    # legend fontsize
plt.rc('figure', titlesize=20)   # fontsize of the figure title
```

Utility Function:

```python
def generate_data_from_gmm(N, pdf_parameters):
    # Determine dimensionality from mixture PDF parameters
    n = pdf_parameters['mu'].shape[1]
    # Output samples and labels
    X = np.zeros([N, n])
    labels = np.zeros(N)

    # Decide randomly which samples will come from each component u_i ~ Uniform(0, 1) for i = 1, ..., N (or 0, ... , N-1 in code)
    u = np.random.rand(N)
    # Determine the thresholds based on the mixture weights/priors for the GMM, which need to sum up to 1
    thresholds = np.cumsum(pdf_parameters['priors'])
    thresholds = np.insert(thresholds, 0, 0) # For intervals of classes

    L = np.array(range(1, len(pdf_parameters['priors'])+1))
    for l in L:
        # Get randomly sampled indices for this component
        indices = np.argwhere((thresholds[l-1] <= u) & (u <= thresholds[l]))[:, 0]
        # Number of samples in this component
        Nl = len(indices)
        labels[indices] = l * np.ones(Nl) - 1
        # If dealing with a univariate Gaussian, use norm.rvs to sample RVs and not the multivariate version
        if n == 1:
            X[indices, 0] =  norm.rvs(pdf_parameters['mu'][l-1], pdf_parameters['Sigma'][l-1], Nl)
        else:
            X[indices, :] =  multivariate_normal.rvs(pdf_parameters['mu'][l-1], pdf_parameters['Sigma'][l-1], Nl)

    return X, labels
```

Evaluation Functions:

```python
def estimate_roc(discriminant_score, labels):
    N_labels = np.array((sum(labels == 0), sum(labels == 1)))

    # Sorting necessary so the resulting FPR and TPR axes plot threshold probabilities in order as a line
    sorted_score = sorted(discriminant_score)

    # Use gamma values that will account for every possible classification split
    # The epsilon is just to account for the two extremes of the ROC curve (TPR=FPR=0 and TPR=FPR=1)
    gammas = ([sorted_score[0] - float_info.epsilon] +
              sorted_score +
              [sorted_score[-1] + float_info.epsilon])

    # Calculate the decision label for each observation for each gamma
    decisions = [discriminant_score >= g for g in gammas]

    # Retrieve indices where False Positives occur
    ind10 = [np.argwhere((d == 1) & (labels == 0)) for d in decisions]
    # Compute False Positive rates (FPR) as a fraction of total samples in the negative class
    p10 = [len(inds) / N_labels[0] for inds in ind10]
    # Retrieve indices where True Postives occur
    ind11 = [np.argwhere((d == 1) & (labels == 1)) for d in decisions]
    # Compute TP rates (TPR) as a fraction of total samples in the positive class
    p11 = [len(inds) / N_labels[1] for inds in ind11]


    # ROC has FPR on the x-axis and TPR on the y-axis, but return others as well for convenience
    #Create a dictionary to store ROC data
    roc = {}
    roc['p10'] = np.array(p10)
    roc['p11'] = np.array(p11)

    return roc, gammas

def get_binary_classification_metrics(predictions, labels):
    N_labels = np.array((sum(labels == 0), sum(labels == 1)))
```

```python
    # Get indices and probability estimates of the four decision scenarios:
    # (True negative, False positive, False negative, True positive)
    class_metrics = {}

    # Calculate True Negative Probability Rate(TNR) - Correctly predicted negative instances
    ind_00 = np.argwhere((predictions == 0) & (labels == 0))
    class_metrics['TNR'] = len(ind_00) / N_labels[0]

    # Calculate False Positive Probability Rate(FPR) - Incorrectly predicted positive instances
    ind_10 = np.argwhere((predictions == 1) & (labels == 0))
    class_metrics['FPR'] = len(ind_10) / N_labels[0]

    # Calculate False Negative Probability Rate(FNR) - Incorrectly predicted negative instances
    ind_01 = np.argwhere((predictions == 0) & (labels == 1))
    class_metrics['FNR'] = len(ind_01) / N_labels[1]

    # Calculate True Positive Probability Rate(TPR) - Correctly predicted positive instances
    ind_11 = np.argwhere((predictions == 1) & (labels == 1))
    class_metrics['TPR'] = len(ind_11) / N_labels[1]

    return class_metrics
```

Classifiers:

```python
def perform_lda(X, labels, C=2):
    """  Fisher's Linear Discriminant Analysis (LDA) on data from two classes (C=2).

    In practice the mean and covariance matrix parameters would be estimated from training samples.

    Args:
        X: Real-valued matrix of samples with shape [N, n], N for sample count and n for dimensionality.
        labels: Labels for each sample, indicating class membership
        C: Number of classes (default 2)

    Returns:
        w: Fisher's LDA project vector, shape [n, 1].
        z: Scalar LDA projections of input samples, shape [N, 1].
    """

    # First, estimate class-conditional PDF mean and covariance matrices from samples
    mu = np.array([np.mean(X[labels == i], axis=0).reshape(-1, 1) for i in range(C)])
    cov = np.array([np.cov(X[labels == i].T) for i in range(C)])

    # Determine between class and within class scatter matrix
    Sb = (mu[1] - mu[0]).dot((mu[1] - mu[0]).T)
    Sw = cov[0] + cov[1]

    # Regular eigenvector problem for matrix Sw^-1 Sb
    lambdas, U = np.linalg.eig(np.linalg.inv(Sw).dot(Sb))
    # Get the indices from sorting lambdas in order of increasing value, with ::-1 slicing to then reverse order
    idx = lambdas.argsort()[::-1]

    # Extract corresponding sorted eigenvectors
    U = U[:, idx]

    # The first eigenvector corresponds to the maximum eigenvalue, which is the LDA solution weight vector
    w = U[:, 0]

    # Calculate scalar LDA projections in matrix form
    z = X.dot(w)

    return w, z


# ERM classification rule (min prob. of error classifier IF 0-1 loss)
def perform_erm_classification(X, Lambda, gmm_params, C):
    # Conditional likelihoods of each x given each class, shape (C, N)
    class_cond_likelihoods = np.array([multivariate_normal.pdf(X, gmm_params['mu'][i], gmm_params['Sigma'][i]) for i in range(C)])

    # Take diag so we have (C, C) shape of priors with prior prob along diagonal
    class_priors = np.diag(gmm_params['priors'])
    # class_priors*likelihood with diagonal matrix creates a matrix of posterior probabilities
    # with each class as a row and N columns for samples, e.g. row 1: [p(y1)p(x1|y1), ..., p(y1)p(xN|y1)]
    class_posteriors = class_priors.dot(class_cond_likelihoods)

    # Conditional risk matrix of size C x N with each class as a row and N columns for samples
    risk_mat = Lambda.dot(class_posteriors)

    # If 0-1 loss, you could instead compute MAP result: np.argmax(class_posteriors, axis=0)
    return np.argmin(risk_mat, axis=0)
```

Question 1

The probability density function (pdf) for a 4-dimensional real-valued random vector X is as follows: p(x) = p(x|L = 0)P(L = 0) + p(x|L = 1)P(L = 1). Here L is the true class label that indicates which class-label-conditioned pdf generates the data.

The class priors are P(L = 0) = 0.35 and P(L = 1) = 0.65. The class class-conditional pdfs are p(x|L = 0) = g(x|m0,C0) and p(x|L = 1) = g(x|m1,C1), where g(x|m,C) is a multivariate Gaussian probability density function with mean vector m and covariance matrix C. The parameters of the class-conditional Gaussian pdfs are:

$$\mathbf{m}_0 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \quad \mathbf{C}_0 = \begin{bmatrix} 2 & -0.5 & 0.3 & 0 \\ -0.5 & 1 & -0.5 & 0 \\ 0.3 & -0.5 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \quad \mathbf{m}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{C}_1 = \begin{bmatrix} 1 & 0.3 & -0.2 & 0 \\ 0.3 & 2 & 0.3 & 0 \\ -0.2 & 0.3 & 1 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

For numerical results requested below, generate 10000 samples according to this data distribution, keep track of the true class labels for each sample. Save the data and use the same data set in all cases.

Solution:

```
N = 10000

#Define Gaussian Mixture Model (GMM) parameters
gmm_pdf = {}

# Class priors
gmm_pdf['priors'] = np.array([0.35, 0.65])
num_classes = len(gmm_pdf['priors'])
# Mean and covariance of data pdfs conditioned on labels
gmm_pdf['mu'] = np.array([[-1, -1, -1,-1],
                          [1, 1, 1,1]])  # Gaussian distributions means
gmm_pdf['Sigma'] = np.array([[[2, -0.5, 0.3,0],
                           [-0.5, 1, -0.5,0],
                           [0.3, -0.5, 1,0],
                           [0,0,0,2]],
                          [[1, 0.3, -0.2,0],
                           [0.3, 2, 0.3,0],
                           [-0.2, 0.3, 1,0],
                           [0,0,0,3]]])  # Gaussian distributions covariance matrices

# Generate data samples and labels from the GMM

X, labels = generate_data_from_gmm(N, gmm_pdf)

n = X.shape[1]
L = np.array(range(num_classes))

# Count, the number of samples per class
N_per_l = np.array([sum(labels == l) for l in L])
print(N_per_l)
```

```
[3587 6413]
```

```
X
```

```
array([[ 0.09850702,  0.43810787, -0.23463046, -3.29455128],
       [ 2.13489683,  0.72361304, -0.08623332,  1.10974243],
       [-0.12310611,  0.29611341,  1.96808881,  1.2285951 ],
       ...,
       [ 1.1345896 , -0.88725276,  0.04791901, -0.52144539],
       [ 1.41192934,  1.02560274,  0.52314976, -0.16805203],
       [ 2.69850055,  0.23836672,  1.40836536,  1.09087948]])
```

Part A: ERM classification using the knowledge of true data pdf:
1. Specify the minimum expected risk classification rule in the form of a likelihood-ratio test: p(x|L=1) p(x|L=0) ? > γ, where the threshold γ is a function of class priors and fixed (nonnegative) loss values for each of the four cases D = i|L = j where D is the decision label that is either 0 or 1, like L.

Solution:

To begin with this Assignment, I generated 10,000 samples from the given probability density Function given in the question, I used 'Numpy' and 'Pandas' libraries. In order to generate the samples,g I used 'multivariate normal' function from 'scipy' to create a multivariate pdf with two classes viz. L=0, and L=1 for probabilities as P(L=0) = 0.35, and P(L=1) = 0.65

The minimum classification rule in the form of likelihood-ratio test is:

$$p(x|L = 1) = g(x|m1, C1)\ p(x|L = 0) = p(x|m0, C0) > \gamma = \frac{p(x|L = 0)}{p(x|L = 1)} \cdot \frac{\lambda 01 - \lambda 00}{\lambda 10 - \lambda 11}$$

The gamma is a threshold function of non-negative loss values which are fixed and of Class priors for four cases of D=i— L=j. The incorrect results are set to the highest cost possible hence to get correct results we have to go to the lowest cost possible.
We will consider λij = Loss (D = i| L = j), where λij is the loss associated with classifying an observation as label i given that the true label was j. The likelihood ratio test used for classifying an observation x is given by:

$$\frac{p(\mathbf{x}\,|\,L=1)}{p(\mathbf{x}\,|\,L=0)} \overset{\text{Decide 1}}{\underset{\text{Decide 0}}{>}} \gamma \triangleq \frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})}\frac{p(L=0)}{p(L=1)}.$$

Equating the RHS of the equation as γ and taking the log of both sides, leads to the following decision rule:

$$\ln p(\mathbf{x}\,|\,L=1) - \ln p(\mathbf{x}\,|\,L=0) \overset{\text{Decide 1}}{\underset{\text{Decide 0}}{\gtrless}} \ln \gamma.$$

Hence,

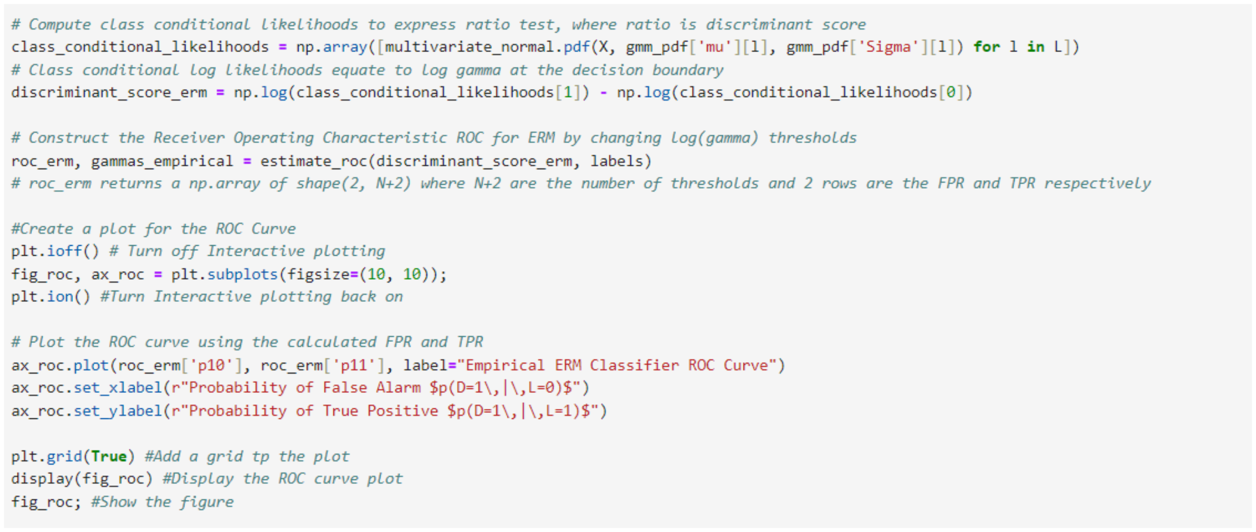
$$\gamma = \frac{0.35}{0.65} * \frac{1-0}{1-0} = 0.538$$

Hence,

$$\frac{p(x|L = 1)}{p(x|L = 0)} > 0.538$$

2. Implement this classifier and apply it on the 10K samples you generated. Vary the threshold γ gradually from 0 to ∞, and for each value of the threshold compute the true positive (detection) probability P(D = 1|L = 1; γ) and the false positive (false alarm) probability P(D = 1|L = 0; γ). Using these paired values, trace/plot an approximation of the ROC curve of the minimum expected risk classifier. Note that at γ = 0 The ROC curve should be at (1 1), and as gamma increases it should traverse towards (0 0). Due to the finite number of samples used to estimate probabilities, your ROC curve approximation should reach this destination value for a finite threshold value. Keep track of (D = 0|L = 1; γ) and P(D = 1|L = 0; γ) values for each gamma value for use in the next section.

Solution:



```
# Compute class conditional likelihoods to express ratio test, where ratio is discriminant score
class_conditional_likelihoods = np.array([multivariate_normal.pdf(X, gmm_pdf['mu'][l], gmm_pdf['Sigma'][l]) for l in L])
# Class conditional log likelihoods equate to log gamma at the decision boundary
discriminant_score_erm = np.log(class_conditional_likelihoods[1]) - np.log(class_conditional_likelihoods[0])

# Construct the Receiver Operating Characteristic ROC for ERM by changing log(gamma) thresholds
roc_erm, gammas_empirical = estimate_roc(discriminant_score_erm, labels)
# roc_erm returns a np.array of shape(2, N+2) where N+2 are the number of thresholds and 2 rows are the FPR and TPR respectively

#Create a plot for the ROC Curve
plt.ioff() # Turn off Interactive plotting
fig_roc, ax_roc = plt.subplots(figsize=(10, 10));
plt.ion() #Turn Interactive plotting back on

# Plot the ROC curve using the calculated FPR and TPR
ax_roc.plot(roc_erm['p10'], roc_erm['p11'], label="Empirical ERM Classifier ROC Curve")
ax_roc.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")

plt.grid(True) #Add a grid tp the plot
display(fig_roc) #Display the ROC curve plot
fig_roc; #Show the figure
```

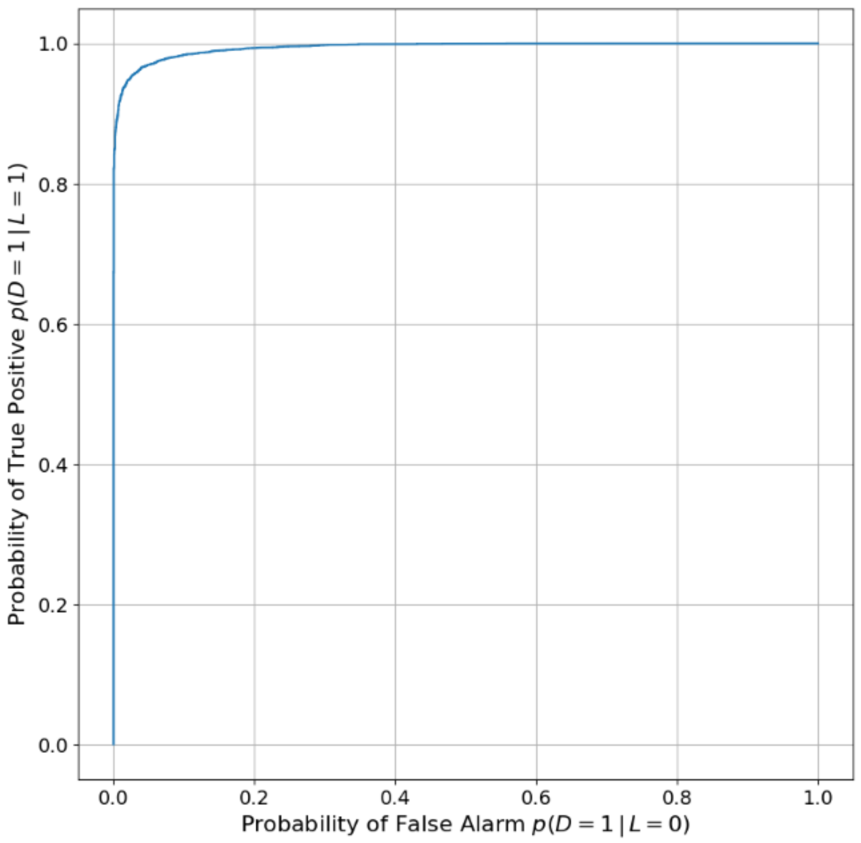


Figure 1: Normal ROC Curve Generated

3. Determine the threshold value that achieves minimum probability of error, and on the ROC curve, superimpose clearly (using a different color/shape marker) the true positive and false positive values attained by this minimum-P(error) classifier. Calculate and report an estimate of the minimum probability of error that is achievable for this data distribution. Note that P(error; γ) = P(D = 1|L = 0; γ)P(L = 0) + P(D = 0|L = 1; γ)P(L = 1). How does your empirically selected γ value that minimizes P(error) compare with the theoretically optimal threshold you compute from priors and loss values?

Solution:
To determine the Minimum Risk, I first constructed the ROC curve. Subsequently, I computed the empirical error probability by taking the dot product of the False Positive and True Positive rates, considering the ratio of labels per sample to the total number of samples. Figure 2 illustrates the ROC curve, highlighting the Minimum Risk, which is depicted as a graph featuring both theoretical and empirical error probabilities.

```python
# ROC returns FPR vs TPR, but prob error needs FNR so take 1-TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
prob_error_empirical = np.array((roc_erm['p10'], 1 - roc_erm['p11'])).T.dot(N_per_l / N)

# Min prob error for the empirically-selected gamma thresholds
min_prob_error_empirical = np.min(prob_error_empirical)
min_ind_empirical = np.argmin(prob_error_empirical)

# Compute theoretical gamma as log-ratio of priors (0-1 loss) -> MAP classification rule
gamma_map = gmm_pdf['priors'][0] / gmm_pdf['priors'][1]
decisions_map = discriminant_score_erm >= np.log(gamma_map)

class_metrics_map = get_binary_classification_metrics(decisions_map, labels)
# To compute probability of error, we need FPR and FNR
min_prob_error_map = np.array((class_metrics_map['FPR'] * gmm_pdf['priors'][0] +
                               class_metrics_map['FNR'] * gmm_pdf['priors'][1]))

# Plot theoretical and empirical
ax_roc.plot(roc_erm['p10'][min_ind_empirical], roc_erm['p11'][min_ind_empirical], 'go', label="Empirical Min Pr(error) ERM",
            markersize=14)
ax_roc.plot(class_metrics_map['FPR'], class_metrics_map['TPR'], 'rx', label="Theoretical Min Pr(error) ERM", markersize=14)
plt.legend()

print("Min Empirical Pr(error) for ERM = {:.3f}".format(min_prob_error_empirical))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas_empirical[min_ind_empirical])))

print("Min Theoretical Pr(error) for ERM = {:.3f}".format(min_prob_error_map))
print("Min Theoretical Gamma = {:.3f}".format(gamma_map))

display(fig_roc)
```

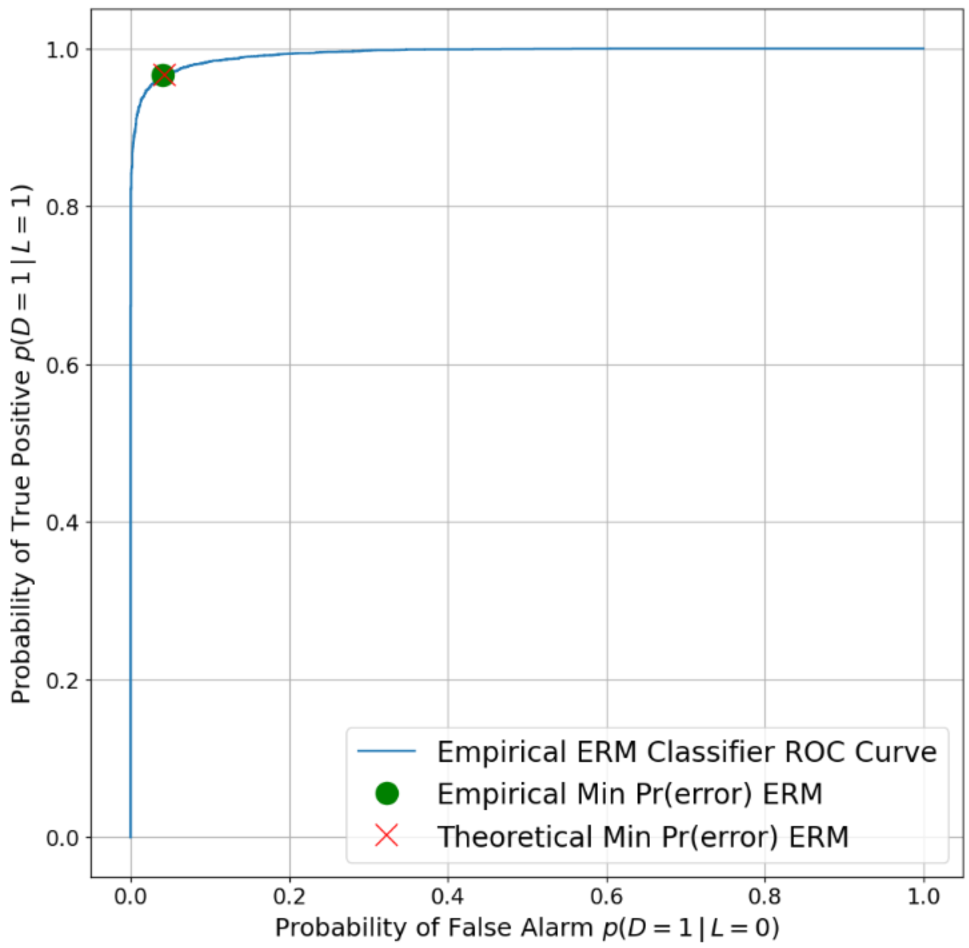
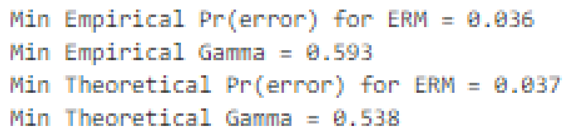


Figure 2: ROC Curve with Minimum Risk

Comparing the Theoretical and Experimental Values of Gamma and Probability Error shown in Figure 2:

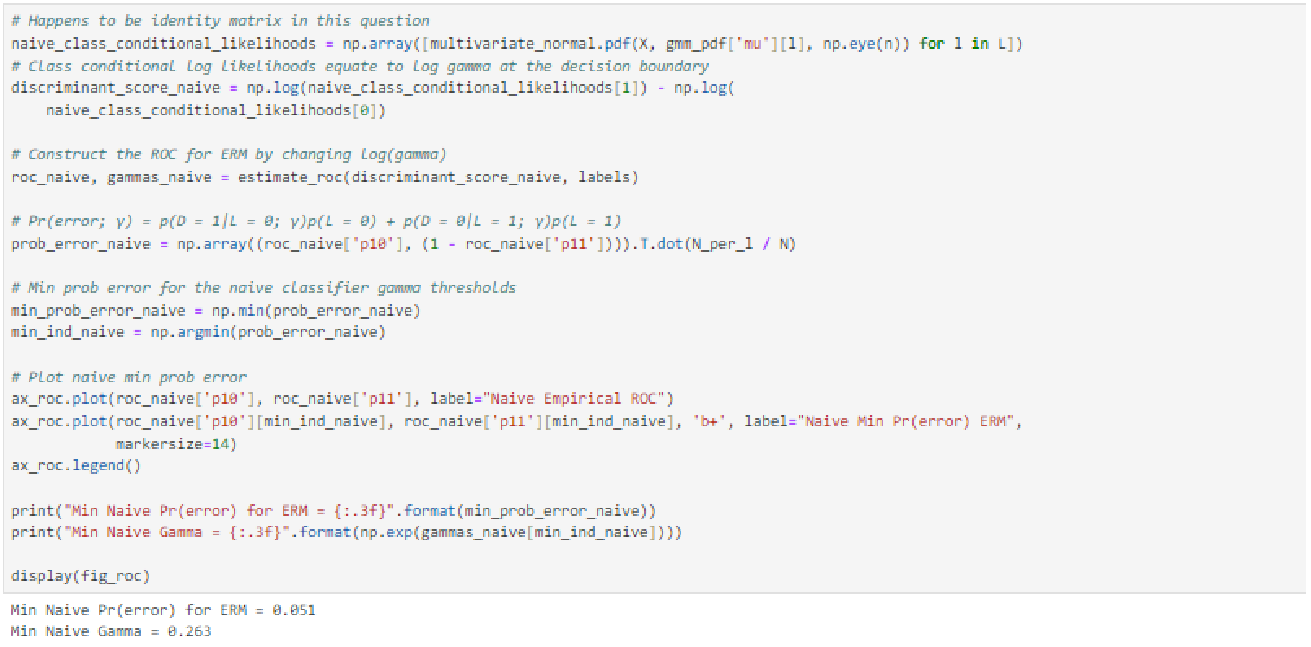


```
Min Empirical Pr(error) for ERM = 0.036
Min Empirical Gamma = 0.593
Min Theoretical Pr(error) for ERM = 0.037
Min Theoretical Gamma = 0.538
```

The empirical and theoretical minimum error rates coincide, indicating their accuracy. The only minor distinction lies in the value of the Gamma parameter.

Part B: ERM classification attempt using incorrect knowledge of data distribution (Naive Bayesian Classifier, which assumes features are independent given each class label)... For this part, assume that you know the true class prior probabilities, but for some reason you think that the class conditional pdfs are both Gaussian with the true means, but (incorrectly) with covariance matrices that are diagonal (with diagonal entries equal to true variances, off-diagonal entries equal to zeros, consistent with the independent feature assumption of Naive Bayes). Analyze the impact of this model mismatch in this Naive Bayesian (NB) approach to classifier design by repeating the same steps in Part A on the same 10K sample data set you generated earlier. Report the same results, answer the same questions. Did this model mismatch negatively impact your ROC curve and minimum achievable probability of error?

Solution:



```python
# Happens to be identity matrix in this question
naive_class_conditional_likelihoods = np.array([multivariate_normal.pdf(X, gmm_pdf['mu'][l], np.eye(n)) for l in L])
# Class conditional log likelihoods equate to log gamma at the decision boundary
discriminant_score_naive = np.log(naive_class_conditional_likelihoods[1]) - np.log(
    naive_class_conditional_likelihoods[0])

# Construct the ROC for ERM by changing log(gamma)
roc_naive, gammas_naive = estimate_roc(discriminant_score_naive, labels)

# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
prob_error_naive = np.array((roc_naive['p10'], (1 - roc_naive['p11']))).T.dot(N_per_l / N)

# Min prob error for the naive classifier gamma thresholds
min_prob_error_naive = np.min(prob_error_naive)
min_ind_naive = np.argmin(prob_error_naive)

# Plot naive min prob error
ax_roc.plot(roc_naive['p10'], roc_naive['p11'], label="Naive Empirical ROC")
ax_roc.plot(roc_naive['p10'][min_ind_naive], roc_naive['p11'][min_ind_naive], 'b+', label="Naive Min Pr(error) ERM",
            markersize=14)
ax_roc.legend()

print("Min Naive Pr(error) for ERM = {:.3f}".format(min_prob_error_naive))
print("Min Naive Gamma = {:.3f}".format(np.exp(gammas_naive[min_ind_naive])))

display(fig_roc)
```

```
Min Naive Pr(error) for ERM = 0.051
Min Naive Gamma = 0.263
```

Code explanation:

1. First, we begin by calculating the class conditional likelihoods for the two classes (0 and 1) using a simplified model. In this case, a multivariate normal probability density function (PDF) is employed with the assumption that the covariance matrix is an identity matrix (np.eye(n)). This means that the features are assumed to be independent, and the PDF is isotropic for both classes. The likelihoods for each class are computed and stored in the naive_class_conditional_likelihoods array.
2. The discriminant score for the naive classifier is calculated by taking the logarithm of the likelihood of class 1 minus the logarithm of the likelihood of class 0. This score represents the decision boundary for the classifier.
3. The ROC curve for the naive classifier is constructed. This involves changing the log(gamma) threshold values. The estimate_roc function is used to calculate the ROC curve and obtain corresponding gamma values.
4. Then we compute the probability of error for different gamma thresholds. The probability of error is determined by a weighted combination of the probability of a false positive (p(D=1|L=0; γ) * p(L=0)) and the probability of a false negative (p(D=0|L=1; γ) * p(L=1)). This probability of error is calculated for various gamma values and stored in the prob_error_naive array.
5. Then we find the minimum probability of error and identifies the corresponding index. This minimum error point represents the optimal operating point for the naive classifier. The minimum probability of error and its associated gamma are stored as min_prob_error_naive and min_ind_naive, respectively.
6. The ROC curve for the naive classifier is plotted and the we finally print out the minimum probability of error and the corresponding gamma for the naive classifier.

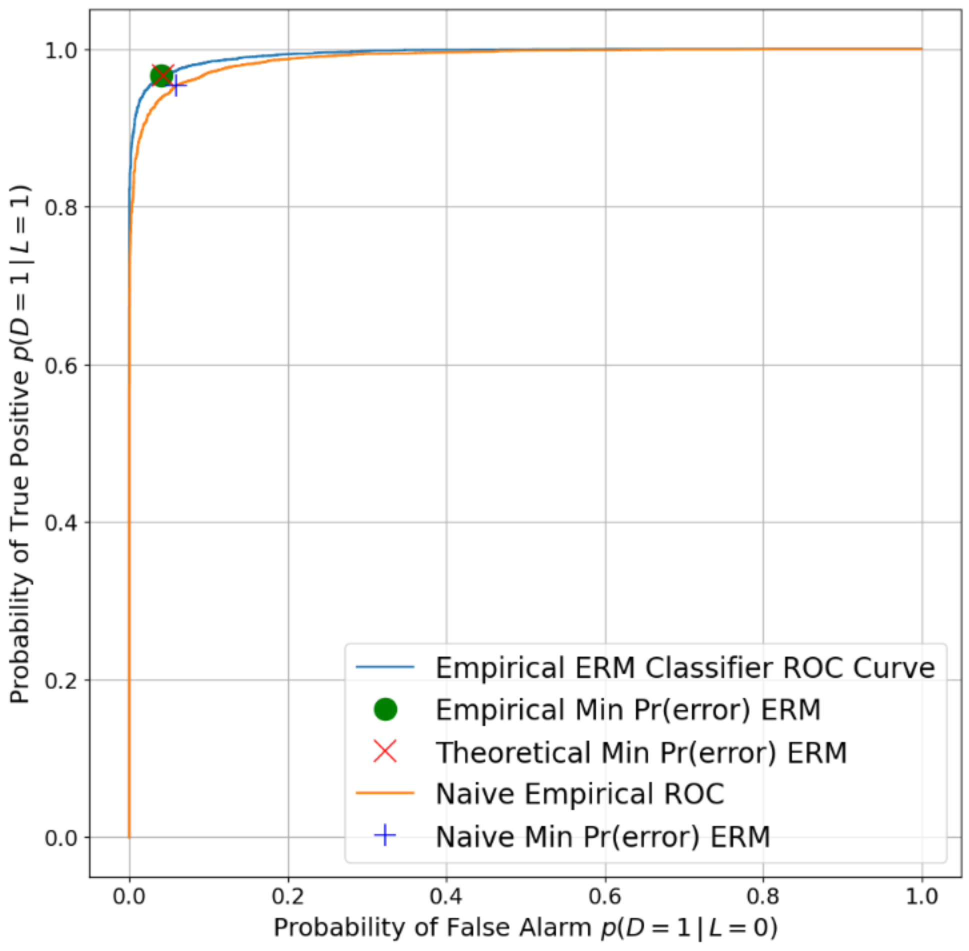
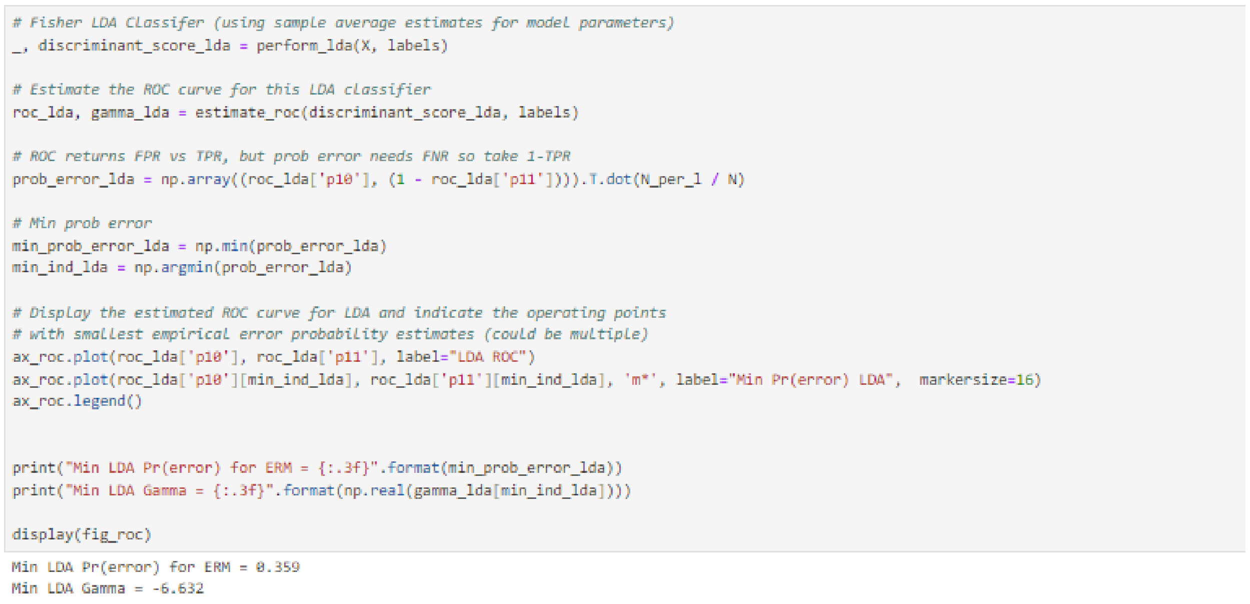


Figure 3: ROC Curve for Naive Bayesian Classifiers

Figure 3 Shows the ROC curve of Naive Bayesian Classifier. The procedure to calculate the gamma and error probabilities are same to Part A. The model mismatch did not negatively impact the ROC curve and the probability error are slightly higher.

Part C: In the third part of this exercise, repeat the same steps as in the previous two cases, but this time using a Fisher Linear Discriminant Analysis (LDA) based classifier. Using the 10K available samples, estimate the class conditional pdf mean and covariance matrices using sample average estimators for mean and covariance. From these estimated mean vectors and covariance matrices, determine the Fisher LDA projection weight vector (via the generalized eigen decomposition of within and between class scatter matrices): wLDA. For the classification rule w T LDAx compared to a threshold τ, which takes values from −∞ to ∞, trace the ROC curve. Identify the threshold at which the probability of error (based on sample count estimates) is minimized, and clearly mark that operating point on the ROC curve estimate. Discuss how this LDA classifier performs relative to the previous two classifiers.

Solution:



```
# Fisher LDA Classifer (using sample average estimates for model parameters)
_, discriminant_score_lda = perform_lda(X, labels)

# Estimate the ROC curve for this LDA classifier
roc_lda, gamma_lda = estimate_roc(discriminant_score_lda, labels)

# ROC returns FPR vs TPR, but prob error needs FNR so take 1-TPR
prob_error_lda = np.array((roc_lda['p10'], (1 - roc_lda['p11']))).T.dot(N_per_l / N)

# Min prob error
min_prob_error_lda = np.min(prob_error_lda)
min_ind_lda = np.argmin(prob_error_lda)

# Display the estimated ROC curve for LDA and indicate the operating points
# with smallest empirical error probability estimates (could be multiple)
ax_roc.plot(roc_lda['p10'], roc_lda['p11'], label="LDA ROC")
ax_roc.plot(roc_lda['p10'][min_ind_lda], roc_lda['p11'][min_ind_lda], 'm*', label="Min Pr(error) LDA",  markersize=16)
ax_roc.legend()


print("Min LDA Pr(error) for ERM = {:.3f}".format(min_prob_error_lda))
print("Min LDA Gamma = {:.3f}".format(np.real(gamma_lda[min_ind_lda])))

display(fig_roc)
```

```
Min LDA Pr(error) for ERM = 0.359
Min LDA Gamma = -6.632
```

Code Explanation:

1. First, we begin by applying Fisher's LDA to the dataset. We use the perform_lda function, which computes LDA using sample average estimates for model parameters. The output includes a discriminant score, which represents the LDA decision boundary.
2. The code proceeds to estimate the ROC curve for this LDA classifier. It uses the estimate_roc function, which calculates the ROC curve based on the discriminant scores and the true class labels.
3. The ROC curve is typically expressed in terms of True Positive Rate (TPR) vs. False Positive Rate (FPR). However, to calculate the probability of error, the code converts TPR to False Negative Rate (FNR) by subtracting TPR from 1. This is done because the probability of error needs both FPR and FNR.
4. The probability of error is computed based on the ROC curve. It is calculated for various operating points (defined by different gamma values) and is a weighted combination of FPR and FNR. The probability of error is stored in the prob_error_lda array.
5. The code identifies the minimum probability of error and determines the index associated with this minimum. This point represents the optimal operating point for the LDA classifier. The minimum probability of error and its corresponding index are stored as min_prob_error_lda and min_ind_lda.
6. The code plots the ROC curve for the LDA classifier. The code prints out the minimum probability of error and the corresponding gamma value for the LDA classifier.

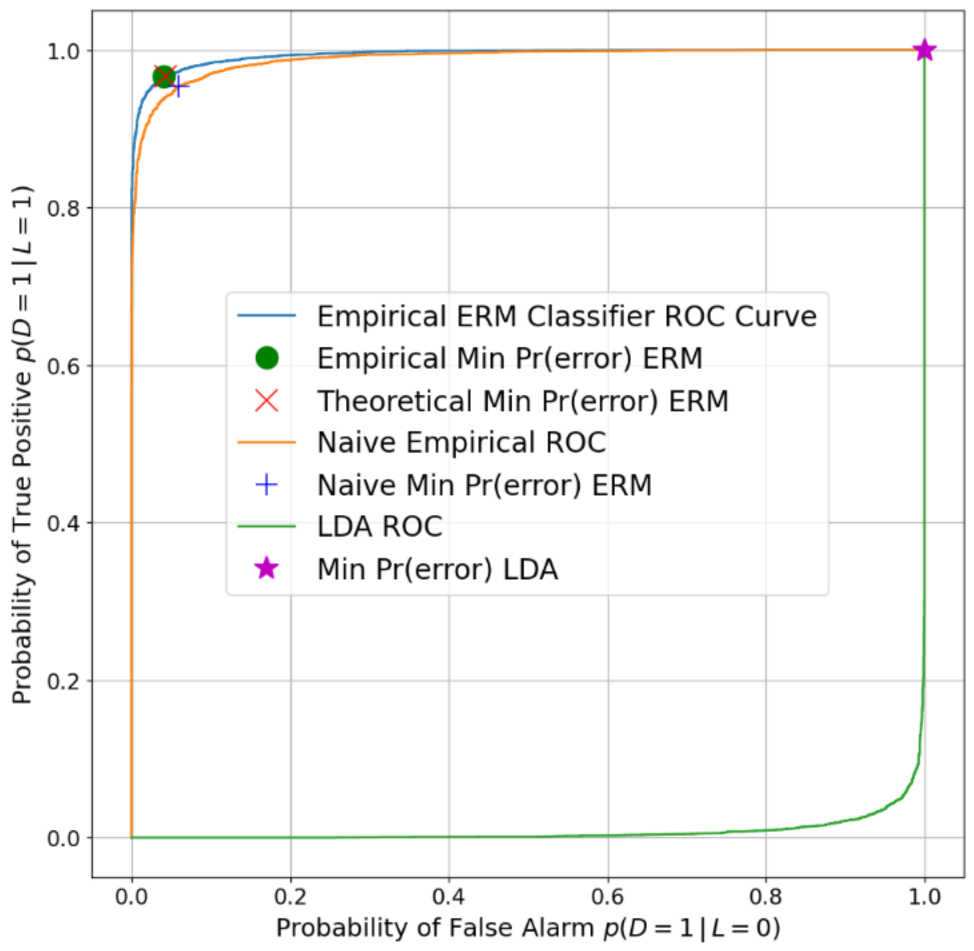


Figure 4: ROC Curve for LDA Classifier

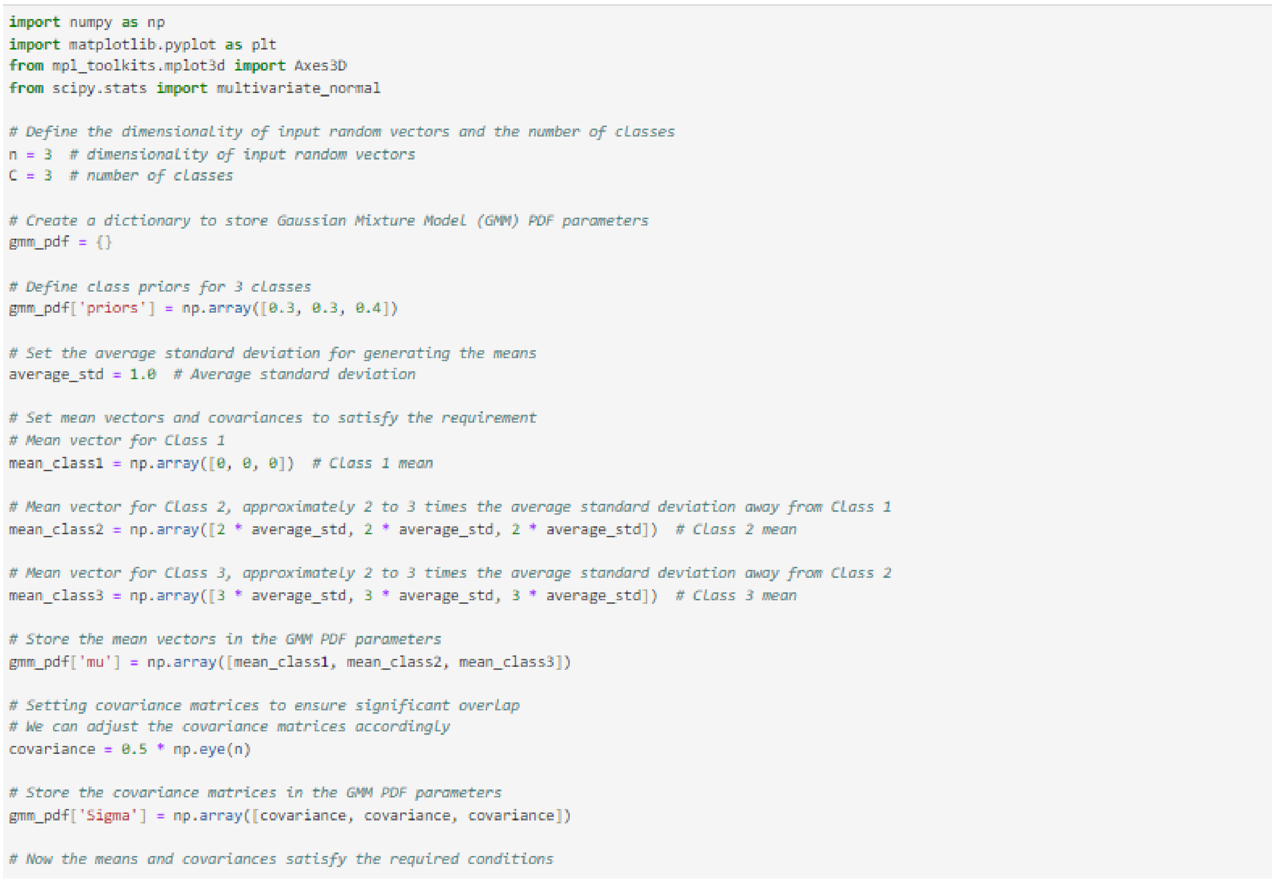
Figure 4 shows the LDA Classifier Minimum Probability which is higher than the ERM classification but less than the Naive Bayesian Classifier, so we can say that Naive Bayesian Classifier gives us less accuracy, second is a Fisher Linear Discriminant Analysis (LDA) based classifier, and the last the most better results are shown by ERM classification.

The numerical experiment is in agreement with the theoretical result.

Question 2

A 3-dimensional random vector X takes values from a mixture of four Gaussians. One of these Gaussians represent the class-conditional pdf for class 1, and another Gaussian represents the classconditional pdf for class 2. Class 3 data originates from a mixture of the remaining 2 Gaussian components with equal weights. For this setting where labels L ∈ {1,2,3}, pick your own classconditional pdfs p(x|L = j), j ∈ {1,2,3} as described. Try to approximately set the distances between means of pairs of Gaussians to approximately 2 to 3 times the average standard deviation of the Gaussian components, so that there is some significant overlap between class-conditional pdfs. Set class priors to 0.3,0.3,0.4.

Solution:



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

# Define the dimensionality of input random vectors and the number of classes
n = 3  # dimensionality of input random vectors
C = 3  # number of classes

# Create a dictionary to store Gaussian Mixture Model (GMM) PDF parameters
gmm_pdf = {}

# Define class priors for 3 classes
gmm_pdf['priors'] = np.array([0.3, 0.3, 0.4])

# Set the average standard deviation for generating the means
average_std = 1.0  # Average standard deviation

# Set mean vectors and covariances to satisfy the requirement
# Mean vector for Class 1
mean_class1 = np.array([0, 0, 0])  # Class 1 mean

# Mean vector for Class 2, approximately 2 to 3 times the average standard deviation away from Class 1
mean_class2 = np.array([2 * average_std, 2 * average_std, 2 * average_std])  # Class 2 mean

# Mean vector for Class 3, approximately 2 to 3 times the average standard deviation away from Class 2
mean_class3 = np.array([3 * average_std, 3 * average_std, 3 * average_std])  # Class 3 mean

# Store the mean vectors in the GMM PDF parameters
gmm_pdf['mu'] = np.array([mean_class1, mean_class2, mean_class3])

# Setting covariance matrices to ensure significant overlap
# We can adjust the covariance matrices accordingly
covariance = 0.5 * np.eye(n)

# Store the covariance matrices in the GMM PDF parameters
gmm_pdf['Sigma'] = np.array([covariance, covariance, covariance])

# Now the means and covariances satisfy the required conditions
```

Part A: Minimum probability of error classification (0-1 loss, also referred to as Bayes Decision rule or MAP classifier).

1. Generate 10000 samples from this data distribution and keep track of the true labels of each sample.

Solution:

```python
# Define the number of samples to generate
N = 10000;

# Generate synthetic data samples from the specified GMM PDF
X, labels = generate_data_from_gmm(N, gmm_pdf)

# Create an array to represent the class labels
L = np.array(range(C))

# Count the number of samples per class
N_per_l = np.array([sum(labels == l) for l in L])

# Print the counts of samples in each class
print(N_per_l)

# Create a figure for plotting the data
fig, ax_gmm = plt.subplots(figsize=(10, 10))

# Plot data points for Class 1 with 'o' markers
ax_gmm.plot(X[labels == 0, 0], X[labels == 0, 1], 'o', label="Class 1", markerfacecolor='none')

# Plot data points for Class 2 with 'p' markers
ax_gmm.plot(X[labels == 1, 0], X[labels == 1, 1], 'p', label="Class 2", markerfacecolor='none')

# Plot data points for Class 3 with 'g^' markers
ax_gmm.plot(X[labels == 2, 0], X[labels == 2, 1], 'g^', label="Class 3", markerfacecolor='none')

# Set labels for the x and y axes
ax_gmm.set_xlabel(r"$x_1$")
ax_gmm.set_ylabel(r"$x_2$")

# Set the aspect ratio of the plot to be equal
ax_gmm.set_aspect('equal')


# Set the title of the plot
plt.title("Data and True Class Labels")
plt.legend()
plt.tight_layout() # Adjust the layout of the plot for better presentation


# The code generates data samples from the GMM and visualizes the data and true class labels.
# The number of samples per class is calculated, and the data is plotted with different markers
# for each class. The plot is labeled, and a legend is displayed for class labels.
```

```
[3090 2964 3946]
```

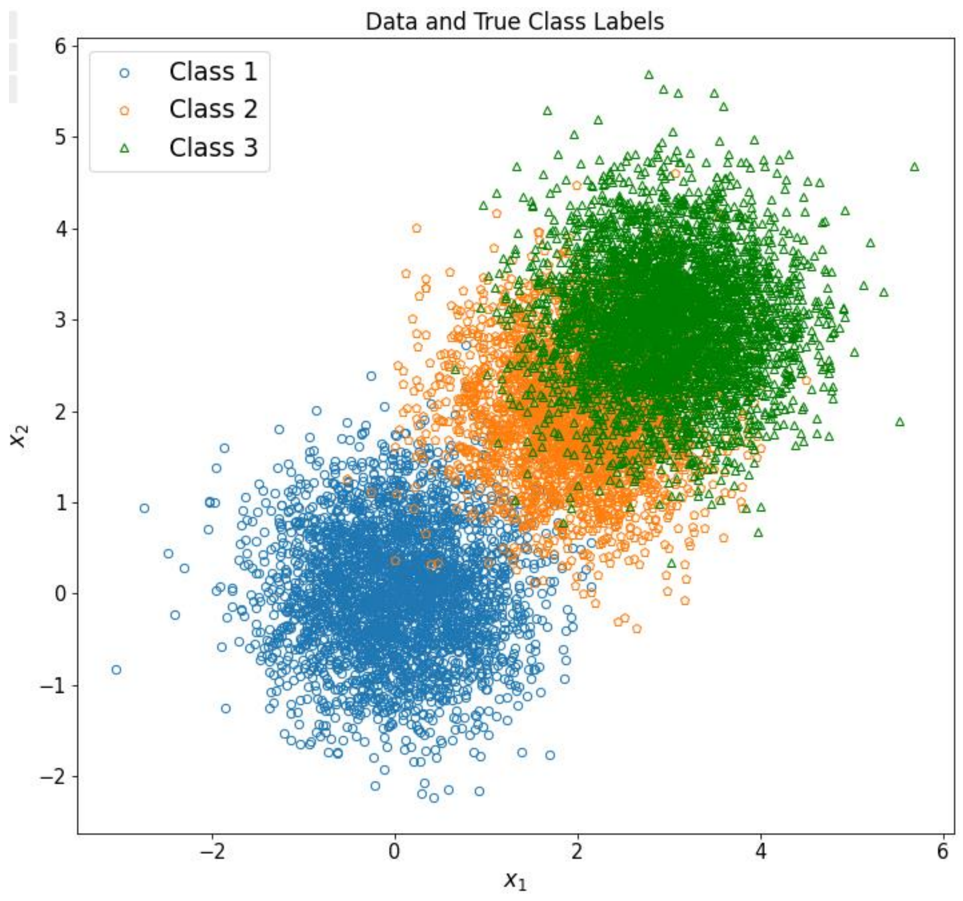


Figure 5: Sample Scatter Plot with three classes

2. Specify the decision rule that achieves minimum probability of error (i.e., use 0-1 loss), implement this classifier with the true data distribution knowledge, classify the 10K samples and count the samples corresponding to each decision-label pair to empirically estimate the confusion matrix whose entries are P(D = i|L = j) for i, j ∈ {1,2,3}.

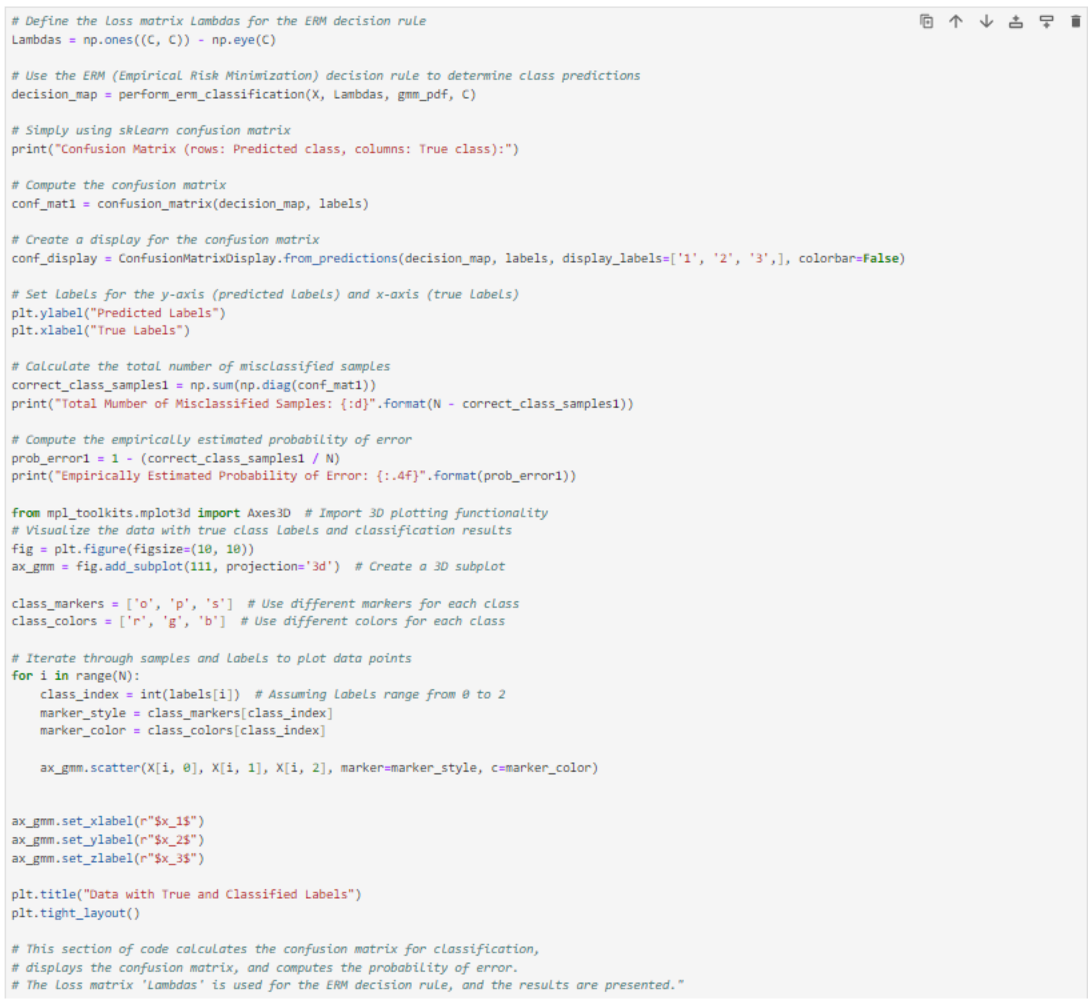
3. Provide a visualization of the data (scatter-plot in 3-dimensional space), and for each sample indicate the true class label with a different marker shape (dot, circle, triangle, square) and whether it was correctly (green) or incorrectly (red) classified with a different marker color as indicated in parentheses.

Solution:

For a given x, we need to choose a class label i that minimizes risk (or loss) associated with choosing this class label. We know that the ERM decision rule for this problem is based on conditional risk:

$$D(\mathbf{x}) = \underset{i\in\{1,2,3\}}{\operatorname{argmin}}\, R(D = i \,|\, \mathbf{x}) = \underset{i\in\{1,2,3\}}{\operatorname{argmin}} \sum_{j=1}^{3} \lambda_{ij} p(\mathbf{x} \,|\, L = j) p(L = j),$$

where the expression expands the class posteriors p(L = j | x) for j ∈ {1, 2, 3} using Bayes rule



```python
# Define the loss matrix Lambdas for the ERM decision rule
Lambdas = np.ones((C, C)) - np.eye(C)

# Use the ERM (Empirical Risk Minimization) decision rule to determine class predictions
decision_map = perform_erm_classification(X, Lambdas, gmm_pdf, C)

# Simply using sklearn confusion matrix
print("Confusion Matrix (rows: Predicted class, columns: True class):")

# Compute the confusion matrix
conf_mat1 = confusion_matrix(decision_map, labels)

# Create a display for the confusion matrix
conf_display = ConfusionMatrixDisplay.from_predictions(decision_map, labels, display_labels=['1', '2', '3',], colorbar=False)

# Set labels for the y-axis (predicted labels) and x-axis (true labels)
plt.ylabel("Predicted Labels")
plt.xlabel("True Labels")

# Calculate the total number of misclassified samples
correct_class_samples1 = np.sum(np.diag(conf_mat1))
print("Total Mumber of Misclassified Samples: {:d}".format(N - correct_class_samples1))

# Compute the empirically estimated probability of error
prob_error1 = 1 - (correct_class_samples1 / N)
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error1))

from mpl_toolkits.mplot3d import Axes3D  # Import 3D plotting functionality
# Visualize the data with true class labels and classification results
fig = plt.figure(figsize=(10, 10))
ax_gmm = fig.add_subplot(111, projection='3d')  # Create a 3D subplot

class_markers = ['o', 'p', 's']  # Use different markers for each class
class_colors = ['r', 'g', 'b']  # Use different colors for each class

# Iterate through samples and labels to plot data points
for i in range(N):
    class_index = int(labels[i])  # Assuming labels range from 0 to 2
    marker_style = class_markers[class_index]
    marker_color = class_colors[class_index]

    ax_gmm.scatter(X[i, 0], X[i, 1], X[i, 2], marker=marker_style, c=marker_color)


ax_gmm.set_xlabel(r"$x_1$")
ax_gmm.set_ylabel(r"$x_2$")
ax_gmm.set_zlabel(r"$x_3$")

plt.title("Data with True and Classified Labels")
plt.tight_layout()

# This section of code calculates the confusion matrix for classification,
# displays the confusion matrix, and computes the probability of error.
# The loss matrix 'Lambdas' is used for the ERM decision rule, and the results are presented."
```

```
Confusion Matrix (rows: Predicted class, columns: True class):
Total Mumber of Misclassified Samples: 744
Empirically Estimated Probability of Error: 0.0744
```

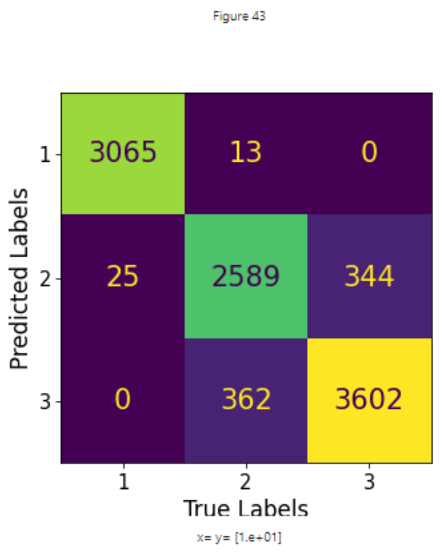


Figure 6: Confusion Matrix for Part A

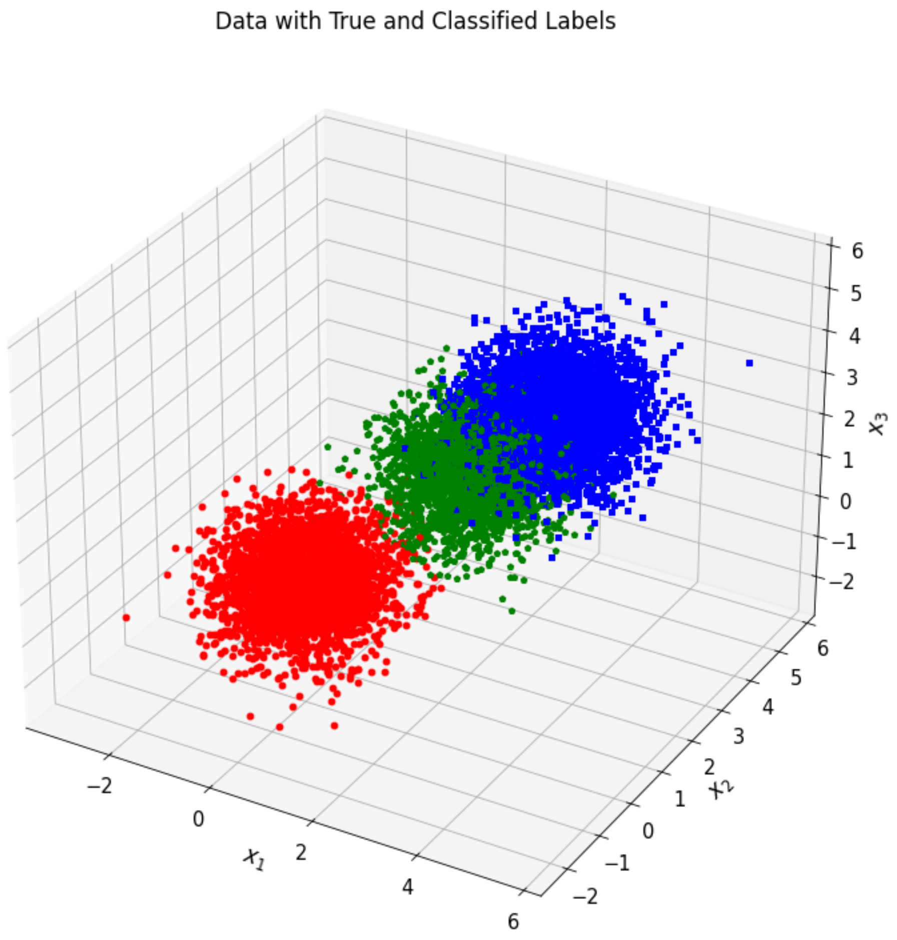


Figure 7: Scatter Plot for Predictions

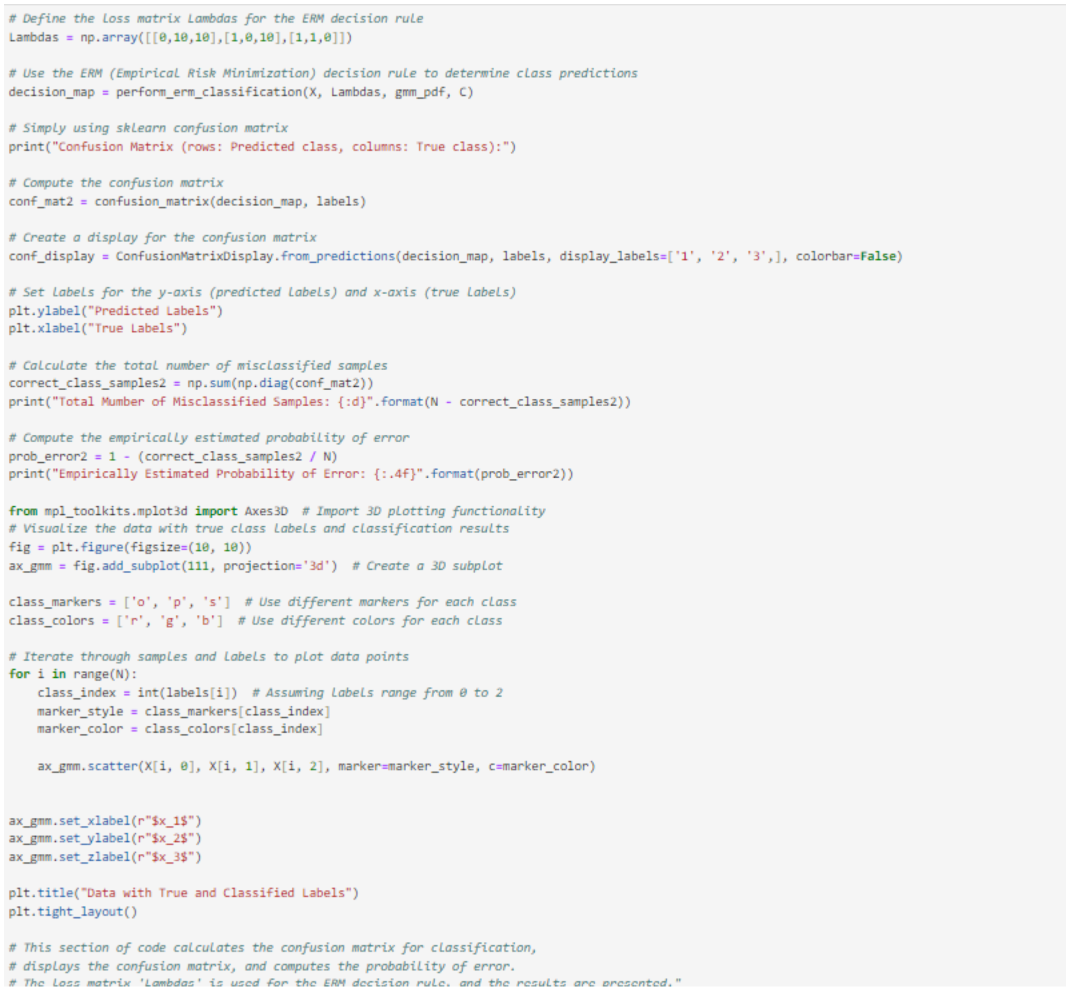
Part B: Repeat the exercise for the ERM classification rule with the following loss matrices which respectively care 10 times or 100 times more about not making mistakes when L = 3:

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

Note that, the (i, j) th entry of the loss matrix indicates the loss incurred by deciding on class i when the true label is j. For this part, using the 10K samples, estimate the minimum expected risk that this optimal ERM classification rule will achieve. Present your results with visual and numerical reprentations. Briefly discuss interesting insights, if any.

Solution:

For, Λ10:



```python
# Define the loss matrix Lambdas for the ERM decision rule
Lambdas = np.array([[0,10,10],[1,0,10],[1,1,0]])

# Use the ERM (Empirical Risk Minimization) decision rule to determine class predictions
decision_map = perform_erm_classification(X, Lambdas, gmm_pdf, C)

# Simply using sklearn confusion matrix
print("Confusion Matrix (rows: Predicted class, columns: True class):")

# Compute the confusion matrix
conf_mat2 = confusion_matrix(decision_map, labels)

# Create a display for the confusion matrix
conf_display = ConfusionMatrixDisplay.from_predictions(decision_map, labels, display_labels=['1', '2', '3',], colorbar=False)

# Set labels for the y-axis (predicted labels) and x-axis (true labels)
plt.ylabel("Predicted Labels")
plt.xlabel("True Labels")

# Calculate the total number of misclassified samples
correct_class_samples2 = np.sum(np.diag(conf_mat2))
print("Total Mumber of Misclassified Samples: {:d}".format(N - correct_class_samples2))

# Compute the empirically estimated probability of error
prob_error2 = 1 - (correct_class_samples2 / N)
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error2))

from mpl_toolkits.mplot3d import Axes3D  # Import 3D plotting functionality
# Visualize the data with true class labels and classification results
fig = plt.figure(figsize=(10, 10))
ax_gmm = fig.add_subplot(111, projection='3d')  # Create a 3D subplot

class_markers = ['o', 'p', 's']  # Use different markers for each class
class_colors = ['r', 'g', 'b']  # Use different colors for each class

# Iterate through samples and labels to plot data points
for i in range(N):
    class_index = int(labels[i])  # Assuming labels range from 0 to 2
    marker_style = class_markers[class_index]
    marker_color = class_colors[class_index]

    ax_gmm.scatter(X[i, 0], X[i, 1], X[i, 2], marker=marker_style, c=marker_color)


ax_gmm.set_xlabel(r"$x_1$")
ax_gmm.set_ylabel(r"$x_2$")
ax_gmm.set_zlabel(r"$x_3$")

plt.title("Data with True and Classified Labels")
plt.tight_layout()

# This section of code calculates the confusion matrix for classification,
# displays the confusion matrix, and computes the probability of error.
# The loss matrix 'Lambdas' is used for the ERM decision rule, and the results are presented."
```

```
Confusion Matrix (rows: Predicted class, columns: True class):
Total Mumber of Misclassified Samples: 1401
Empirically Estimated Probability of Error: 0.1401
```

Figure 45

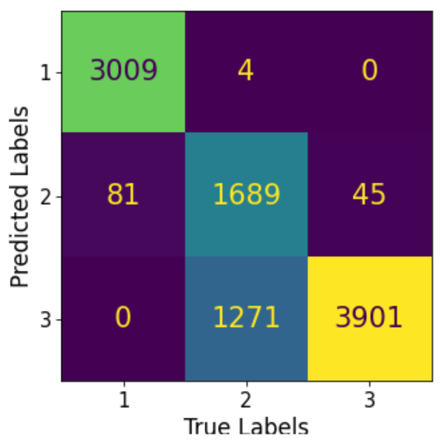


Figure 8: Confusion Matrix for $\Lambda10$

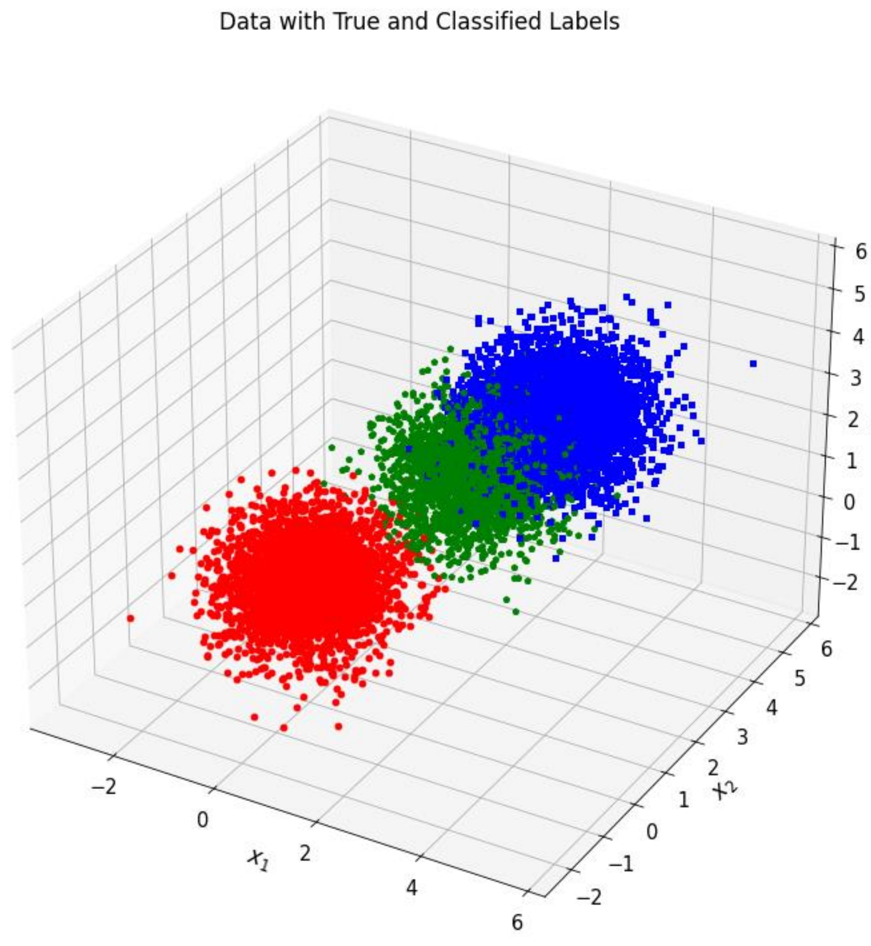
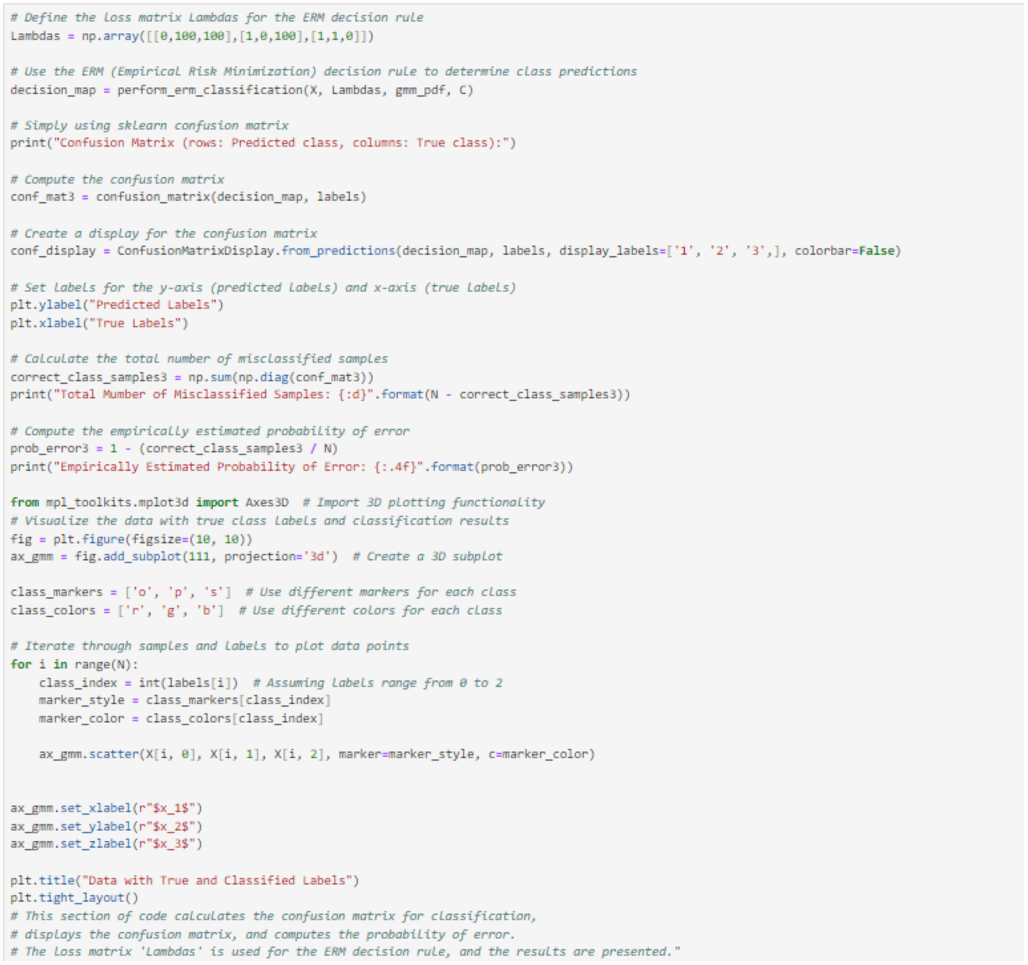
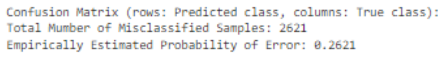


Figure 9: Scatter Plot for Predictions of Λ_{10}

For Λ100:

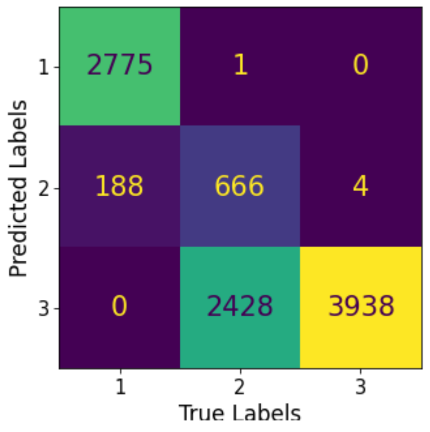


```python
# Define the loss matrix Lambdas for the ERM decision rule
Lambdas = np.array([[0,100,100],[1,0,100],[1,1,0]])

# Use the ERM (Empirical Risk Minimization) decision rule to determine class predictions
decision_map = perform_erm_classification(X, Lambdas, gmm_pdf, C)

# Simply using sklearn confusion matrix
print("Confusion Matrix (rows: Predicted class, columns: True class):")

# Compute the confusion matrix
conf_mat3 = confusion_matrix(decision_map, labels)

# Create a display for the confusion matrix
conf_display = ConfusionMatrixDisplay.from_predictions(decision_map, labels, display_labels=['1', '2', '3',], colorbar=False)

# Set labels for the y-axis (predicted labels) and x-axis (true labels)
plt.ylabel("Predicted Labels")
plt.xlabel("True Labels")

# Calculate the total number of misclassified samples
correct_class_samples3 = np.sum(np.diag(conf_mat3))
print("Total Number of Misclassified Samples: {:d}".format(N - correct_class_samples3))

# Compute the empirically estimated probability of error
prob_error3 = 1 - (correct_class_samples3 / N)
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error3))

from mpl_toolkits.mplot3d import Axes3D  # Import 3D plotting functionality
# Visualize the data with true class labels and classification results
fig = plt.figure(figsize=(10, 10))
ax_gmm = fig.add_subplot(111, projection='3d')  # Create a 3D subplot

class_markers = ['o', 'p', 's']  # Use different markers for each class
class_colors = ['r', 'g', 'b']  # Use different colors for each class

# Iterate through samples and labels to plot data points
for i in range(N):
    class_index = int(labels[i])  # Assuming labels range from 0 to 2
    marker_style = class_markers[class_index]
    marker_color = class_colors[class_index]

    ax_gmm.scatter(X[i, 0], X[i, 1], X[i, 2], marker=marker_style, c=marker_color)


ax_gmm.set_xlabel(r"$x_1$")
ax_gmm.set_ylabel(r"$x_2$")
ax_gmm.set_zlabel(r"$x_3$")

plt.title("Data with True and Classified Labels")
plt.tight_layout()
# This section of code calculates the confusion matrix for classification,
# displays the confusion matrix, and computes the probability of error.
# The loss matrix 'Lambdas' is used for the ERM decision rule, and the results are presented."
```



```
Confusion Matrix (rows: Predicted class, columns: True class):
Total Mumber of Misclassified Samples: 2621
Empirically Estimated Probability of Error: 0.2621
```

Figure 15



	1	2	3
1	2775	1	0
2	188	666	4
3	0	2428	3938

x= y= [4.]

Figure 16

Figure 10: Confusion Matrix for Λ100

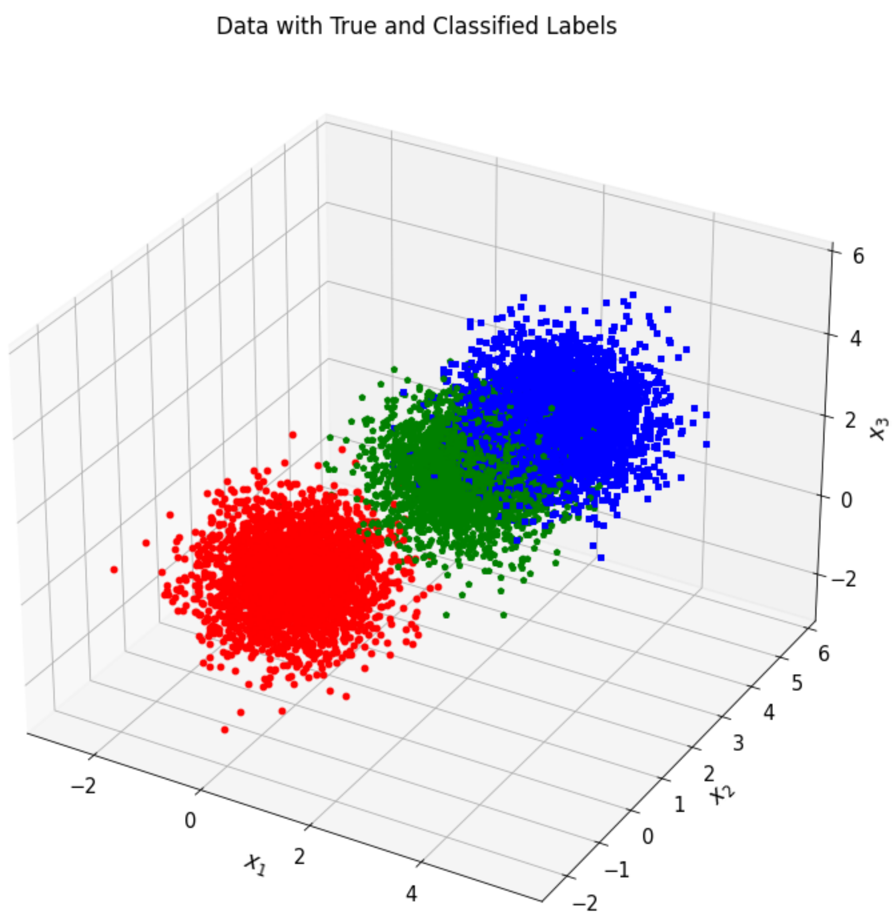


Figure 10: Scatter Plot for Predictions of Λ100

The ERM (Empirical Risk Minimization) classifier in Part B, which utilizes a loss matrix, places a heavier penalty on misclassifications between distant classes compared to the classifier in Part A. This is in contrast to the Part A classifier, which treats all misclassifications equally. As a consequence, the Part B classifier may exhibit a higher error rate because it prioritizes correctly classifying outlier samples that could be mistakenly assigned to distant and incorrect classes. However, this approach may lead to a lower penalty for errors when dealing with frequently occurring adjacent classes in comparison to the Part A classifier.

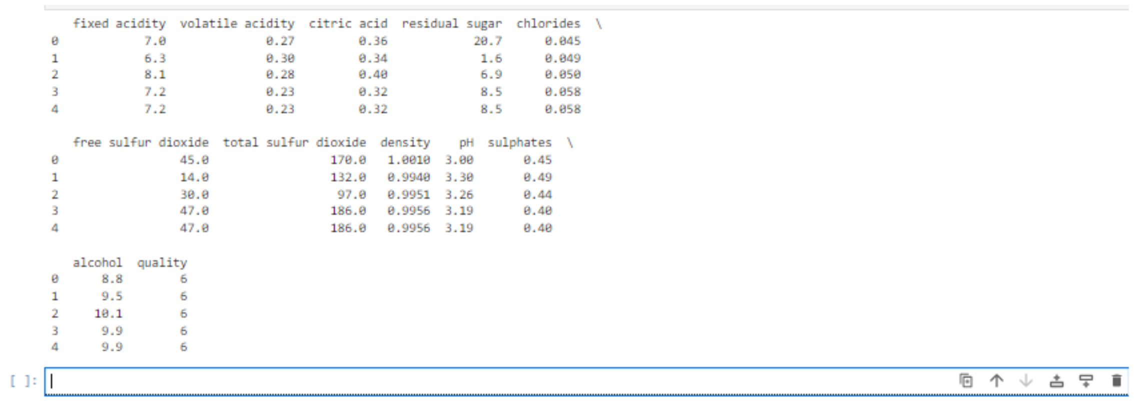
In particular, we anticipate that the Part B classifier will yield lower false probability scores than the Part A classifier in situations where there is a higher risk associated with misclassifying a sample as a distant class, as indicated in the confusion matrices. This effect becomes more pronounced when the class-conditional probability density functions overlap significantly. We can also observe that the error rate for 100 instances is more substantial than that for 10 instances, highlighting the impact of the loss matrix on the classification performance.

Question 3:
Implement minimum-probability-of-error classifiers for these problems, assuming that the class conditional pdf of features for each class you encounter in these examples is a Gaussian.

Solution:
The white wine dataset consists of 11 features and class labels from 0 to 10 indicating the scores of wine quality the total number of samples are 4898.

```python
import pandas as pd

# URL for the Wine Quality dataset in CSV format
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv'

# Read the dataset with the correct delimiter (semicolon)
wine_df = pd.read_csv(url, delimiter=';')

# Check the first few rows of the DataFrame
print(wine_df.head())
```

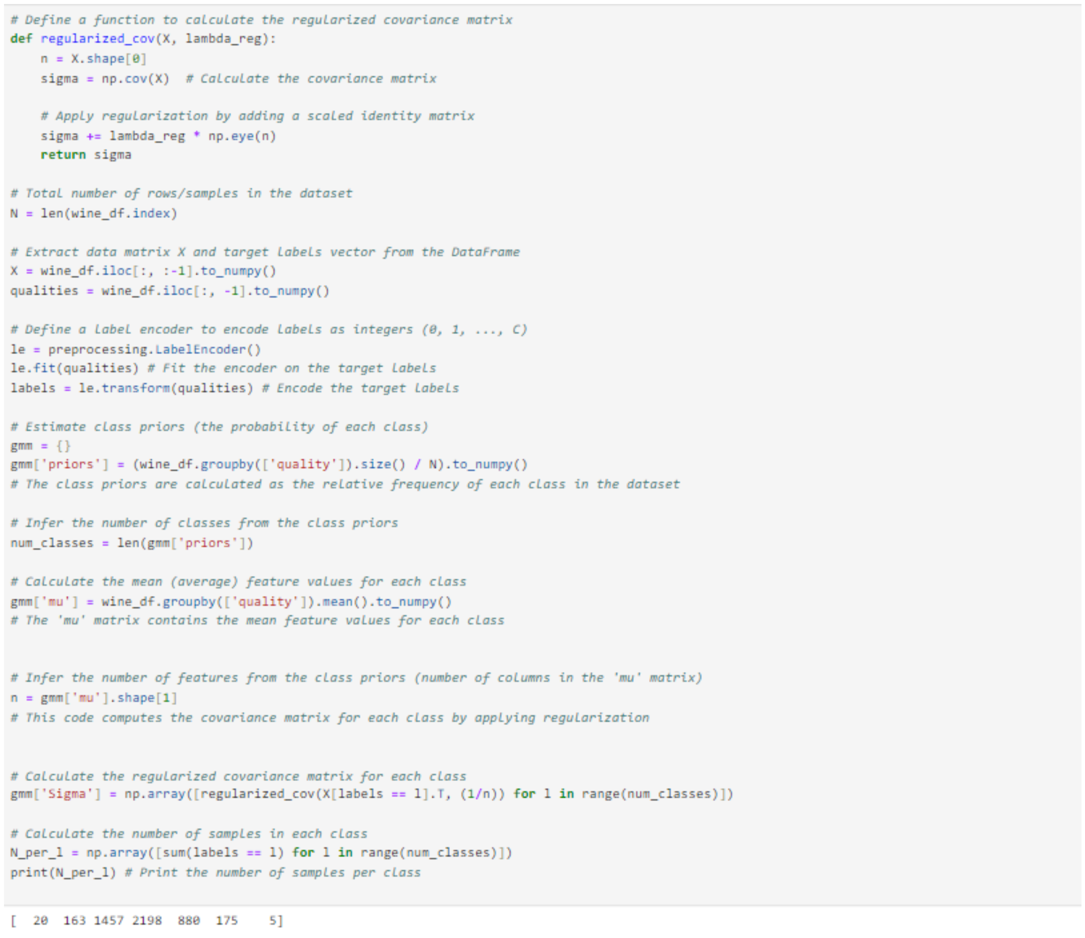
The dataset output that is obtained when the above code is run is as follows:



```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.0              0.27         0.36            20.7      0.045
1            6.3              0.30         0.34             1.6      0.049
2            8.1              0.28         0.40             6.9      0.050
3            7.2              0.23         0.32             8.5      0.058
4            7.2              0.23         0.32             8.5      0.058

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 45.0                 170.0   1.0010  3.00       0.45
1                 14.0                 132.0   0.9940  3.30       0.49
2                 30.0                  97.0   0.9951  3.26       0.44
3                 47.0                 186.0   0.9956  3.19       0.40
4                 47.0                 186.0   0.9956  3.19       0.40

   alcohol  quality
0      8.8        6
1      9.5        6
2     10.1        6
3      9.9        6
4      9.9        6
```
```
[ ]: |
```

Using all available samples from a class, with sample averages, estimate mean vectors and covariance matrices. Using sample counts, also estimate class priors. In case your sample estimates of covariance matrices are ill-conditioned, consider adding a regularization term to your covariance estimate as in: CRegularized = CSampleAverage +λI where λ > 0 is a small regularization parameter that ensures the regularized covariance matrix CRegularized has all eigenvalues larger than this parameter.

Solution:

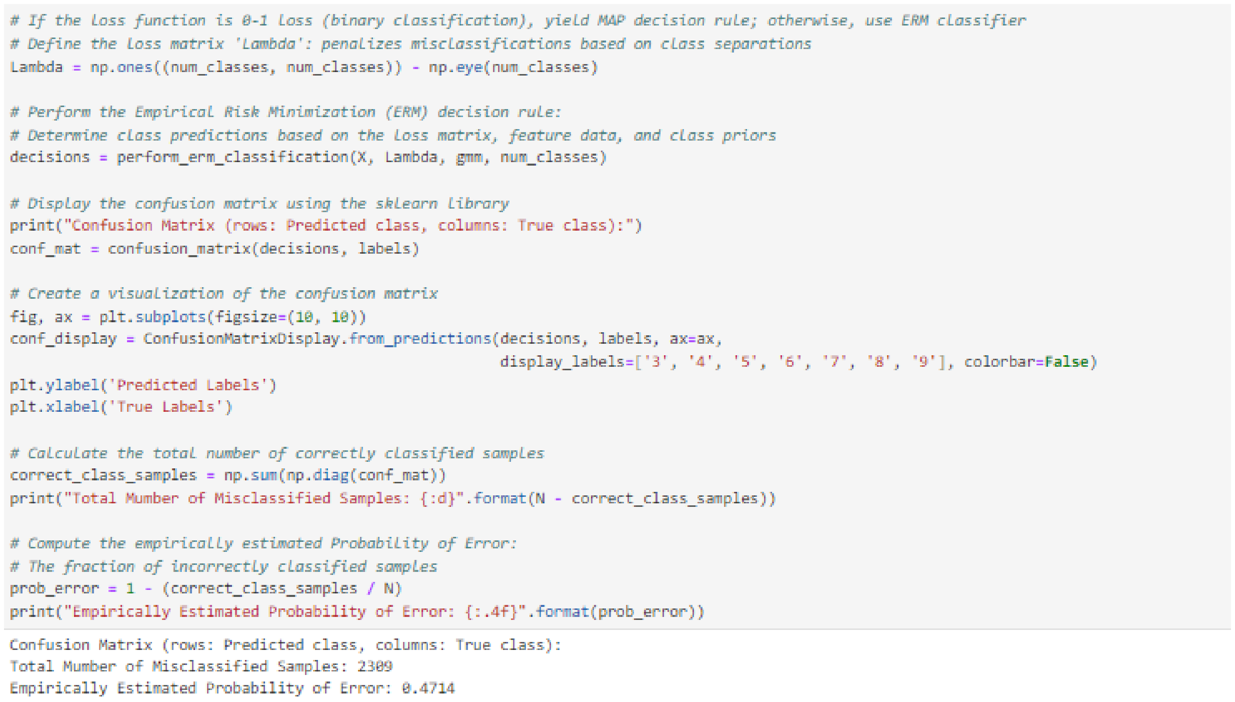


```python
# Define a function to calculate the regularized covariance matrix
def regularized_cov(X, lambda_reg):
    n = X.shape[0]
    sigma = np.cov(X)  # Calculate the covariance matrix

    # Apply regularization by adding a scaled identity matrix
    sigma += lambda_reg * np.eye(n)
    return sigma

# Total number of rows/samples in the dataset
N = len(wine_df.index)

# Extract data matrix X and target labels vector from the DataFrame
X = wine_df.iloc[:, :-1].to_numpy()
qualities = wine_df.iloc[:, -1].to_numpy()

# Define a label encoder to encode labels as integers (0, 1, ..., C)
le = preprocessing.LabelEncoder()
le.fit(qualities) # Fit the encoder on the target labels
labels = le.transform(qualities) # Encode the target labels

# Estimate class priors (the probability of each class)
gmm = {}
gmm['priors'] = (wine_df.groupby(['quality']).size() / N).to_numpy()
# The class priors are calculated as the relative frequency of each class in the dataset

# Infer the number of classes from the class priors
num_classes = len(gmm['priors'])

# Calculate the mean (average) feature values for each class
gmm['mu'] = wine_df.groupby(['quality']).mean().to_numpy()
# The 'mu' matrix contains the mean feature values for each class


# Infer the number of features from the class priors (number of columns in the 'mu' matrix)
n = gmm['mu'].shape[1]
# This code computes the covariance matrix for each class by applying regularization


# Calculate the regularized covariance matrix for each class
gmm['Sigma'] = np.array([regularized_cov(X[labels == l].T, (1/n)) for l in range(num_classes)])

# Calculate the number of samples in each class
N_per_l = np.array([sum(labels == l) for l in range(num_classes)])
print(N_per_l) # Print the number of samples per class
```

```
[  20  163 1457 2198  880  175    5]
```

With these estimated (trained) Gaussian class conditional pdfs and class priors, apply the minimum-P(error) classification rule on all (training) samples, count the errors, and report the error probability estimate you obtain for each problem. Also report the confusion matrices for both datasets, for this classification rule

Solution:



```
# If the loss function is 0-1 loss (binary classification), yield MAP decision rule; otherwise, use ERM classifier
# Define the loss matrix 'Lambda': penalizes misclassifications based on class separations
Lambda = np.ones((num_classes, num_classes)) - np.eye(num_classes)

# Perform the Empirical Risk Minimization (ERM) decision rule:
# Determine class predictions based on the loss matrix, feature data, and class priors
decisions = perform_erm_classification(X, Lambda, gmm, num_classes)

# Display the confusion matrix using the sklearn library
print("Confusion Matrix (rows: Predicted class, columns: True class):")
conf_mat = confusion_matrix(decisions, labels)

# Create a visualization of the confusion matrix
fig, ax = plt.subplots(figsize=(10, 10))
conf_display = ConfusionMatrixDisplay.from_predictions(decisions, labels, ax=ax,
                                                       display_labels=['3', '4', '5', '6', '7', '8', '9'], colorbar=False)
plt.ylabel('Predicted Labels')
plt.xlabel('True Labels')

# Calculate the total number of correctly classified samples
correct_class_samples = np.sum(np.diag(conf_mat))
print("Total Number of Misclassified Samples: {:d}".format(N - correct_class_samples))

# Compute the empirically estimated Probability of Error:
# The fraction of incorrectly classified samples
prob_error = 1 - (correct_class_samples / N)
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error))
```

```
Confusion Matrix (rows: Predicted class, columns: True class):
Total Number of Misclassified Samples: 2309
Empirically Estimated Probability of Error: 0.4714
```

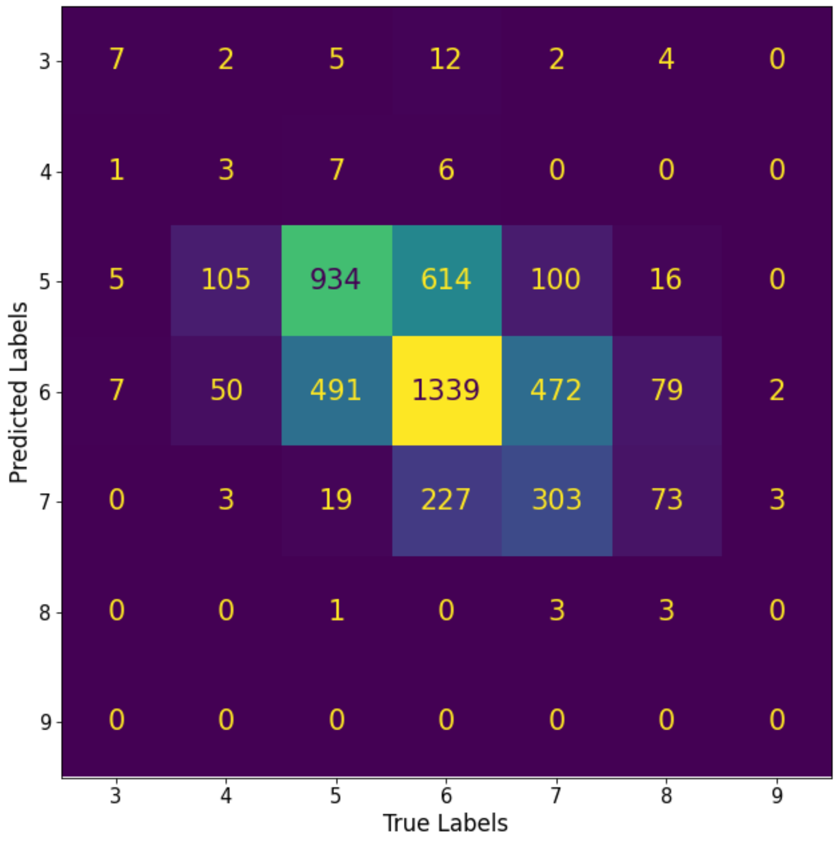
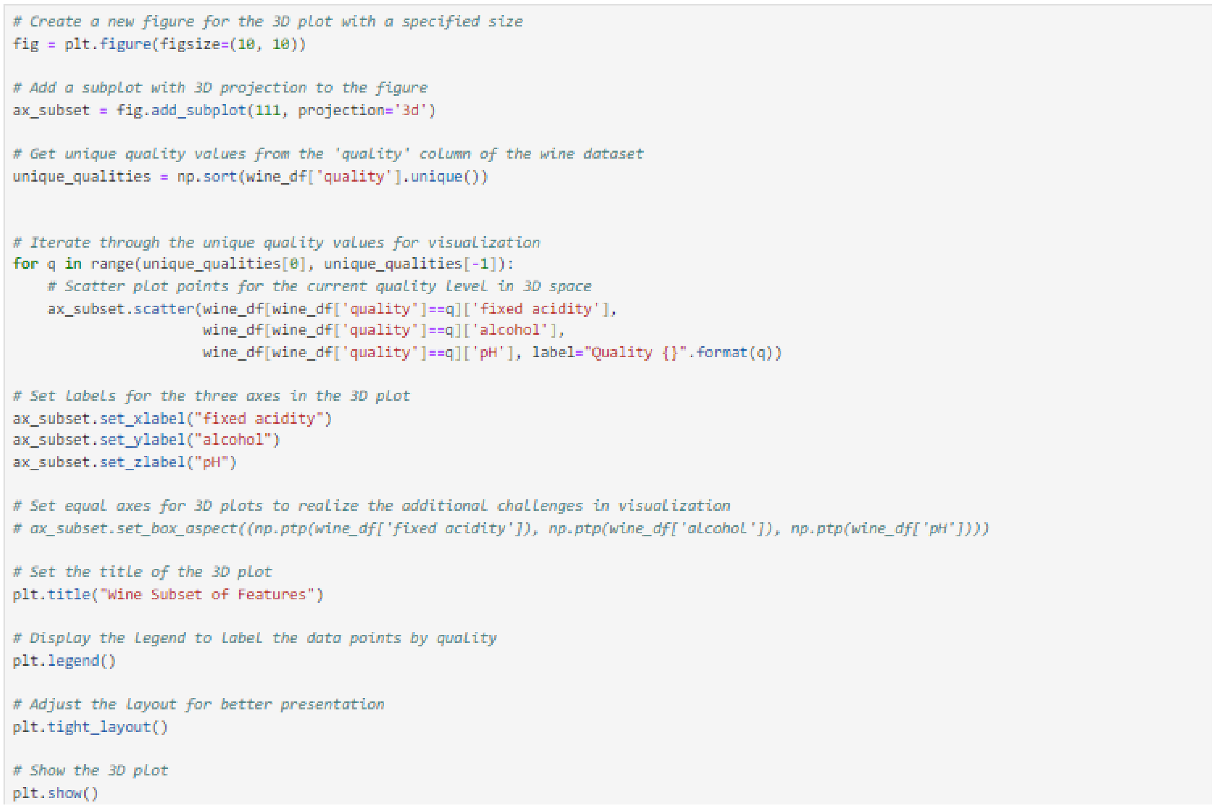


Figure 11: Confusion Matrix of Wine Database to detect the Misclassified Sample

Visualize the datasets in various 2 or 3 dimensional projections (either subsets of features, or using the first few principal components). Discuss if Gaussian class conditional models are appropriate for these datasets and how your model choice might have influenced the confusion matrix and probability of error values you obtained in the experiments conducted above. Make sure you explain in rigorous detail what your modeling assumptions are, how you estimated/selected necessary parameters for your model and classification rule, and describe your analyses in mathematical terms supplemented by numerical and visual results in a way that conveys your understanding of what you have accomplished and demonstrated.

Solution:



```
# Create a new figure for the 3D plot with a specified size
fig = plt.figure(figsize=(10, 10))

# Add a subplot with 3D projection to the figure
ax_subset = fig.add_subplot(111, projection='3d')

# Get unique quality values from the 'quality' column of the wine dataset
unique_qualities = np.sort(wine_df['quality'].unique())


# Iterate through the unique quality values for visualization
for q in range(unique_qualities[0], unique_qualities[-1]):
    # Scatter plot points for the current quality level in 3D space
    ax_subset.scatter(wine_df[wine_df['quality']==q]['fixed acidity'],
                      wine_df[wine_df['quality']==q]['alcohol'],
                      wine_df[wine_df['quality']==q]['pH'], label="Quality {}".format(q))

# Set labels for the three axes in the 3D plot
ax_subset.set_xlabel("fixed acidity")
ax_subset.set_ylabel("alcohol")
ax_subset.set_zlabel("pH")

# Set equal axes for 3D plots to realize the additional challenges in visualization
# ax_subset.set_box_aspect((np.ptp(wine_df['fixed acidity']), np.ptp(wine_df['alcohol']), np.ptp(wine_df['pH'])))

# Set the title of the 3D plot
plt.title("Wine Subset of Features")

# Display the legend to label the data points by quality
plt.legend()

# Adjust the layout for better presentation
plt.tight_layout()

# Show the 3D plot
plt.show()
```

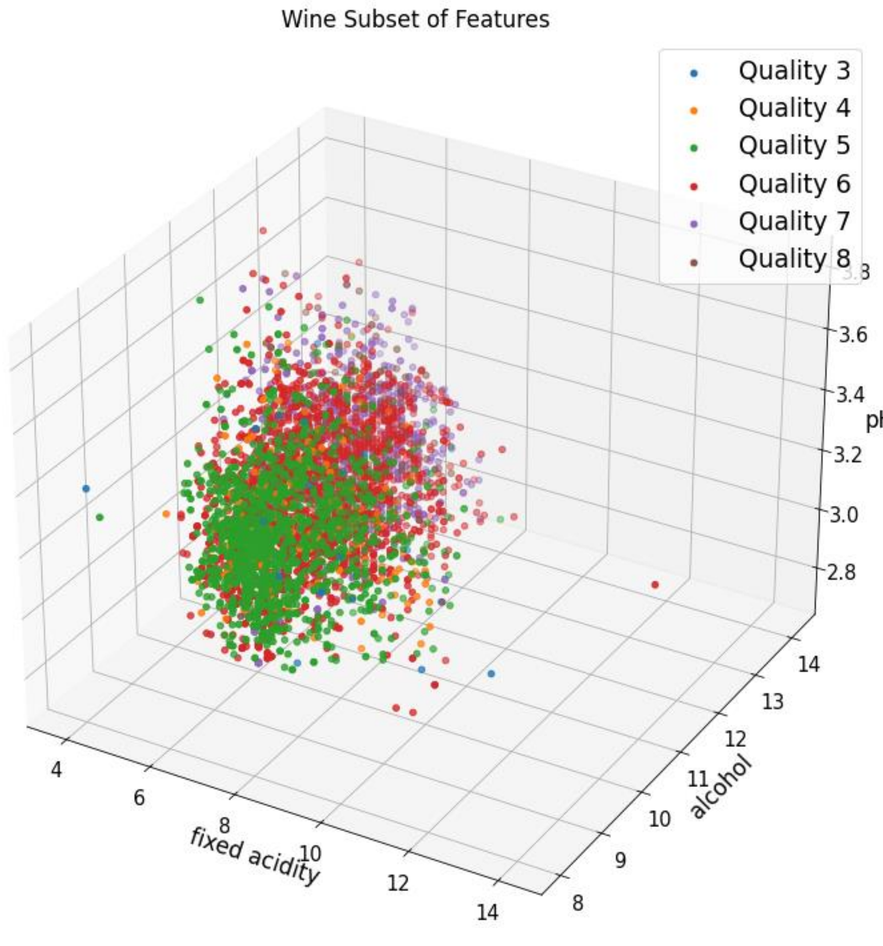
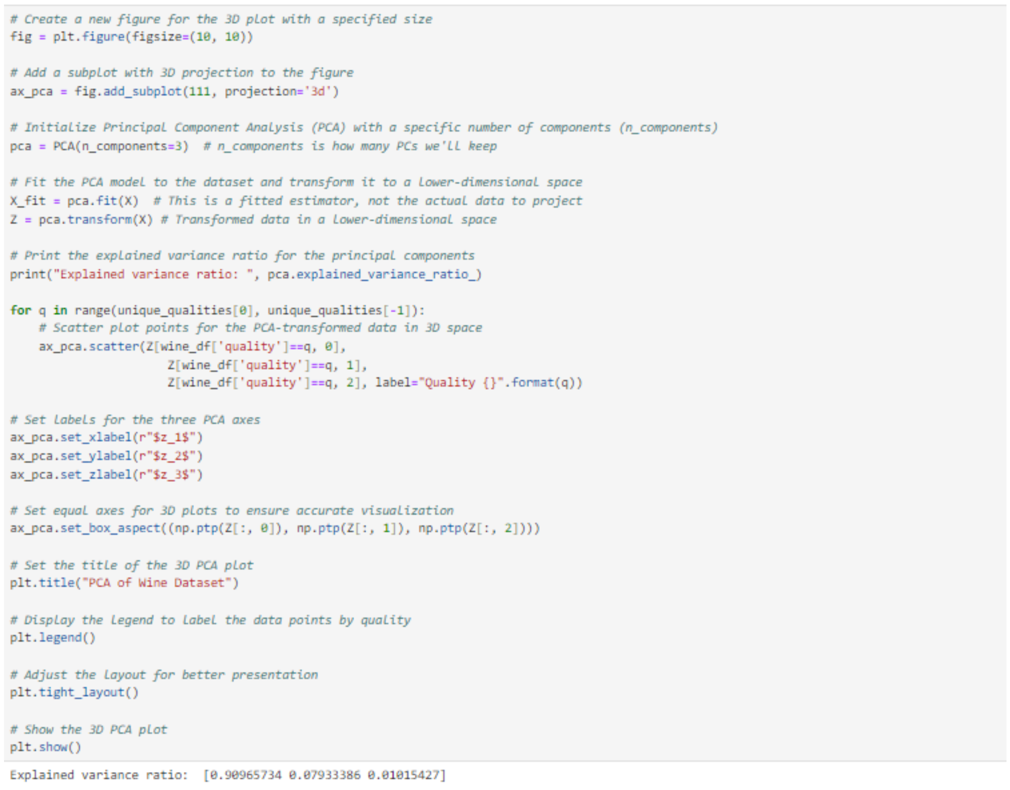


Figure 12: Scatter Plot of the subset of Wine database

It would be more effective to utilize a dimensionality reduction method, such as Principal Component Analysis (PCA), to gain a clearer understanding of whether our chosen model (based on Gaussian class-conditional distributions) is suitable for the wine-quality classification task.



```
# Create a new figure for the 3D plot with a specified size
fig = plt.figure(figsize=(10, 10))

# Add a subplot with 3D projection to the figure
ax_pca = fig.add_subplot(111, projection='3d')

# Initialize Principal Component Analysis (PCA) with a specific number of components (n_components)
pca = PCA(n_components=3)  # n_components is how many PCs we'll keep

# Fit the PCA model to the dataset and transform it to a lower-dimensional space
X_fit = pca.fit(X)  # This is a fitted estimator, not the actual data to project
Z = pca.transform(X) # Transformed data in a lower-dimensional space

# Print the explained variance ratio for the principal components
print("Explained variance ratio: ", pca.explained_variance_ratio_)

for q in range(unique_qualities[0], unique_qualities[-1]):
    # Scatter plot points for the PCA-transformed data in 3D space
    ax_pca.scatter(Z[wine_df['quality']==q, 0],
                   Z[wine_df['quality']==q, 1],
                   Z[wine_df['quality']==q, 2], label="Quality {}".format(q))

# Set labels for the three PCA axes
ax_pca.set_xlabel(r"$z_1$")
ax_pca.set_ylabel(r"$z_2$")
ax_pca.set_zlabel(r"$z_3$")

# Set equal axes for 3D plots to ensure accurate visualization
ax_pca.set_box_aspect((np.ptp(Z[:, 0]), np.ptp(Z[:, 1]), np.ptp(Z[:, 2])))

# Set the title of the 3D PCA plot
plt.title("PCA of Wine Dataset")

# Display the legend to label the data points by quality
plt.legend()

# Adjust the layout for better presentation
plt.tight_layout()

# Show the 3D PCA plot
plt.show()
```

```
Explained variance ratio:  [0.90965734 0.07933386 0.01015427]
```

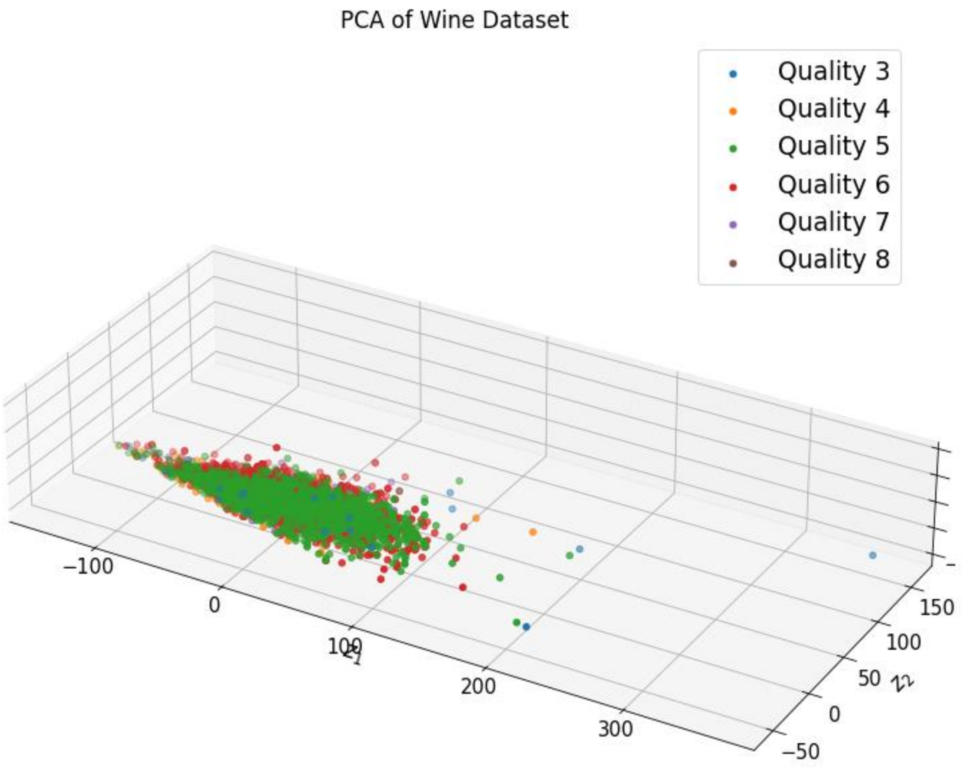
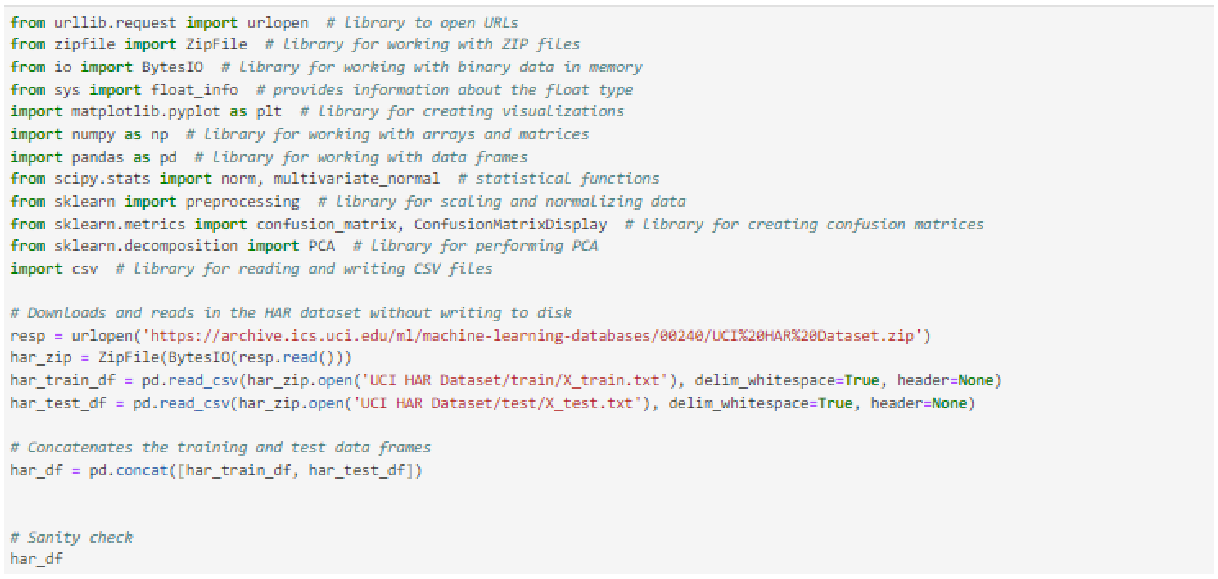


Figure 13: PCA Scatter Plot of the subset of Wine database

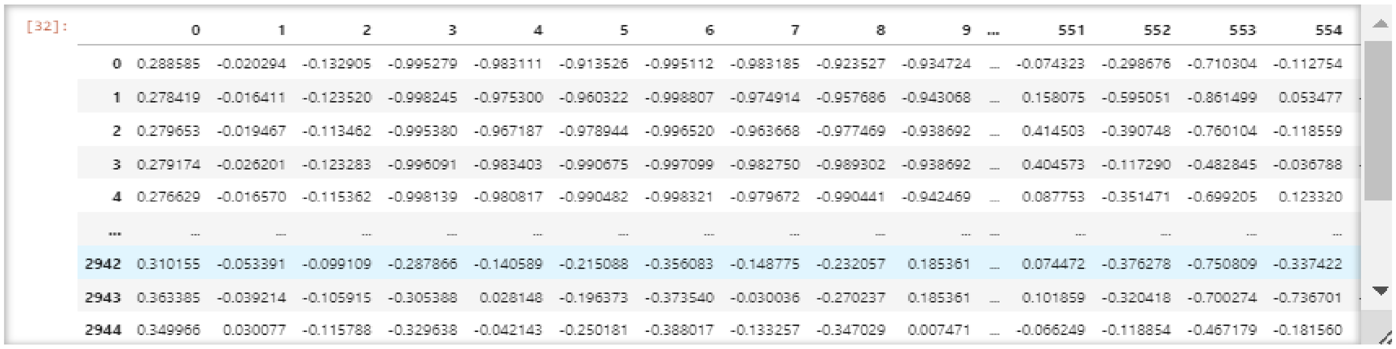
The PCA procedure effectively reduced the dimensionality of the data, primarily capturing the fraction of variance in the first three Principal Components (PCs). This transformation resulted in a dataset with a general Gaussian-like shape, characterized by elliptical patterns. However, it's important to note that in this specific case, where the Central Limit Theorem may not fully apply, the data is unlikely to conform to a typical Gaussian distribution. As a result, our assumption of a Gaussian model introduces errors into the analysis.
The main source of high probability of error estimates in this problem is likely due to irreducible error. The visual representations of the projected data reveal significant overlap between the class-specific Gaussian Probability Density Functions (PDFs). This overlap implies that our model will encounter challenges in accurately discriminating between classes based on conditional likelihood. The confusion matrix further underscores the impact of this overlap, as it shows that classes are frequently mislabeled as middle wine quality scores (e.g., 5, 6, or 7). These middle-quality scores are the most common, and their class-conditional PDFs contribute to the majority of data spread.
In summary, it's advisable to explore alternative models for this task to achieve clearer class differentiation, as the Gaussian assumption does not entirely align with the data characteristics.

HAR – Human Activity Recognition

Human Activity Recognition dataset, which consists of 561 features and 6 activity labels. There are 10299 samples.



```
from urllib.request import urlopen  # Library to open URLs
from zipfile import ZipFile  # Library for working with ZIP files
from io import BytesIO  # Library for working with binary data in memory
from sys import float_info  # provides information about the float type
import matplotlib.pyplot as plt  # Library for creating visualizations
import numpy as np  # Library for working with arrays and matrices
import pandas as pd  # Library for working with data frames
from scipy.stats import norm, multivariate_normal  # statistical functions
from sklearn import preprocessing  # Library for scaling and normalizing data
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay  # Library for creating confusion matrices
from sklearn.decomposition import PCA  # Library for performing PCA
import csv  # Library for reading and writing CSV files

# Downloads and reads in the HAR dataset without writing to disk
resp = urlopen('https://archive.ics.uci.edu/ml/machine-learning-databases/00240/UCI%20HAR%20Dataset.zip')
har_zip = ZipFile(BytesIO(resp.read()))
har_train_df = pd.read_csv(har_zip.open('UCI HAR Dataset/train/X_train.txt'), delim_whitespace=True, header=None)
har_test_df = pd.read_csv(har_zip.open('UCI HAR Dataset/test/X_test.txt'), delim_whitespace=True, header=None)

# Concatenates the training and test data frames
har_df = pd.concat([har_train_df, har_test_df])


# Sanity check
har_df
```



[32]:

	0	1	2	3	4	5	6	7	8	9	...	551	552	553	554
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	-0.074323	-0.298676	-0.710304	-0.112754
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	0.158075	-0.595051	-0.861499	0.053477
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	0.414503	-0.390748	-0.760104	-0.118559
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	0.404573	-0.117290	-0.482845	-0.036788
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	0.087753	-0.351471	-0.699205	0.123320
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2942	0.310155	-0.053391	-0.099109	-0.287866	-0.140589	-0.215088	-0.356083	-0.148775	-0.232057	0.185361	...	0.074472	-0.376278	-0.750809	-0.337422
2943	0.363385	-0.039214	-0.105915	-0.305388	0.028148	-0.196373	-0.373540	-0.030036	-0.270237	0.185361	...	0.101859	-0.320418	-0.700274	-0.736701
2944	0.349966	0.030077	-0.115788	-0.329638	-0.042143	-0.250181	-0.388017	-0.133257	-0.347029	0.007471	...	-0.066249	-0.118854	-0.467179	-0.181560

```python
# Extracts the data matrix X
X = har_df.to_numpy()

# Creates a 3D plot to visualize the PCA
fig = plt.figure(figsize=(10, 10))
ax_pca = fig.add_subplot(111, projection='3d')

# Performs PCA with 3 principal components and fits the data
pca = PCA(n_components=3)
X_fit = pca.fit(X)
Z = pca.transform(X)

# Prints out the explained variance ratio of the PCA
print("Explained variance ratio: ", pca.explained_variance_ratio_)

# Plots the transformed data in 3D
ax_pca.scatter(Z[:, 0], Z[:, 1], Z[:, 2])
ax_pca.set_xlabel(r"$z_1$")
ax_pca.set_ylabel(r"$z_2$")
ax_pca.set_zlabel(r"$z_3$")
ax_pca.set_box_aspect((np.ptp(Z[:, 0]), np.ptp(Z[:, 1]), np.ptp(Z[:, 2])))

# Sets the title and displays the plot
plt.title("PCA of HAR Dataset")
plt.tight_layout()
plt.show()
```

```
Explained variance ratio:  [0.62227069 0.04772595 0.04018191]
```

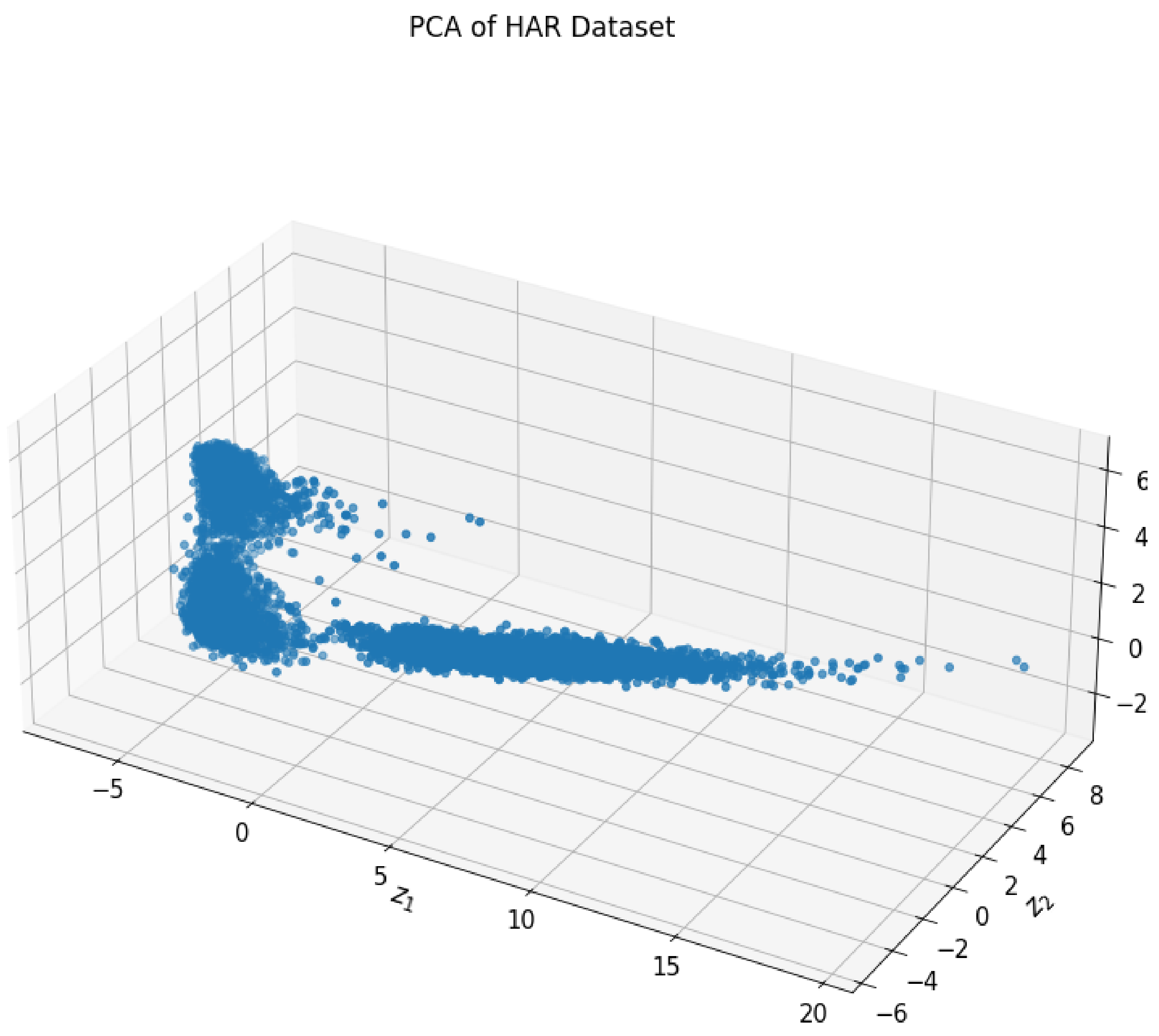


Figure 14: Empirically Estimated Probability Error Output