EECE5644 Assignment 1

Mariona Jaramillo Civill

October 8, 2024

NUID: 002413201 - jaramillocivill.m@northeastern.edu

Problem 1

First, the 10000 data samples according to the data distribution provided have been generated and saved.

Part A

The minimum expected risk classification rule in the case of 2 Gaussian class-conditionals can be expressed in the form of a likelihood-ratio test. Let λ_{ij} be the fixed non-negative loss values for each of the four cases (loss incurred by deciding class i given that the true class is j), where $i, j \in \{0, 1\}$, then:

$$\text{Decide } L_1 \text{ if } \frac{g(x|m_1, C_1)}{g(x|m_0, C_0)} \geq \frac{(\lambda_{10} - \lambda_{00})P(L_0)}{(\lambda_{01} - \lambda_{11})P(L_1)} = \gamma, \quad \text{otherwise decide } L_0$$

where $P(L_0) = 0.35$ and $P(L_1) = 0.65$.

After varying the threshold from 0 to ∞ and computing the rates for the true positive probability and false positive probability for every given threshold, the ROC curve of the minimum expected risk classifier has been plotted. Moreover, the threshold gamma that achieves minimum probability of error has been plotted in red on top of the ROC curve, i.e, the gamma that minimizes the probability of error:

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

The theoretically optimal threshold (assuming 0-1 loss, i.e. $\lambda_{00} = \lambda_{11} = 0$ and $\lambda_{01} = \lambda_{10} = 1$) comes from using the previous expression and the value of the priors:

$$\frac{g(x|m_1, C_1)}{g(x|m_0, C_0)} \geq \frac{0.35}{0.65} = 0.53$$

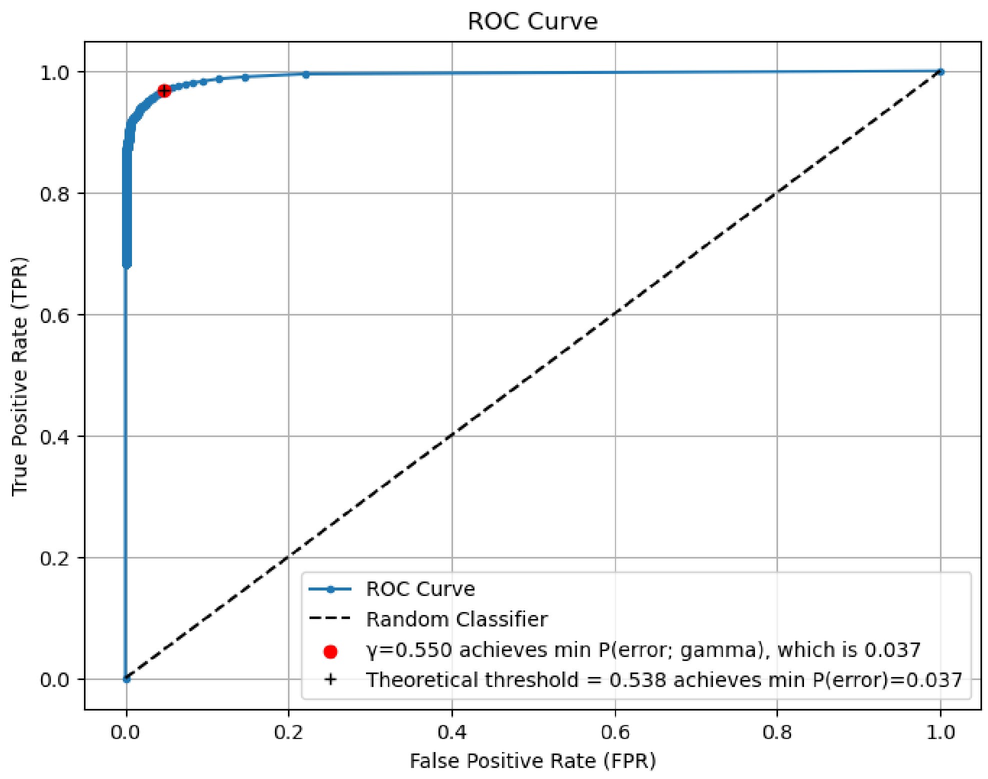


Figure 1: Approximation of the ROC curve of the minimum expected risk classifier

In Figure 1 we observe that the theoretically optimal threshold 0.538 is really close to the empirically optimal threshold 0.550 and the minimum probability of error achieved is the same for both cases. This makes sense as we are performing Expected Risk Minimization classification based on the fact that we know the true parameters of the pdfs that have generated the data.

Part B

In this case, we are using the same data samples generated previously, but now we don't know all the true parameters of the pdfs that have generated the data. In this Naive Bayesian approach we assume that the features are independent, meaning that the covariance matrix is diagonal.
In Figure 2 we can see that the ROC Curves for the original model and the Naive Bayes approach are almost the same. Also, the minimum probability of error achieved by the Naive Bayes approach is really close but slightly superior to the correct model, as in this case, we are not using the complete true distribution parameters to compute the class conditionals.

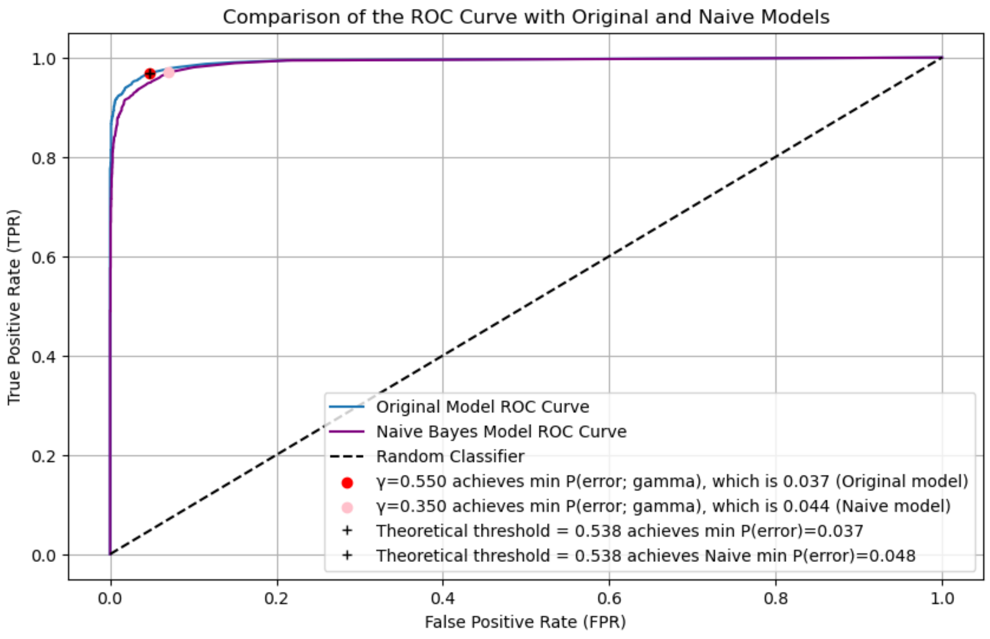


Figure 2: Approximation of the ROC curve of the minimum expected risk classifier for the original and Naive Bayes model

Part C

In the last part of this question, we don't assume we know the true or partially correct parameters from the Gaussian distributions. Instead, we have estimated the class conditional pdf's means μ_{avg} and covariance matrices Cov_{avg} using sample averages for these.
To classify the data, LDA has been used. The steps have been to compute the between class scatter matrix $S_b = \mu_{avg0} - \mu_{avg1} \otimes \mu_{avg0} - \mu_{avg1}$ and the within class scatter matrix $S_w = Cov_{avg0} + Cov_{avg1}$, to then find the projection vector w_{LDA} that maximizes the mean distance and minimizes the sum of the covariance between the two considered class sets. The optimal w_{LDA} is a generalized eigenvector of the matrix pair (S_b, S_w) that corresponds to the largest generalized eigenvalue $\frac{w^T S_b w}{w^T S_w w}$.
After this computation, we can get the LDA discriminant scores for every sample of data by performing the projection of the sample using w_{LDA}: $score = w_{LDA}^T x$.
These scores are then compared to a threshold τ that sweeps all values from $-\infty$ to $+\infty$. To make sure that this threshold effectively captures all the scores, we have taken every possible τ that is a mid point between scores. For every threshold, we have computed the corresponding decisions of belonging to one class or the other and the probability of error has been computed in every case:

$$\begin{aligned} P(\text{error}; \tau) &= P(D = 1|L = 0; \tau)P(L = 0) + P(D = 0|L = 1; \tau)P(L = 1) \\ &= \text{FPR\_lda}(\tau) \cdot p(L_0) + (1 - \text{TPR\_lda}(\tau)) \cdot p(L_1) \end{aligned}$$

The minimum probability of error has been achieved with $\tau = -0.212$, which gives a

0.042 probability of error, as can be seen in Figure 3.

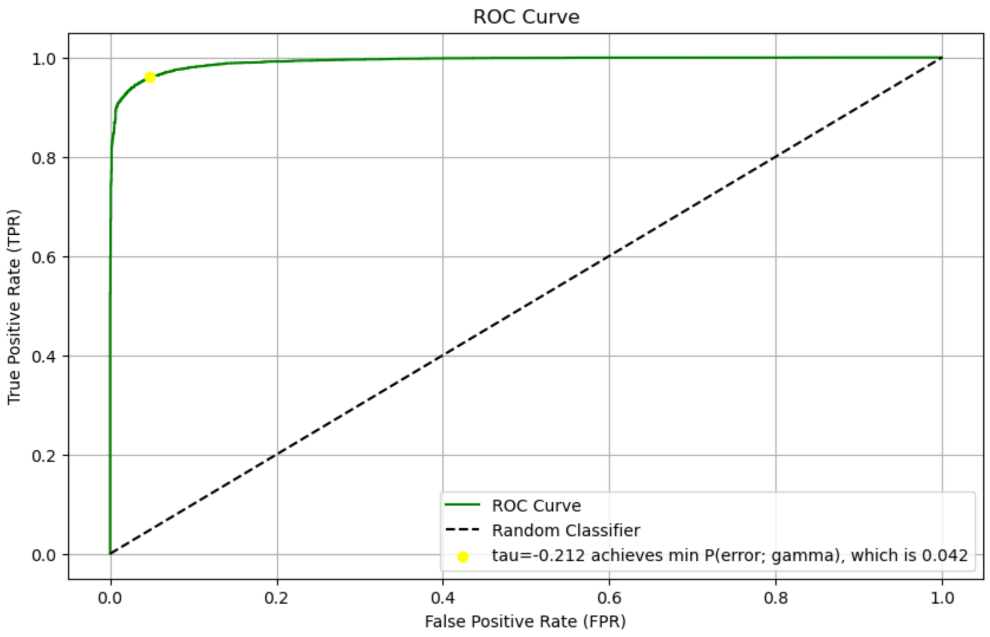


Figure 3: Approximation of the ROC curve of the minimum expected risk classifier for the LDA model

Moreover, a ROC curve of the minimum expected risk classifier for all the three considered cases has been provided. We can see that all 3 classifiers achieve almost the same minimum probability of error and the difference between their ROC curves is almost negligible.
The LDA min probability is slightly superior to the known data distribution pdfs classification, but still really close, meaning that the LDA classifier is a good option too to classify the data when the parameters of the data distribution are not known

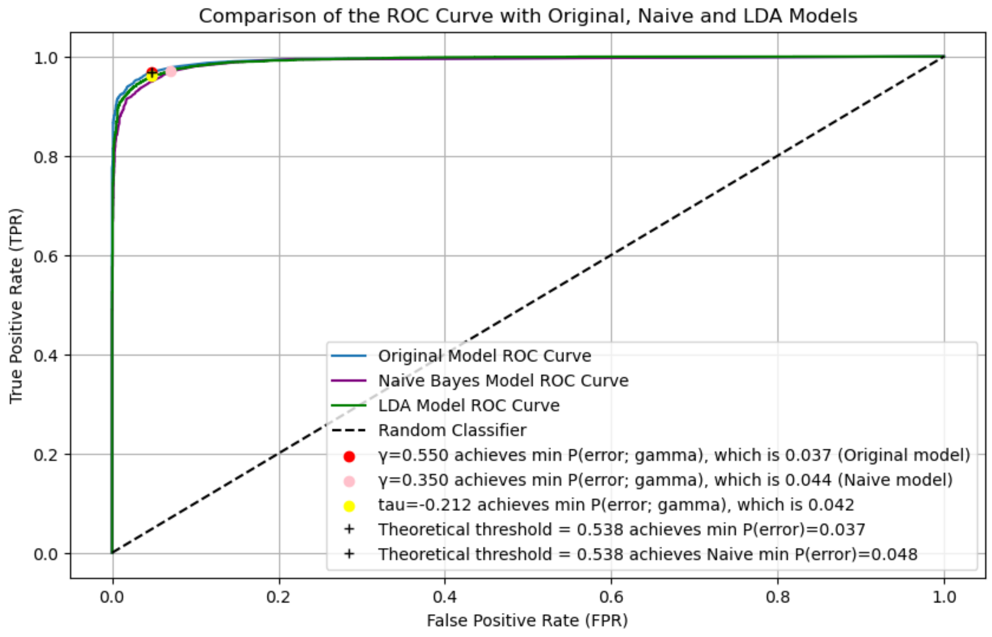
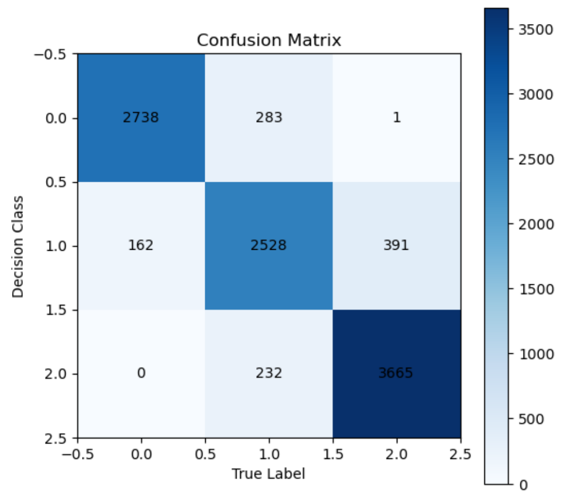


Figure 4: Approximation of the ROC curve of the minimum expected risk classifier for all models combined

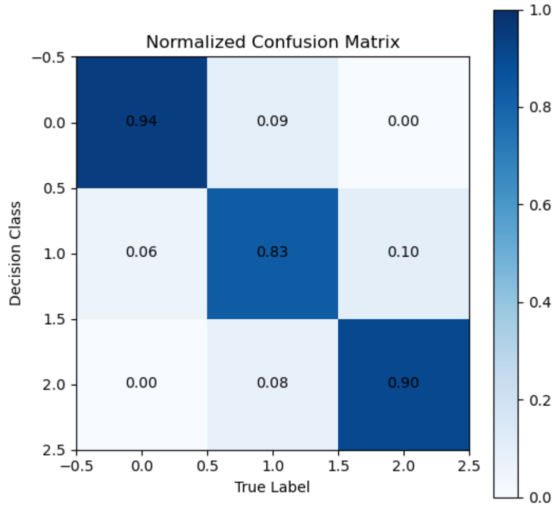
Problem 2

Part A

First of all, 10000 data samples have been generated from the data distribution detailed in Problem 2 (a 3 dimensional vector X taking values from a mixture of four Gaussians, where the first and second Gaussians correspond to the class-conditional pdf of the first 2 classes and the third and fourth Gaussians contribute by 1/2 to the class conditional pdf of the third classs). Also, we have made sure that the data was significantly overlapping.
The generated data has been classified by using the decision rule for classification that minimized the probability of error, which is the same as using the 0-1 loss. In Figure 5, the confusion matrix and normalized confusion matrix is provided for this case, where the sample counting for every decision-label pair has been computed. The majority of samples are correctly classified (samples in the diagonal), and the minimum error obtained after performing this classification has been 0.1069.
Moreover, a scatter plot in 3-dimensional space has been provided in Figure 6. Each sample is marked with a dot, circle or triangle depending on which class it belongs to, and it has been colored with green or red (correctly or incorrectly classified).



(a) 0-1 loss confusion matrix



(b) 0-1 loss normalized confusion matrix

Figure 5

Part B

Part B consists on repeating the same steps as in Part A but now using loss matrices that care 10 and 100 times more respectively about not making mistakes when the true class is 2

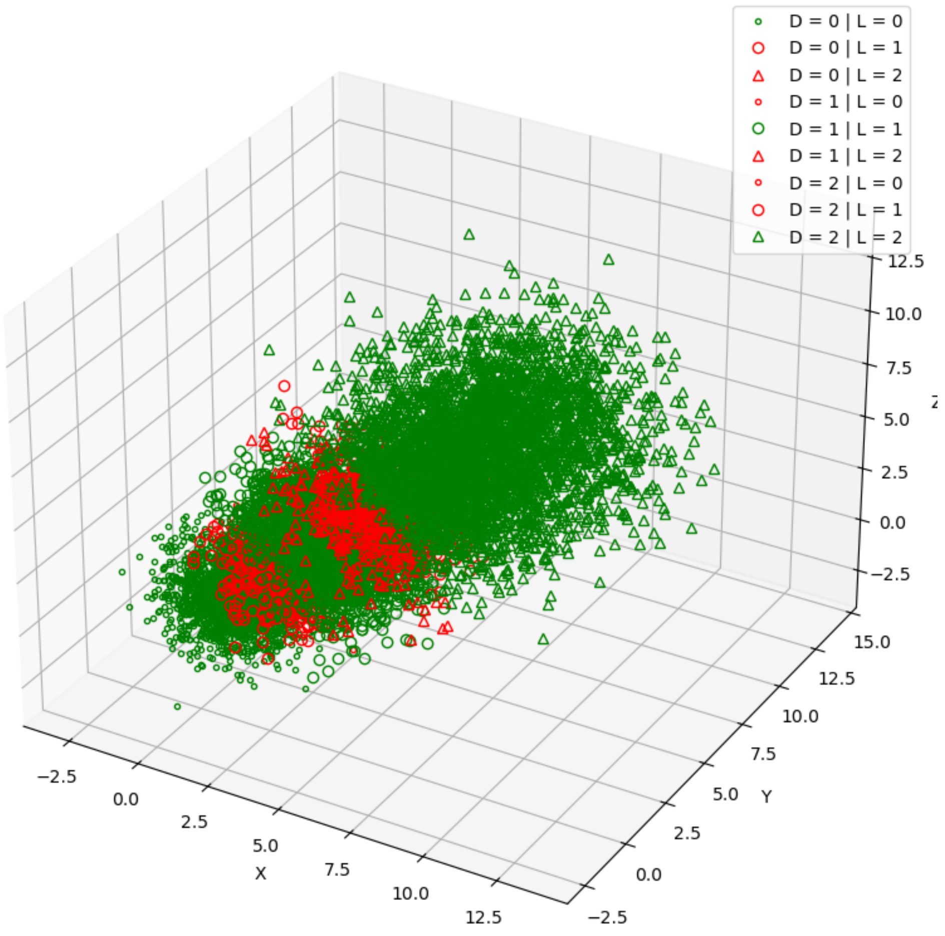


Figure 6: Scatter plot of the data indicating with green or red if the sample was correctly or incorrectly classified respectively

or 3 but is classified as 1.

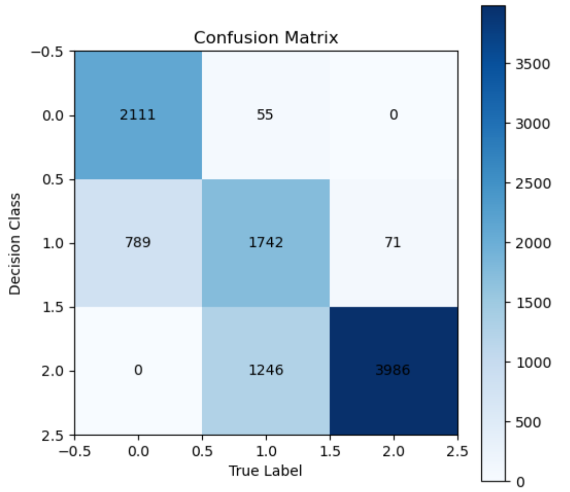
$$L_{10} = \begin{bmatrix} 0 & 10 & 10 \\ 1 & 0 & 10 \\ 1 & 1 & 0 \end{bmatrix}, \quad L_{100} = \begin{bmatrix} 0 & 100 & 100 \\ 1 & 0 & 100 \\ 1 & 1 & 0 \end{bmatrix},$$

Now, as we are not dealing with the 0-1 loss matrix anymore, instead of choosing the classifier that minimizes the probability of error, we will choose the optimal classifier for the Expected Risk Minimization rule.
All decision risks across options are computed as the product of the loss matrix and all class

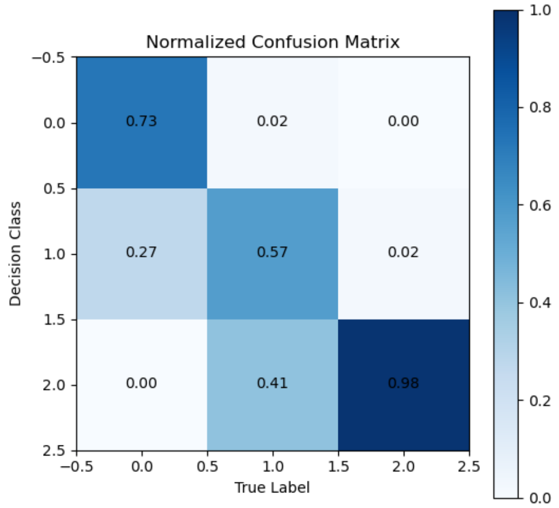
posteriors:

$$\begin{bmatrix} R(D=1 \mid x) \\ R(D=2 \mid x) \\ \vdots \\ R(D=C \mid x) \end{bmatrix} = \begin{bmatrix} \lambda_{0,0} & \lambda_{0,1} & \cdots & \lambda_{0,C} \\ \lambda_{1,0} & \lambda_{1,1} & \cdots & \lambda_{1,C} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{C,0} & \lambda_{C,1} & \cdots & \lambda_{C,C} \end{bmatrix} \begin{bmatrix} P(L=1 \mid x) \\ P(L=2 \mid x) \\ \vdots \\ P(L=C \mid x) \end{bmatrix}$$

For every sample x, the decision that has less risks is the one adopted. As the risk of every decision just depends on the sample, if we make the minimum-risk decisions for each individual x, we will end up minimizing the $\mathbb{E}_X$[Risk]. The confusion matrices and scatter plot visualization of the classified data for the $L_1 0$ and $L_1 00$ loss matrices are provided from Figure 7 to Figure 10.



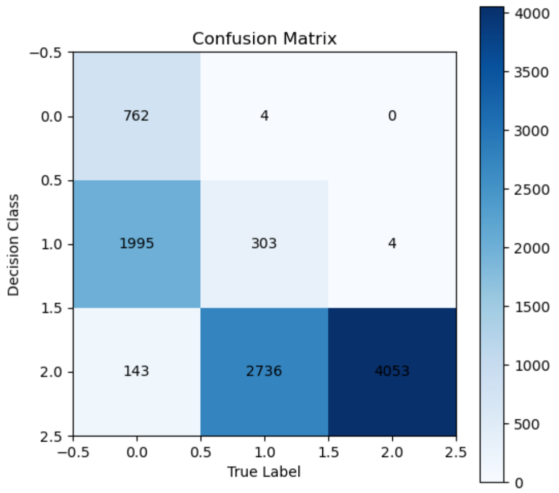
(a) 10-loss matrix confusion matrix



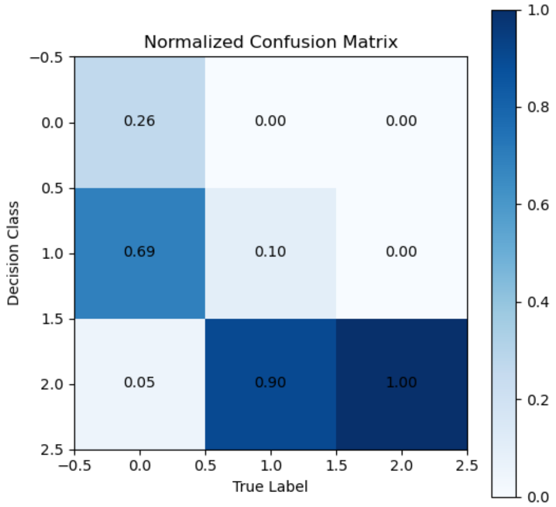
(b) 10-loss normalized confusion matrix

Figure 7

By analyzing the confusion matrices we see that when we use the 10 loss confusion matrix, we don't make as much mistakes when the true class is 2 or 3 but is classified as 1, compared to the 0-1 loss. This fact is accentuated in the results of the 100 loss matrix, which is exactly what we wanted to achieve. As a consequence, we miss-classify more samples in the lower diagonal region of the confusion matrix. The minimum expected risk achieved with the 10 loss matrix is 2.04 and with the 100 loss matrix is 48.74. We also see in Figure 10 that the amount of samples incorrectly classified is higher, meaning that not having mistakes in the cases we care more, implies having way more mistakes in the rest of cases.



(a) 100-loss matrix confusion matrix



(b) 100-loss normalized confusion matrix

Figure 8

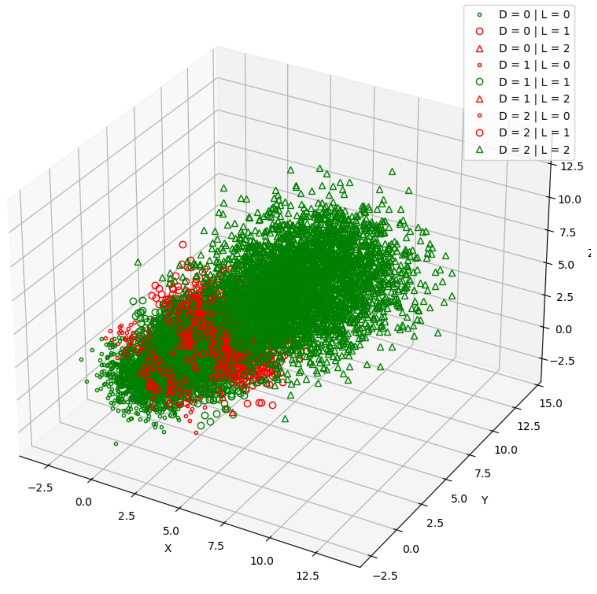


Figure 9: Scatter plot of the data (loss matrix 10) indicating with green or red if the sample was correctly or incorrectly classified respectively

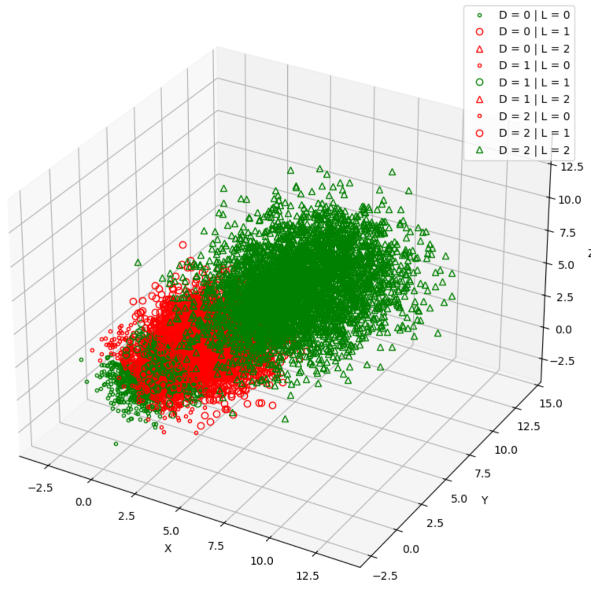


Figure 10: Scatter plot of the data (loss matrix 100) indicating with green or red if the sample was correctly or incorrectly classified respectively

Problem 3

In the last problem, the data samples are already generated. We will consider two datasets: Wine Quality Dataset and Human Activity Recognition Dataset.
The modeling assumptions that we are doing is that the class-conditional pdfs of features for each class we enconunter are multivariate Gaussians that have as mean vectors and covariance matrices the ones that we estimate from sample averaging. Given that the sample estimates for the covariance can be ill-conditioned, we have added a small regularization parameter to the covariances obtained by sample averaging:

$$\mathbf{C}_{\text{Regularized}} = \mathbf{C}_{\text{SampleAverage}} + \lambda \mathbf{I}, \quad where \quad \lambda = \alpha \frac{\text{tr}(\mathbf{C}_{\text{SampleAverage}})}{\text{rank}(\mathbf{C}_{\text{SampleAverage}})}$$

The minimum probability of error classification rule is applied to all samples and the minimum probability of error estimate and confusion matrices are reported.

Wine Quality Dataset

This dataset has 4898 samples, each with 11 features. Each sample can belong to a class ranging from 0 to 10.
In Figure 11 we see that the projection of the features Sulphates, Alcohol and pH doesn't help into clearly seeing the difference between samples belonging to different qualities. In general terms, we see the same trend for the first 3 components in the PCA analysis, as the samples belonging to different qualities seem to have a lot of overlapping.
In this case, where the Central Limit Theorem may not fully apply, the data is unlikely to follow Gaussian distribution, meaning that this assumption will introduce errors in the classification.
However, the classification that minimizes the probability of error has been performed, obtaining a 0.5 probability, which is what we would obtain with a random classifier. The confusion matrix is also provided.

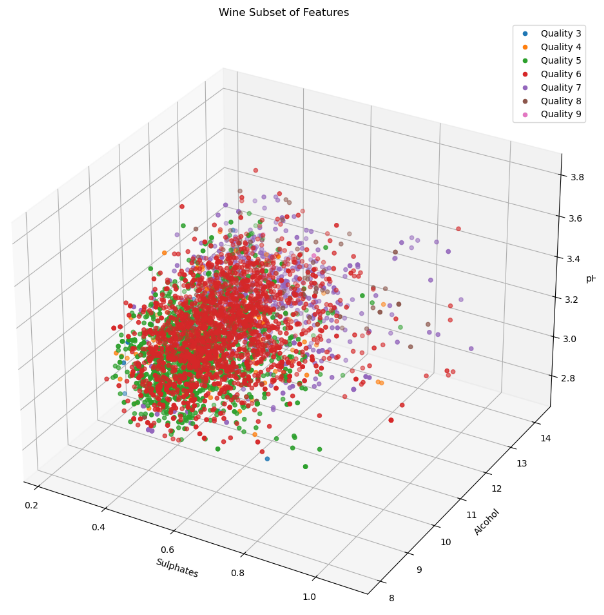


Figure 11: 3D projection of a subset of features for the Wine dataset

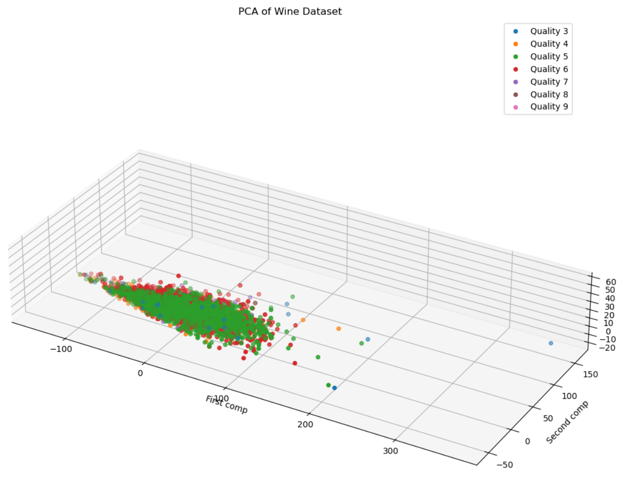


Figure 12: 3D projection of the first three principal components obtained from PCA for the Wine dataset

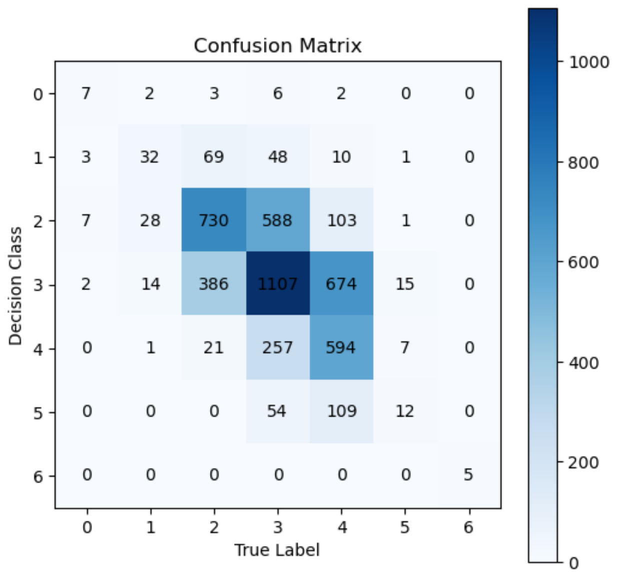


Figure 13: Confusion matrix after performing ERM with 0-1 loss error for the Wine dataset

Human Activity Recognition Dataset

This dataset has 10299 samples, each with 561 features. Each sample can belong to a class ranging from 1 to 6.

In Figure 15, we observe that some labels are well-separated into distinct clusters. This separation suggests that we may achieve better classification results compared to the wine dataset, especially considering the Gaussian assumptions we have made.
The minimum probability of error obtained has been of 0.25. In this case, the classifier achieves better results than the random classifier, as it could be predicted by observing the PCA. The confusion matrix is also provided.

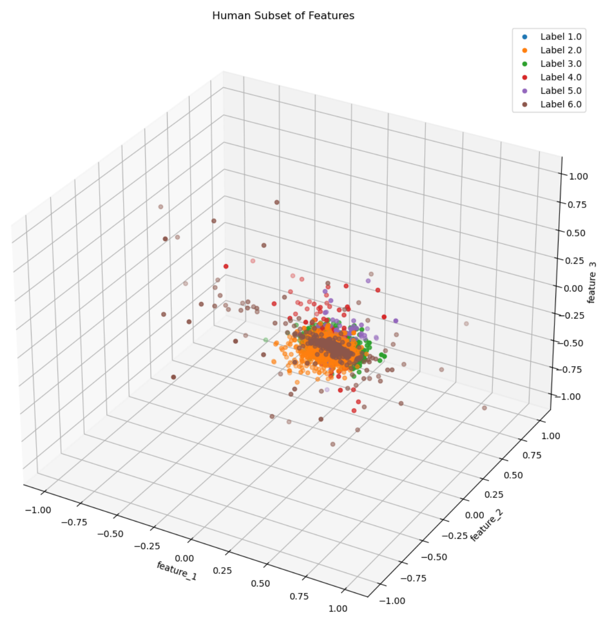


Figure 14: 3D projection of a subset of features for the Human dataset

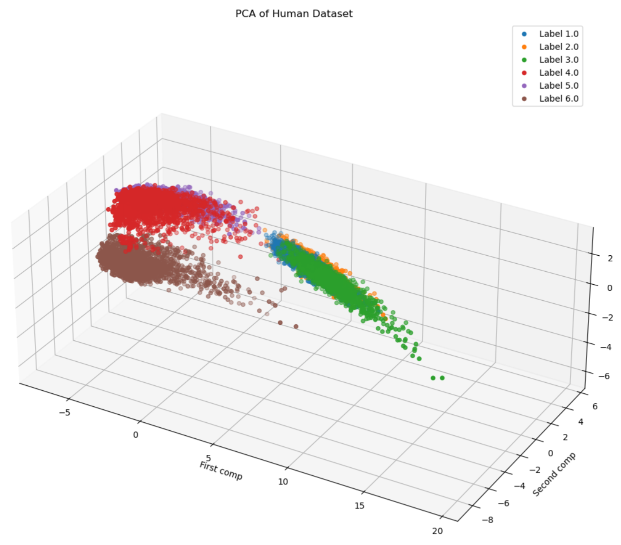
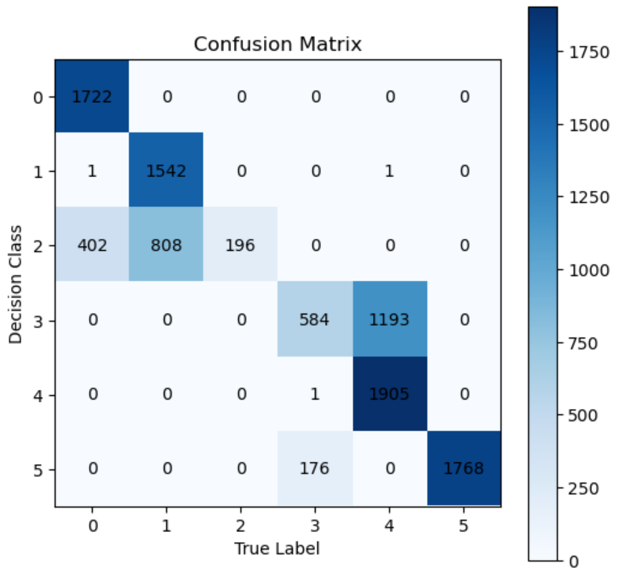


Figure 15: 3D projection of the first three principal components obtained from PCA for the Human dataset



Decision Class \ True Label	0	1	2	3	4	5
0	1722	0	0	0	0	0
1	1	1542	0	0	1	0
2	402	808	196	0	0	0
3	0	0	0	584	1193	0
4	0	0	0	1	1905	0
5	0	0	0	176	0	1768

Figure 16: Confusion matrix after performing ERM with 0-1 loss error for the Human dataset

Appendix: Code

Listing 1: Question 1 Part A

```
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import pandas as pd
from scipy.linalg import eigh

np.random.seed(99)
# constants
num_samples = 10000
num_dim = 4
num_labels = 2
priors = np.array([0.35, 0.65])

# parameters of class-conditional Gaussians pdfs
mean0 = np.array([-1,-1,-1,-1])
cov0 = np.array([[2, -0.5, 0.3, 0], [-0.5, 1, -0.5, 0], [0.3, -0.5, 1, 0], [0, 0, 0, 2]])
mean1 = np.array([1, 1, 1, 1])
cov1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])

mean = np.array([mean0, mean1])
cov = np.array([cov0, cov1])

# generate samples
class_labels = np.random.choice(a=[0,1], size=num_samples, p=priors)
x = np.zeros(shape=[num_samples, num_dim])
for i in range(num_samples):
    x[i,:] = np.random.multivariate_normal(mean[class_labels[i]], cov[class_labels[i]])

# posterior pdf
pdf_x_given_l0 = multivariate_normal.pdf(x, mean=mean[0], cov=cov[0])
pdf_x_given_l1 = multivariate_normal.pdf(x, mean=mean[1], cov=cov[1])

gamma_values = np.linspace(start=0, stop=1000, num=20000)
gamma_values = np.append(gamma_values, np.inf)

# For every gamma, we compute the TPR, FPR and probabilities of error
TPRs = []
FPRs = []
prob_errors = []

for gamma in gamma_values:
    ratio = pdf_x_given_l1/pdf_x_given_l0
    decisions = ratio > gamma # np array of 1,0 decisions

    TP = np.sum((decisions == 1) & (class_labels == 1))
    FP = np.sum((decisions == 1) & (class_labels == 0))
    FN = np.sum((decisions == 0) & (class_labels == 1))
    TN = np.sum((decisions == 0) & (class_labels == 0))

    TPR = TP/(TP+FN) # true positive detection probability
    FPR = FP/(FP+TN) # false positive probability
    FNR = FN/(FN+TP)
    prob_error = FPR * priors[0] + FNR * priors[1]

    TPRs.append(TPR)
    FPRs.append(FPR)
    prob_errors.append(prob_error)

min_prob_error = min(prob_errors)
optimal_gamma_index = prob_errors.index(min_prob_error)
optimal_gamma = gamma_values[optimal_gamma_index]

# theoretical optimal gamma
decisions_theo = (pdf_x_given_l1/pdf_x_given_l0) > priors[0]/priors[1] # np array of 1,0
    decisions

TP_theo = np.sum((decisions_theo == 1) & (class_labels == 1))
FP_theo = np.sum((decisions_theo == 1) & (class_labels == 0))
FN_theo = np.sum((decisions_theo == 0) & (class_labels == 1))
TN_theo = np.sum((decisions_theo == 0) & (class_labels == 0))

```

```
TPR_theo = TP_theo/(TP_theo+FN_theo) # true positive detection probability
FPR_theo = FP_theo/(FP_theo+TN_theo) # false positive probability
FNR_theo = FN_theo/(FN_theo+TP_theo)
prob_error_theo = FPR_theo * priors[0] + FNR_theo * priors[1]

# Plotting the ROC Curve and highlighting the optimal gamma
plt.figure(figsize=(8, 6))
plt.plot(FPRs, TPRs, label='ROC Curve', marker = '.', zorder=2)
plt.plot([0, 1], [0, 1], 'k—', label='Random Classifier') # Dashed diagonal line (random
    classifier)
plt.scatter(FPRs[optimal_gamma_index], TPRs[optimal_gamma_index], color='red', zorder=3,
    label=f' ={optimal_gamma:.3f} achieves min P(error; gamma), which is
    {min_prob_error:.3f}')
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves min P(error)={prob_error_theo:.3f}', color = 'black', zorder=4)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Listing 2: Question 1 Part B

```
# naive covariance matrices
cov0_naive = np.diag(np.diag(cov0))
cov1_naive = np.diag(np.diag(cov1))

mean = np.array([mean0, mean1])
cov_naive = np.array([cov0_naive, cov1_naive])

# posterior pdf
pdf_x_given_l0_naive = multivariate_normal.pdf(x, mean=mean[0], cov=cov_naive[0])
pdf_x_given_l1_naive = multivariate_normal.pdf(x, mean=mean[1], cov=cov_naive[1])

# For every gamma, we compute the TPR, FPR and probabilities of error
TPRs_naive = []
FPRs_naive = []
prob_errors_naive = []

for gamma in gamma_values:
    ratio = pdf_x_given_l1_naive/pdf_x_given_l0_naive
    decisions = ratio > gamma # np array of 1,0 decisions

    TP = np.sum((decisions == 1) & (class_labels == 1))
    FP = np.sum((decisions == 1) & (class_labels == 0))
    FN = np.sum((decisions == 0) & (class_labels == 1))
    TN = np.sum((decisions == 0) & (class_labels == 0))

    TPR = TP/(TP+FN) # true positive detection probability
    FPR = FP/(FP+TN) # false positive probability
    FNR = FN/(FN+TP)
    prob_error = FPR * priors[0] + FNR * priors[1]

    TPRs_naive.append(TPR)
    FPRs_naive.append(FPR)
    prob_errors_naive.append(prob_error)

min_prob_error_naive = min(prob_errors_naive)
optimal_gamma_index_naive = prob_errors_naive.index(min_prob_error_naive)
optimal_gamma_naive = gamma_values[optimal_gamma_index_naive]

# theoretical optimal gamma
decisions_naive_theo = (pdf_x_given_l1_naive/pdf_x_given_l0_naive) > priors[0]/priors[1] #
    np array of 1,0 decisions

TP_naive_theo = np.sum((decisions_naive_theo == 1) & (class_labels == 1))
FP_naive_theo = np.sum((decisions_naive_theo == 1) & (class_labels == 0))
FN_naive_theo = np.sum((decisions_naive_theo == 0) & (class_labels == 1))
TN_naive_theo = np.sum((decisions_naive_theo == 0) & (class_labels == 0))

TPR_naive_theo = TP_naive_theo/(TP_naive_theo+FN_naive_theo) # true positive detection
    probability
FPR_naive_theo = FP_naive_theo/(FP_naive_theo+TN_naive_theo) # false positive probability
FNR_naive_theo = FN_naive_theo/(FN_naive_theo+TP_naive_theo)
prob_error_naive_theo = FPR_naive_theo * priors[0] + FNR_naive_theo * priors[1]
```

```
# Plotting the ROC Curve and highlighting the optimal gamma
plt.figure(figsize=(10, 6))
plt.plot(FPRs, TPRs, label='Original Model ROC Curve', zorder=2)
plt.plot(FPRs_naive, TPRs_naive, label='Naive Bayes Model ROC Curve', zorder=2, color =
    'purple')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')  # Dashed diagonal line (random
    classifier)
plt.scatter(FPRs[optimal_gamma_index], TPRs[optimal_gamma_index], color='red', zorder=3,
    label=f'  ={optimal_gamma:.3f} achieves min P(error; gamma), which is
    {min_prob_error:.3f} (Original model)')
plt.scatter(FPRs_naive[optimal_gamma_index_naive], TPRs_naive[optimal_gamma_index_naive],
    color='pink', zorder=3, label=f'  ={optimal_gamma_naive:.3f} achieves min P(error;
    gamma), which is {min_prob_error_naive:.3f} (Naive model)')
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves min P(error)={prob_error_theo:.3f}', color = 'black', zorder=4)
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves Naive min P(error)={prob_error_naive_theo:.3f}', color = 'black', zorder=4)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Comparison of the ROC Curve with Original and Naive Models')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Listing 3: Question 1 Part C

```
# data samples x and its class labels have been generated in part A
# estimate the parameters (mean and covariance) from the generated data (average estimators)

indices_class0 = np.where(class_labels == 0)
indices_class1 = np.where(class_labels == 1)

estimated_mean0 = x[indices_class0].mean(axis=0)
estimated_mean1 = x[indices_class1].mean(axis=0)

estimated_cov0 = np.cov(x[indices_class0], rowvar=False)
estimated_cov1 = np.cov(x[indices_class1], rowvar=False)

# calculate between class scatter matrix
sb = np.outer(estimated_mean0 - estimated_mean1, estimated_mean0 - estimated_mean1)
# calculate within class scatter matrix
sw = estimated_cov0 + estimated_cov1
# Calculating eigenvalues and eigenvectors
# eigh computes the eigenvalues and eigenvectors of a matrix (or two matrices in the
    generalized case).
# It is used here to solve the generalized eigenvalue problem: sb*w = lambda*sw*w
_, eigvec = eigh(sb, sw)

# eigh returns the eigenvalues in ascending order
w_LDA = eigvec[:, -1]

lda_discriminant_scores = np.dot(x, w_LDA)
sort_scores = np.sort(lda_discriminant_scores)

#compute threshold values as mid points between scores
tau_sweep = []

for i in range(0,9999):
    tau_sweep.append((sort_scores[i] + sort_scores[i+1])/2.0)

decision_lda = []
TPRs_lda = [None] * len(tau_sweep)
FPRs_lda = [None] * len(tau_sweep)
FNRs_lda = [None] * len(tau_sweep)
prob_errors_lda = [None] * len(tau_sweep)

for (i, tau) in enumerate(tau_sweep):
    decision_lda = (lda_discriminant_scores >= tau)
    TPRs_lda[i] = (np.size(np.where((decision_lda == 1) & (class_labels ==
    1)))/np.size(np.where(class_labels == 1)))
    FPRs_lda[i] = (np.size(np.where((decision_lda == 1) & (class_labels ==
    0)))/np.size(np.where(class_labels == 0)))

    prob_errors_lda[i] = FPRs_lda[i] * priors[0] + (1 - TPRs_lda[i]) * priors[1]

```

```
# Plotting the ROC Curve
plt.figure(figsize=(10, 6))
plt.plot(FPRs_lda, TPRs_lda, label='ROC Curve', zorder=2, color = 'green')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')  # Dashed diagonal line (random
    classifier)
plt.scatter(FPRs_lda[np.argmin(prob_errors_lda)], TPRs_lda[np.argmin(prob_errors_lda)],
    color='yellow', zorder=3, label=f'tau={tau_sweep[np.argmin(prob_errors_lda)]:.3f}
    achieves min P(error; gamma), which is {np.min(prob_errors_lda):.3f}')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# Combined ROC curve
plt.figure(figsize=(10, 6))
plt.plot(FPRs, TPRs, label='Original Model ROC Curve', zorder=2)
plt.plot(FPRs_naive, TPRs_naive, label='Naive Bayes Model ROC Curve', zorder=2, color =
    'purple')
plt.plot(FPRs_lda, TPRs_lda, label='LDA Model ROC Curve', zorder=2, color = 'green')
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')  # Dashed diagonal line (random
    classifier)
plt.scatter(FPRs[optimal_gamma_index], TPRs[optimal_gamma_index], color='red', zorder=3,
    label=f'  ={optimal_gamma:.3f} achieves min P(error; gamma), which is
    {min_prob_error:.3f} (Original model)')
plt.scatter(FPRs_naive[optimal_gamma_index_naive], TPRs_naive[optimal_gamma_index_naive],
    color='pink', zorder=3, label=f'  ={optimal_gamma_naive:.3f} achieves min P(error;
    gamma), which is {min_prob_error_naive:.3f} (Naive model)')
plt.scatter(FPRs_lda[np.argmin(prob_errors_lda)], TPRs_lda[np.argmin(prob_errors_lda)],
    color='yellow', zorder=3, label=f'tau={tau_sweep[np.argmin(prob_errors_lda)]:.3f}
    achieves min P(error; gamma), which is {np.min(prob_errors_lda):.3f}')
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves min P(error)={prob_error_theo:.3f}', color = 'black', zorder=4)
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves Naive min P(error)={prob_error_naive_theo:.3f}', color = 'black', zorder=4)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Comparison of the ROC Curve with Original, Naive and LDA Models')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Listing 4: Question 2 Part A

```
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix


np.random.seed(99)

rng = np.random.default_rng()

def calculate_expected_risk(confusion_matrix, loss_matrix):
    # Since confusion_matrix is a NumPy array, you can directly use it
    total_samples = np.sum(confusion_matrix)  # No need for .values
    risk = 0
    num_classes = confusion_matrix.shape[0]  # Get the number of classes

    # Calculate expected risk based on the confusion matrix and loss matrix
    for i in range(num_classes):
        for j in range(num_classes):
            risk += (confusion_matrix[i, j] / total_samples) * loss_matrix[i, j]

    return risk

priors = np.array([0.3, 0.3, 0.4])

# mean and covariance for every gaussian distribution
gaussian_parameters = [
    (np.array([1, 1, 1]), np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])),
    (np.array([3, 3, 3]), np.array([[2, 0, 0], [0, 2, 0], [0, 0, 2]])),
    (np.array([5, 5, 5]), np.array([[3, 0, 0], [0, 3, 0], [0, 0, 3]])),
    (np.array([7, 7, 7]), np.array([[4, 0, 0], [0, 4, 0], [0, 0, 4]]))
]

num_samples = 10000
num_dim = 3
num_classes = 3
class_labels = np.random.choice(a=[0,1,2], size=num_samples, p=priors)
x = np.zeros(shape=[num_samples, num_dim])
for i in range(num_samples):
    class_label = class_labels[i]
    if class_label in range(2):
        mean, cov = gaussian_parameters[class_label]
        x[i,:] = np.random.multivariate_normal(mean, cov)
    else:
        gaussian_choice = rng.choice([2, 3]) # select either the 3rd or 4th gaussian with
    0.5 prob each
        mean, cov = gaussian_parameters[gaussian_choice]
        x[i,:] = np.random.multivariate_normal(mean, cov)

posteriors = np.zeros((num_samples, num_classes))

for class_label in range(num_classes):
    if class_label in range(2):
        mean, cov = gaussian_parameters[class_label]
        posteriors[:, class_label] = multivariate_normal.pdf(x, mean, cov) *
    priors[class_label]
    else:
        posteriors[:, class_label] = 0.5 * (multivariate_normal.pdf(x,
    gaussian_parameters[2][0], gaussian_parameters[2][1]) +
                                              multivariate_normal.pdf(x,
    gaussian_parameters[3][0], gaussian_parameters[3][1])) * priors[class_label]



# Bayes classifier with 0-1 loss (minimum probability of error classification rule)
# The decision rule is the argmax of the posterior
decisions = np.argmax(posteriors, axis = 1)

# Create the confusion matrix (prediced_labels are decisions here)
conf_matrix = confusion_matrix(class_labels, decisions).T

# Plotting the confusion matrix
plt.figure(figsize=(6, 6))
plt.imshow(conf_matrix, cmap='Blues')

plt.title("Confusion Matrix")
plt.xlabel('True Label')
plt.ylabel('Decision Class')
plt.colorbar()

for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        plt.text(j, i, conf_matrix[i, j], ha='center', va='center', color='black')

plt.show()


# Normalize the confusion matrix by row (i.e., by the number of actual instances per class)
col_sums = conf_matrix.sum(axis=0, keepdims=True)
normalized_decision_conf_matrix = conf_matrix / col_sums

# Plot the normalized confusion matrix
plt.figure(figsize=(6, 6))
plt.imshow(normalized_decision_conf_matrix, cmap='Blues', vmin=0, vmax=1)

plt.title("Normalized Confusion Matrix")
plt.xlabel('True Label')
plt.ylabel('Decision Class')
plt.colorbar()

for i in range(normalized_decision_conf_matrix.shape[0]):
    for j in range(normalized_decision_conf_matrix.shape[1]):
        plt.text(j, i, f'{normalized_decision_conf_matrix[i, j]:.2f}',
                 ha='center', va='center', color='black')

# Display the plot
```

```
plt.show()


# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.set_facecolor('white')
fig.patch.set_facecolor('white')

# Set of marker shapes
marker_shapes = '.o^'

# Loop through each class and decision to plot
for d in range(num_classes):
    for c in range(num_classes):
        indices = np.where((decisions == d) & (class_labels == c))
        points = x[indices]
        color = 'g' if d == c else 'r'
        label = f"D = {d} | L = {c}"
        ax.plot(points[:, 0], points[:, 1], points[:, 2], marker=marker_shapes[c],
    color=color,
                markerfacecolor='none', linestyle='None', label=label)

# Adjust padding for axis labels
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Add legend and show plot
ax.legend()
plt.show()

loss_matrix_1 = np.array([[0, 1, 1],
                          [1, 0, 1],
                          [1, 1, 0]])
expected_risk_1 = calculate_expected_risk(conf_matrix.T, loss_matrix_1)

print(f"Minimum Expected Risk: {expected_risk_1}")
```

Listing 5: Question 2 Part B

```
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix


np.random.seed(99)

rng = np.random.default_rng()

priors = np.array([0.3, 0.3, 0.4])

# mean and covariance for every gaussian distribution
gaussian_parameters = [
    (np.array([1, 1, 1]), np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])),
    (np.array([3, 3, 3]), np.array([[2, 0, 0], [0, 2, 0], [0, 0, 2]])),
    (np.array([5, 5, 5]), np.array([[3, 0, 0], [0, 3, 0], [0, 0, 3]])),
    (np.array([7, 7, 7]), np.array([[4, 0, 0], [0, 4, 0], [0, 0, 4]]))
]

num_samples = 10000
num_dim = 3
num_classes = 3
class_labels = np.random.choice(a=[0,1,2], size=num_samples, p=priors)
x = np.zeros(shape=[num_samples, num_dim])
for i in range(num_samples):
    class_label = class_labels[i]
    if class_label in range(2):
        mean, cov = gaussian_parameters[class_label]
        x[i,:] = np.random.multivariate_normal(mean, cov)
    else:
        gaussian_choice = rng.choice([2, 3]) # select either the 3rd or 4th gaussian with
    0.5 prob each
        mean, cov = gaussian_parameters[gaussian_choice]
        x[i,:] = np.random.multivariate_normal(mean, cov)
```

```
posteriors = np.zeros((num_samples, num_classes))

for class_label in range(num_classes):
    if class_label in range(2):
        mean, cov = gaussian_parameters[class_label]
        posteriors[:, class_label] = multivariate_normal.pdf(x, mean, cov) *
    priors[class_label]
    else:
        posteriors[:, class_label] = 0.5 * (multivariate_normal.pdf(x,
    gaussian_parameters[2][0], gaussian_parameters[2][1]) +
                                            multivariate_normal.pdf(x,
    gaussian_parameters[3][0], gaussian_parameters[3][1])) * priors[class_label]


loss_matrix_10 = np.array([[0, 10, 10],
                           [1, 0, 10],
                           [1, 1, 0]])

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

loss_matrices = [loss_matrix_10, loss_matrix_100]

for loss_matrix in loss_matrices:

    # Risks are calculated by multiplying the posterior probabilities with the transpose of
    the loss matrix
    # ERM is computed by determining the class with the minimum risk for each sample
    risks = np.dot(posteriors, loss_matrix.T)
    decisions = np.argmin(risks, axis=1)

    # Create the confusion matrix (prediced_labels are decisions here)
    conf_matrix = confusion_matrix(class_labels, decisions).T

    # Plotting the confusion matrix
    plt.figure(figsize=(6, 6))
    plt.imshow(conf_matrix, cmap='Blues')

    plt.title("Confusion Matrix")
    plt.xlabel('True Label')
    plt.ylabel('Decision Class')
    plt.colorbar()

    for i in range(conf_matrix.shape[0]):
        for j in range(conf_matrix.shape[1]):
            plt.text(j, i, conf_matrix[i, j], ha='center', va='center', color='black')

    plt.show()


    # Normalize the confusion matrix by row (i.e., by the number of actual instances per
    class)
    col_sums = conf_matrix.sum(axis=0, keepdims=True)
    normalized_decision_conf_matrix = conf_matrix / col_sums

    # Plot the normalized confusion matrix
    plt.figure(figsize=(6, 6))
    plt.imshow(normalized_decision_conf_matrix, cmap='Blues', vmin=0, vmax=1)

    plt.title("Normalized Confusion Matrix")
    plt.xlabel('True Label')
    plt.ylabel('Decision Class')
    plt.colorbar()

    for i in range(normalized_decision_conf_matrix.shape[0]):
        for j in range(normalized_decision_conf_matrix.shape[1]):
            plt.text(j, i, f'{normalized_decision_conf_matrix[i, j]:.2f}',
                     ha='center', va='center', color='black')

    # Display the plot
    plt.show()

    # Create a 3D scatter plot
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_facecolor('white')
    fig.patch.set_facecolor('white')

    # Set of marker shapes
    marker_shapes = '.o^'

    # Loop through each class and decision to plot
    for d in range(num_classes):
        for c in range(num_classes):
            indices = np.where((decisions == d) & (class_labels == c))
            points = x[indices]
            color = 'g' if d == c else 'r'
            label = f"D = {d} | L = {c}"
            ax.plot(points[:, 0], points[:, 1], points[:, 2], marker=marker_shapes[c],
    color=color,
                    markerfacecolor='none', linestyle='None', label=label)

    # Adjust padding for axis labels
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    # Add legend and show plot
    ax.legend()
    plt.show()

    expected_risk = calculate_expected_risk(conf_matrix.T, loss_matrix)

    print(f"Minimum Expected Risk: {expected_risk}")
```

Listing 6: Question 3 Functions

```
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
from sklearn.decomposition import PCA
from numpy.linalg import matrix_rank

np.random.seed(7)
np.set_printoptions(suppress=True) # it suppresses the scientific notation for small numbers

def regularized_cov(X, alfa):
    """
    Computes the regularized covariance matrix (useful in case there are ill-conditioned
    matrices)

    Parameters:
    - X: data (features x samples)
    - alfa: reg parameter (lambda = alfa * trace(C_sampleAverage)/rank(C_sampleAverage))

    Returns:
    - Reg cov matrix
    """

    # rowvar=False: If False, each column represents a feature, and the covariance is
    calculated across the rows (observations)
    C_sampleAverage = np.cov(X, rowvar = False)
    num_features = C_sampleAverage.shape[0]
    lambda_reg = alfa * np.trace(C_sampleAverage)/num_features
    C_regularized = C_sampleAverage + lambda_reg * np.eye(num_features)
    return C_regularized

def erm_classification(X, loss_matrix, priors, mean, cov, num_classes):
    """
    Performs Expected Risk Minimization classification to minimize the probability of error
    (when the number of decisions is equal to the number of class labels)

    Parameters:
    - X: data (features x samples)
    - loss_matrix (d,l) (loss incurred by deciding d given a sample from class/label l)
    - priors, mean, cov (gaussian multivariate parameters)
    - num_classes
    Returns:
    - The predicted/ decided class labels for every sample to minimize the probability of
    error
    """

    # compute class conditionals (likelihoods) and posteriors (num_classes x num_samples)
    class_conditional = np.array([multivariate_normal.pdf(X, mean[c], cov[c]) for c in
    range(num_classes)])
    print(class_conditional)

    posteriors = np.array([class_conditional[i] * priors[i] for i in range(num_classes)])
    # risk matrix (num_classes x num_samples)
    risk_matrix = loss_matrix @ posteriors

    return np.argmin(risk_matrix, axis=0)
```

Listing 7: Question 3 Wine Dataset

```
wine_df = pd.read_csv('dataset_question_3/wine/winequality-white.csv',
                        delimiter=';')

# extract data (features) and labels (last column)
X = wine_df.iloc[:, :-1].to_numpy()
labels = wine_df.iloc[:, -1].to_numpy()

# Encode labels using LabelEncoder
# https://scikit-learn.org/dev/modules/generated/sklearn.preprocessing.LabelEncoder.html
le = preprocessing.LabelEncoder()
labels = le.fit_transform(labels)

# class means, covariance matrix and priors estimates (with sample averages)
class_means = wine_df.groupby('quality').mean().to_numpy()
num_samples = wine_df.shape[0]
alfa = 0.00001
class_priors = wine_df.groupby('quality').size().to_numpy() / num_samples
num_classes = len(class_priors)
class_reg_cov_mat = np.array([regularized_cov(X[labels == l], alfa) for l in
    range(num_classes)])

# cost matrix for 0-1 loss
loss_matrix = np.ones((num_classes, num_classes)) - np.eye(num_classes)

# perform Expected Risk Minimization classification to minimize the probability of error
# and get the decision/predicted class labels for every sample
predicted_labels = erm_classification(X, loss_matrix, class_priors, class_means,
    class_reg_cov_mat, num_classes)

# compute confusion matrix
conf_matrix = confusion_matrix(predicted_labels, labels).T

# Plotting the confusion matrix
plt.figure(figsize=(6, 6))
plt.imshow(conf_matrix, cmap='Blues')

plt.title("Confusion Matrix")
plt.xlabel('True Label')
plt.ylabel('Decision Class')
plt.colorbar()

for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        plt.text(j, i, conf_matrix[i, j], ha='center', va='center', color='black')

plt.show()

num_errors = num_samples - np.sum(np.diag(conf_matrix))
print("Error sample count: {:d}".format(num_errors))

prob_error_estimate = num_errors/num_samples
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error_estimate))

# Visualization of the dataset in 3 dimensional chosen projections
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

```

```
unique_qualities = np.sort(wine_df['quality'].unique())

# Plot each quality level
for q in unique_qualities:
    quality_data = wine_df[wine_df['quality'] == q]
    ax.scatter(quality_data['sulphates'], quality_data['alcohol'], quality_data['pH'],
    label=f"Quality {q}")

plt.title("Wine Subset of Features")
plt.legend()
plt.tight_layout() # adjusts margins, padding, and spacing between subplots to make sure
    everything fits properly within the figure.

ax.set_xlabel("Sulphates")
ax.set_ylabel("Alcohol")
ax.set_zlabel("pH")

plt.show()

# Create a figure and a 3D subplot
fig = plt.figure(figsize=(10, 10))
ax_pca = fig.add_subplot(111, projection='3d')


# Visualization of the dataset in the 3 first components coming from the PCA
pca = PCA(n_components=3)  # we want the first 3 principal components
X_fit = pca.fit(X)
Z = pca.transform(X)  # Project the data into the principal component 3D space

print("Explained variance ratio: ", pca.explained_variance_ratio_)

unique_qualities = np.sort(wine_df['quality'].unique())

# Plot each quality level
for q in unique_qualities:
    quality_indices = wine_df['quality'] == q
    ax_pca.scatter(Z[quality_indices, 0], Z[quality_indices, 1], Z[quality_indices, 2],
    label=f"Quality {q}")

# Set equal aspect ratio for the 3D plot
ax_pca.set_box_aspect((np.ptp(Z[:, 0]), np.ptp(Z[:, 1]), np.ptp(Z[:, 2])))

plt.title("PCA of Wine Dataset")
plt.legend()
plt.tight_layout()

ax_pca.set_xlabel("First comp")
ax_pca.set_ylabel("Second comp")
ax_pca.set_zlabel("Third comp")

plt.show()
```

Listing 8: Question 3 Human Dataset

```
X_train = np.loadtxt('dataset_question_3/human/X_train.txt')
X_test = np.loadtxt('dataset_question_3/human/X_test.txt')
y_train = np.loadtxt('dataset_question_3/human/y_train.txt')
y_test = np.loadtxt('dataset_question_3/human/y_test.txt')

# Convert NumPy arrays to pandas DataFrames
X_train_df = pd.DataFrame(X_train)
X_test_df = pd.DataFrame(X_test)
y_train_df = pd.DataFrame(y_train)
y_test_df = pd.DataFrame(y_test)
# Concatenate the training and test dataframes for X (features) and labels
X = pd.concat([X_train_df, X_test_df], ignore_index=True)
labels = pd.concat([y_train_df, y_test_df], ignore_index=True)
# Create headers for X (features) and labels
num_features = X.shape[1]  # Number of columns in X
feature_headers = [f'feature_{i+1}' for i in range(num_features)]
label_header = ['label']

# Assign the headers to X and labels
X.columns = feature_headers
labels.columns = label_header

```

```python
# Combine X (features) and labels into one dataframe, with labels as the last column
human_df = pd.concat([X, labels], axis=1)

# extract data (features) and labels (last column)
X = human_df.iloc[:, :-1].to_numpy()
labels = human_df.iloc[:, -1].to_numpy()

# Encode labels using LabelEncoder
# https://scikit-learn.org/dev/modules/generated/sklearn.preprocessing.LabelEncoder.html
le = preprocessing.LabelEncoder()
labels = le.fit_transform(labels)
unique_labels = np.unique(labels)

# class means, covariance matrix and priors estimates (with sample averages)
class_means = human_df.groupby('label').mean().to_numpy()
num_samples = human_df.shape[0]
alfa = 0.9
class_priors = human_df.groupby('label').size().to_numpy() / num_samples
num_classes = len(class_priors)

class_reg_cov_mat = np.array([regularized_cov(X[labels == l], alfa) for l in unique_labels])
print(class_reg_cov_mat)
# cost matrix for 0-1 loss
loss_matrix = np.ones((num_classes, num_classes)) - np.eye(num_classes)

# perform Expected Risk Minimization classification to minimize the probability of error
# and get the decision/predicted class labels for every sample
predicted_labels = erm_classification(X, loss_matrix, class_priors, class_means,
    class_reg_cov_mat, num_classes)

# compute confusion matrix
conf_matrix = confusion_matrix(predicted_labels, labels).T

# Plotting the confusion matrix
plt.figure(figsize=(6, 6))
plt.imshow(conf_matrix, cmap='Blues')

plt.title("Confusion Matrix")
plt.xlabel('True Label')
plt.ylabel('Decision Class')
plt.colorbar()

for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        plt.text(j, i, conf_matrix[i, j], ha='center', va='center', color='black')

plt.show()

num_errors = num_samples - np.sum(np.diag(conf_matrix))
print("Error sample count: {:d}".format(num_errors))

prob_error_estimate = num_errors/num_samples
print("Empirically Estimated Probability of Error: {:.4f}".format(prob_error_estimate))

# Visualization of the dataset in 3 dimensional chosen projections
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

unique_labels = np.sort(human_df['label'].unique())

# Plot each label level
for q in unique_labels:
    label_data = human_df[human_df['label'] == q]
    ax.scatter(label_data['feature_1'], label_data['feature_2'], label_data['feature_3'],
    label=f"Label {q}")

plt.title("Human Subset of Features")
plt.legend()
plt.tight_layout() # adjusts margins, padding, and spacing between subplots to make sure
    everything fits properly within the figure.

ax.set_xlabel("feature_1")
ax.set_ylabel("feature_2")
ax.set_zlabel("feature_3")

plt.show()
```

```
# Create a figure and a 3D subplot
fig = plt.figure(figsize=(10, 10))
ax_pca = fig.add_subplot(111, projection='3d')


# Visualization of the dataset in the 3 first components coming from the PCA
pca = PCA(n_components=3)  # we want the first 3 principal components
X_fit = pca.fit(X)
Z = pca.transform(X)  # Project the data into the principal component 3D space

print("Explained variance ratio: ", pca.explained_variance_ratio_)

# Plot each label level
for q in unique_labels:
    label_indices = human_df['label'] == q
    ax_pca.scatter(Z[label_indices, 0], Z[label_indices, 1], Z[label_indices, 2],
    label=f"Label {q}")

# Set equal aspect ratio for the 3D plot
ax_pca.set_box_aspect((np.ptp(Z[:, 0]), np.ptp(Z[:, 1]), np.ptp(Z[:, 2])))

plt.title("PCA of Human Dataset")
plt.legend()
plt.tight_layout()

ax_pca.set_xlabel("First comp")
ax_pca.set_ylabel("Second comp")
ax_pca.set_zlabel("Third comp")

plt.show()
```