p1

June 1, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
```

[2]:
```
# set up the parameters of the class-conditioned Gaussian pdfs
mu_0 = np.array([-1, 1, -1, 1])
mu_1 = np.array([1, 1, 1, 1])

sigma_0 = np.array([[2, -0.5, 0.3, 0],
                    [-0.5, 1, -0.5, 0],
                    [0.3, -0.5, 1, 0],
                    [0, 0, 0, 2]])
sigma_1 = np.array([[1, 0.3, -0.2, 0],
                    [0.3, 2, 0.3, 0],
                    [-0.2, 0.3, 1, 0],
                    [0, 0, 0, 3]])

# set up the parameters of the class priors
p_0 = 0.7
p_1 = 0.3
```

[3]:
```
# generate 10000 samples according to the data distribution
samples = 10000
x = np.random.multivariate_normal(mu_0, sigma_0, samples)
y = np.random.multivariate_normal(mu_1, sigma_1, samples)

# separate our final data set into data and labels
data = []
labels = []

# use the class priors to generate the labels
for i in range(samples):
    if np.random.rand() < p_0:
        data.append(x[i])
        labels.append(0)
    else:
        data.append(y[i])
        labels.append(1)
```

```
# convert the data and labels to numpy arrays
data = np.array(data)
labels = np.array(labels)
```

[4]:
```
# set up the parameters of the loss matrix, using 0-1 loss
loss_matrix = np.array([[0, 1], [1, 0]])
```

0.0.1 Part A

1. Specify the minimum expected risk classification rule in the form of a likelihood ratio test where the threshold is a function of class priors and fixed loss values for each of the four possible outcomes

[10]:
```
# return the ratio f_{X|Y}(x|1) / f_{X|Y}(x|0)
def likelihood_ratio_test(x: np.array, mu_0: np.array, mu_1: np.array, sigma_0:␣
↪np.array, sigma_1: np.array) -> np.array:
    """
    Given a sample x, return the ratio f_{X|Y}(x|1) / f_{X|Y}(x|0)

    Args:
        x (np.array): a sample or samples from the data distribution
        mu_0 (np.array): the mean of the class 0 Gaussian pdf
        mu_1 (np.array): the mean of the class 1 Gaussian pdf
        sigma_0 (np.array): the covariance matrix of the class 0 Gaussian pdf
        sigma_1 (np.array): the covariance matrix of the class 1 Gaussian pdf

    Returns:
        np.array: the ratio f_{X|Y}(x|1) / f_{X|Y}(x|0)
    """
    ratios_x_0 = multivariate_normal.pdf(x, mean=mu_0, cov=sigma_0)
    ratios_x_1 = multivariate_normal.pdf(x, mean=mu_1, cov=sigma_1)
    return ratios_x_1 / ratios_x_0
```

[11]:
```
# define the minimum expected risk classification rule
def minimum_expected_risk_classification_rule(x: np.array, threshold: float,␣
↪mu_0: np.array, mu_1: np.array, sigma_0: np.array, sigma_1: np.array) -> np.
↪array:
    """
    Given a sample x and a threshold, return the classification of x according␣
↪to the minimum expected risk classification rule

    Args:
        x (np.array): a sample or samples from the data distribution
        mu_0 (np.array): the mean of the class 0 Gaussian pdf
        mu_1 (np.array): the mean of the class 1 Gaussian pdf
        sigma_0 (np.array): the covariance matrix of the class 0 Gaussian pdf
        sigma_1 (np.array): the covariance matrix of the class 1 Gaussian pdf
```

```
        threshold (float): the threshold for the likelihood ratio test

    Returns:
        np.array: the classification of x according to the minimum expected␣
↪risk classification rule. 0 for class 0, 1 for class 1
    """
    ratios = likelihood_ratio_test(x, mu_0, mu_1, sigma_0, sigma_1)
    classifications = np.where(ratios >= threshold, 1, 0)
    return classifications
```

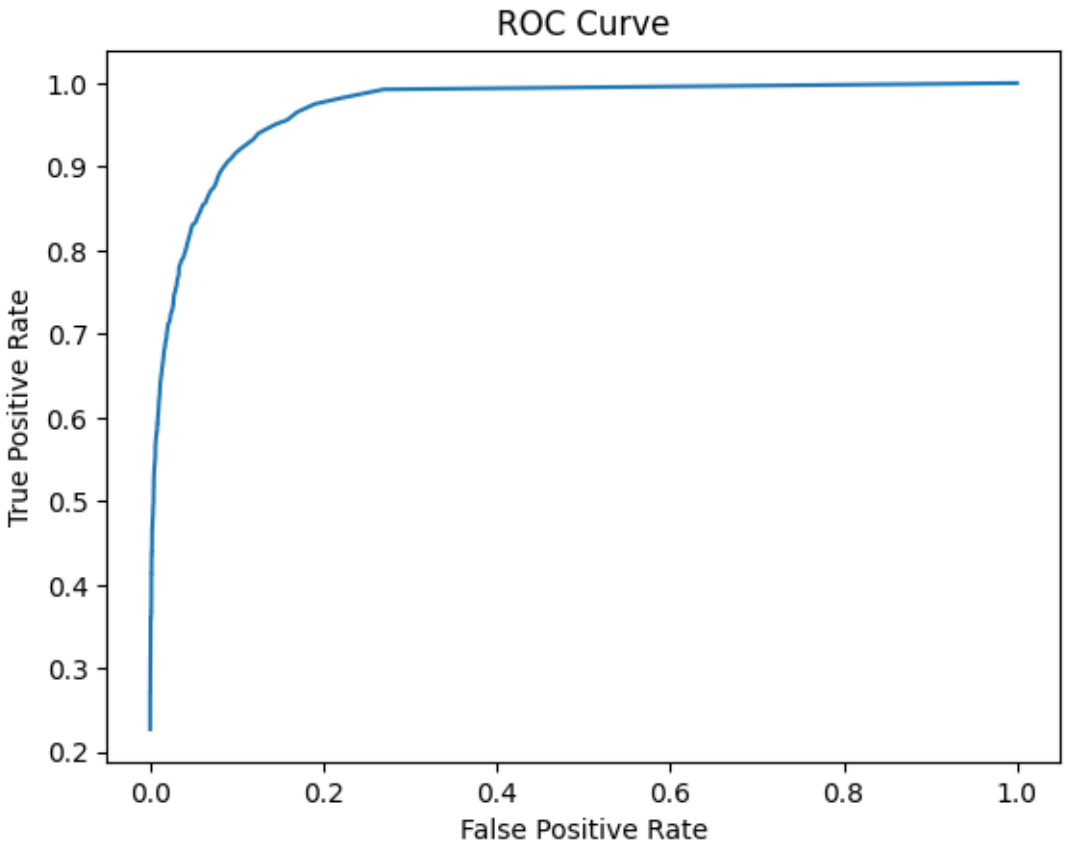
2. Implement the classifier and apply it to the data set. Vary the threshold gradually from 0 to infinity and for each value of the threshold, compute the true positive rate and the false positive probabilities. Using these paired values, plot the ROC curve.

[12]:
```
# obtain the true positive rate and false positive rate for each threshold for␣
↪the ROC curve
def ROC_curve(data: np.array, labels: np.array, thresholds: np.array, mu_0: np.
↪array = mu_0, mu_1: np.array = mu_1, sigma_0: np.array = sigma_0, sigma_1:␣
↪np.array = sigma_1) -> tuple[np.array, np.array]:
    """
    Given data, labels, and thresholds, return the true positive rates and␣
↪false positive rates for the ROC curve

    Args:
        data (np.array): the data set
        labels (np.array): the labels for the data set
        thresholds (np.array): the thresholds for the likelihood ratio test
        mu_0 (np.array, optional): the mean of the class 0 Gaussian pdf.␣
↪Defaults to mu_0.
        mu_1 (np.array, optional): the mean of the class 1 Gaussian pdf.␣
↪Defaults to mu_1.
        sigma_0 (np.array, optional): the covariance matrix of the class 0␣
↪Gaussian pdf. Defaults to sigma_0.
        sigma_1 (np.array, optional): the covariance matrix of the class 1␣
↪Gaussian pdf. Defaults to sigma_1.

    Returns:
        tuple[np.array, np.array]: the true positive rates and false positive␣
↪rates for the ROC curve
    """
    # compute the predicted labels for all thresholds and data points at once
    predictions = np.array([minimum_expected_risk_classification_rule(data,␣
↪threshold, mu_0, mu_1, sigma_0, sigma_1) for threshold in thresholds])

    # calculate the number of true positives, false positives, and total␣
↪positives for each threshold
    true_positives = np.sum((predictions == 1) & (labels == 1), axis=1)
```

```
    false_positives = np.sum((predictions == 1) & (labels == 0), axis=1)
    total_positives = np.sum(labels == 1)

    # compute the true positive rates and false positive rates
    true_positive_rates = true_positives / total_positives
    false_positive_rates = false_positives / np.sum(labels == 0)

    return true_positive_rates, false_positive_rates
```

[17]:
```
# have some constants for the ROC curve loop
thresholds = np.linspace(0, 1000, 10000, endpoint=False)
```

[19]:
```
# plot the ROC curve
true_positive_rate, false_positive_rate = ROC_curve(data, labels, thresholds)

plt.plot(false_positive_rate, true_positive_rate)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

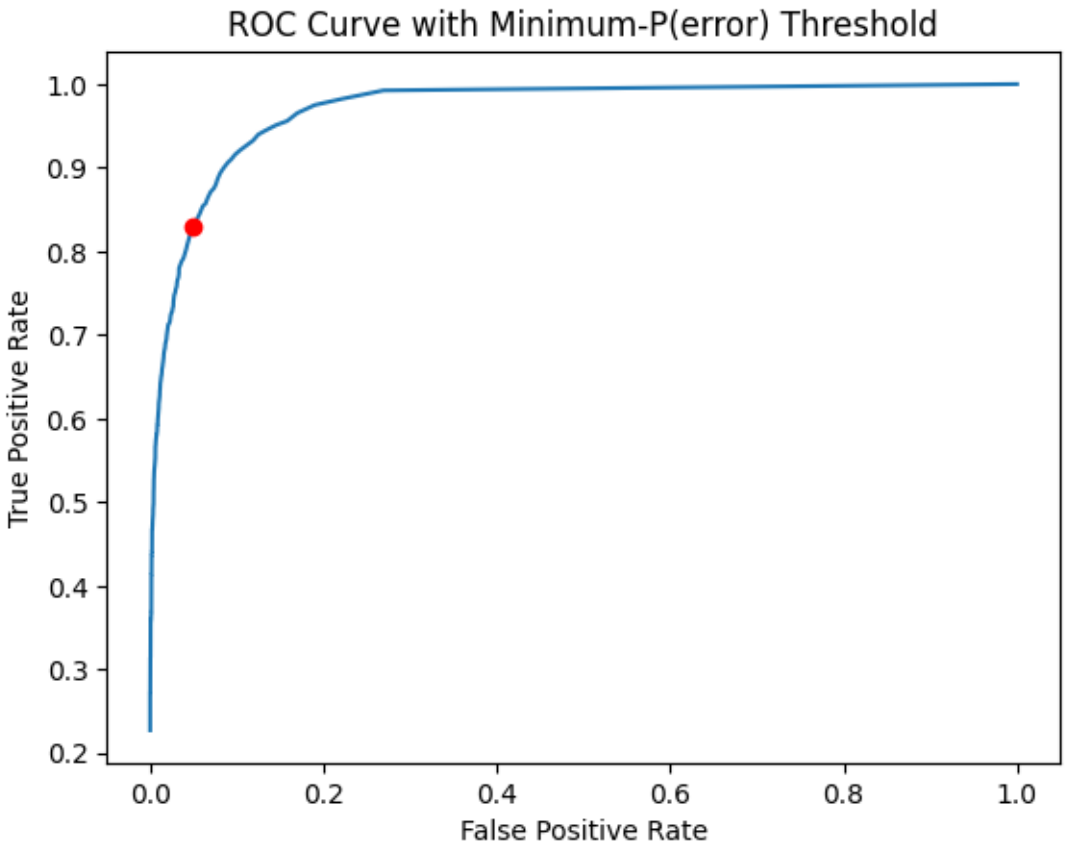


3. Determine the threshold that minimizes the probability of error. On the ROC curve, superimpose the true positive and false positive probabilities for this minimum-P(error) threshold. Calculate and report an estimate of the minimum probability of error achievable for this data distribution

[21]:
```
# obtain the probability of error for each threshold that is only based on the␣
↪data and not priors
def probability_of_error(data: np.array, labels: np.array, thresholds: np.
↪array, mu_0: np.array = mu_0, mu_1: np.array = mu_1, sigma_0: np.array =␣
↪sigma_0, sigma_1: np.array = sigma_1) -> np.array:
    """
    Given data, labels, and a threshold, return the probability of error

    Args:
        data (np.array): the data set
        labels (np.array): the labels for the data set
        thresholds (np.array): the thresholds for the likelihood ratio test
        mu_0 (np.array, optional): the mean of the class 0 Gaussian pdf.␣
↪Defaults to mu_0.
        mu_1 (np.array, optional): the mean of the class 1 Gaussian pdf.␣
↪Defaults to mu_1.
        sigma_0 (np.array, optional): the covariance matrix of the class 0␣
↪Gaussian pdf. Defaults to sigma_0.
        sigma_1 (np.array, optional): the covariance matrix of the class 1␣
↪Gaussian pdf. Defaults to sigma_1.

    Returns:
        np.array: the probability of error for each threshold
    """
    # compute the predicted labels for all thresholds and data points at once
    predictions = np.array([minimum_expected_risk_classification_rule(data,␣
↪threshold, mu_0, mu_1, sigma_0, sigma_1) for threshold in thresholds])

    # calculate the number of errors for each threshold
    errors = np.sum(predictions != labels, axis=1)

    # compute the probability of error
    p_error = errors / len(data)

    # return the probability of error
    return p_error
```

[22]:
```
# plot the minimum-P(error) threshold
errors = probability_of_error(data, labels, thresholds)
min_error_index = np.argmin(errors)
min_error = np.min(errors)
min_error_threshold = thresholds[min_error_index]
```

```
# plot the ROC curve (previously determined) with the minimum-P(error) threshold
plt.plot(false_positive_rate, true_positive_rate)
plt.plot(false_positive_rate[min_error_index],␣
 ↪true_positive_rate[min_error_index], 'ro')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve with Minimum-P(error) Threshold')
plt.show()
```



[34]:
```
# P(error; threshold) = P(D=1|Y=0)P(Y=0) + P(D=0|Y=1)P(Y=1)
def probability_of_error_optimal(data: np.array, labels: np.array, thresholds:␣
 ↪np.array, mu_0: np.array = mu_0, mu_1: np.array = mu_1, sigma_0: np.array =␣
 ↪sigma_0, sigma_1: np.array = sigma_1) -> np.array:
    """
    Given data, labels, and an array of thresholds, return the probability of␣
 ↪error for each threshold taking into account
    the priors and loss matrix

    Args:
```

```
        data (np.array): the data set
        labels (np.array): the labels for the data set
        thresholds (np.array): an array of thresholds for the likelihood ratio␣
↪test
        mu_0 (np.array, optional): the mean of the class 0 Gaussian pdf.␣
↪Defaults to mu_0.
        mu_1 (np.array, optional): the mean of the class 1 Gaussian pdf.␣
↪Defaults to mu_1.
        sigma_0 (np.array, optional): the covariance matrix of the class 0␣
↪Gaussian pdf. Defaults to sigma_0.
        sigma_1 (np.array, optional): the covariance matrix of the class 1␣
↪Gaussian pdf. Defaults to sigma_1.

    Returns:
        np.array: an array of probabilities of error for each threshold, taking␣
↪into account the priors and loss matrix
    """
    # compute the predicted labels for all thresholds and data points at once
    predictions = np.array([minimum_expected_risk_classification_rule(data,␣
↪threshold, mu_0, mu_1, sigma_0, sigma_1) for threshold in thresholds])

    # compute the number of errors for each class
    errors_0 = np.sum((labels == 1) & (predictions == 0), axis=1)
    errors_1 = np.sum((labels == 0) & (predictions == 1), axis=1)

    # compute the probability of error for each threshold
    p_errors_0 = errors_0 / len(data)
    p_errors_1 = errors_1 / len(data)

    # compute the probability of error taking into account the priors and loss␣
↪matrix for each threshold
    p_errors = p_errors_0 * p_0 * loss_matrix[0][1] + p_errors_1 * p_1 *␣
↪loss_matrix[1][0]

    return p_errors
```

[35]:
```
# determine the theoretical optimal threshold from priors and loss values
errors_optimal = probability_of_error_optimal(data, labels, thresholds)
min_error_index_optimal = np.argmin(errors_optimal)
min_error_optimal = np.min(errors_optimal)
min_error_optimal_threshold = thresholds[min_error_index_optimal]
```

[36]:
```
# print the estimated minimum probability of error
print("The estimated minimum probability of error is: ", min_error_optimal)
# print the empirical selected threshold
print("The empirical selected threshold is: ", min_error_threshold)
```

```
# print the theoretical optimal threshold
print("The theoretical optimal threshold is: ", min_error_optimal_threshold)
```

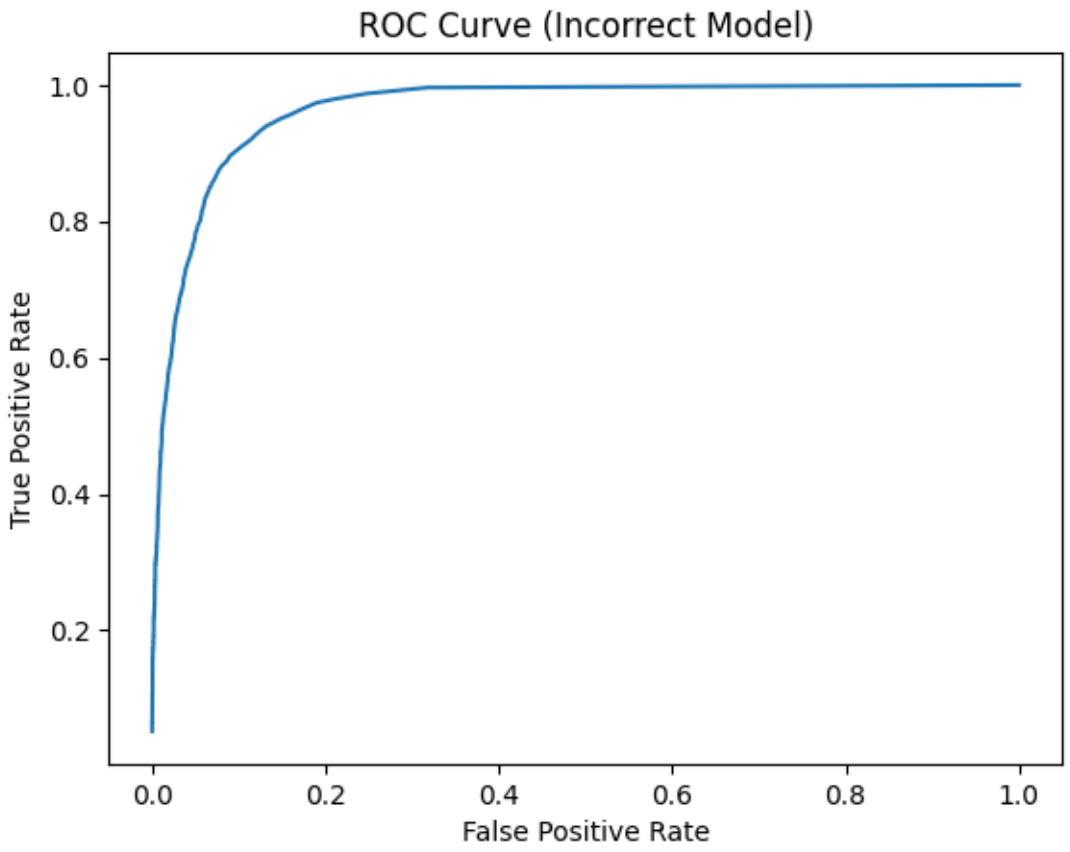
```
The estimated minimum probability of error is:  0.03838
The empirical selected threshold is:  2.8000000000000003
The theoretical optimal threshold is:  1.2000000000000002
```

0.0.2 Part B: ERM classification using incorrect knowledge of data distribution

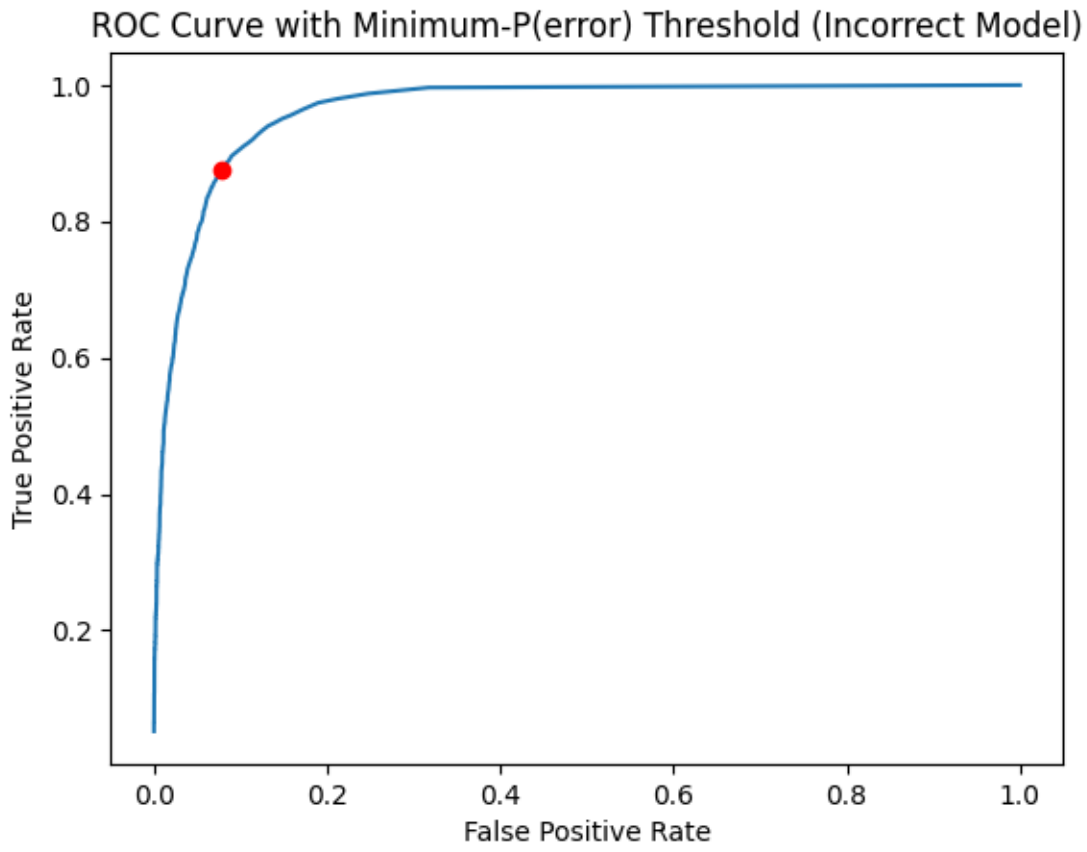
Assume we know the true class priors, but the class conditional pdfs are both gaussian with true means but the covariance matrices are diagonal (with diagonal entries equal to true variances, but the off-diagonal entries are zero). Analyze the impact of this model mismatch by implementing the ERM classification rule using the incorrect model. Repeat the same steps in part a on the same data set

[26]:
```
# set up the diagonal covariance matrices
sigma_0_incorrect = np.array([[2, 0, 0, 0],
                              [0, 1, 0, 0],
                              [0, 0, 1, 0],
                              [0, 0, 0, 2]])
sigma_1_incorrect = np.array([[1, 0, 0, 0],
                              [0, 2, 0, 0],
                              [0, 0, 1, 0],
                              [0, 0, 0, 3]])
```

[28]:
```
# plot the ROC curve for the incorrect model
true_positive_rate_incorrect, false_positive_rate_incorrect = ROC_curve(data,␣
 ↪labels, thresholds, mu_0, mu_1, sigma_0_incorrect, sigma_1_incorrect)

plt.plot(false_positive_rate_incorrect, true_positive_rate_incorrect)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve (Incorrect Model)')
plt.show()
```



[29]:
```
# plot the minimum-P(error) threshold for the incorrect model
errors_incorrect = probability_of_error(data, labels, thresholds, mu_0, mu_1,␣
 ↪sigma_0_incorrect, sigma_1_incorrect)
min_error_index_incorrect = np.argmin(errors_incorrect)
min_error_incorrect = np.min(errors_incorrect)
min_error_threshold_incorrect = thresholds[min_error_index_incorrect]

# plot the ROC curve (previously determined) with the minimum-P(error) threshold
plt.plot(false_positive_rate_incorrect, true_positive_rate_incorrect)
plt.plot(false_positive_rate_incorrect[min_error_index_incorrect],␣
 ↪true_positive_rate_incorrect[min_error_index_incorrect], 'ro')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve with Minimum-P(error) Threshold (Incorrect Model)')
plt.show()
```



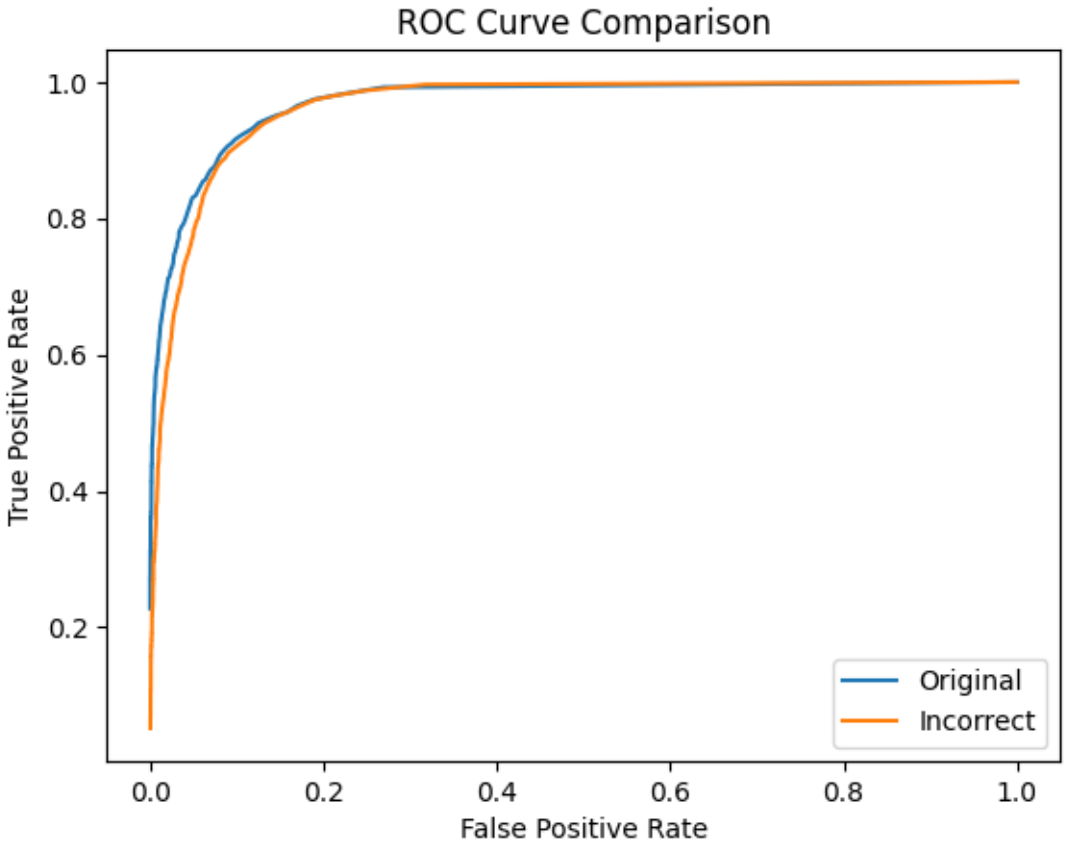
[30]:
```
# determine the theoretical optimal threshold from priors and loss values for␣
 ↪the incorrect model
errors_optimal_incorrect = probability_of_error_optimal(data, labels,␣
 ↪thresholds, mu_0, mu_1, sigma_0_incorrect, sigma_1_incorrect)
min_error_index_optimal_incorrect = np.argmin(errors_optimal_incorrect)
min_error_optimal_incorrect = np.min(errors_optimal_incorrect)
min_error_optimal_threshold_incorrect =␣
 ↪thresholds[min_error_index_optimal_incorrect]
```

[31]:
```
# print the estimated minimum probability of error for the incorrect model
print("The estimated minimum probability of error with the incorrect model is:␣
 ↪", min_error_optimal_incorrect)
# print the empirical selected threshold for the incorrect model
print("The empirical selected threshold with the incorrect model is: ",␣
 ↪min_error_threshold_incorrect)
# print the theoretical optimal threshold for the incorrect model
print("The theoretical optimal threshold with the incorrect model is: ",␣
 ↪min_error_optimal_threshold_incorrect)
```

```
The estimated minimum probability of error with the incorrect model is:  0.04036
```

```
The empirical selected threshold with the incorrect model is:  2.2
The theoretical optimal threshold with the incorrect model is:  1.0
```

[37]:
```
# plot the original ROC curve and the ROC curve for the incorrect model on the␣
 ↪same plot
plt.plot(false_positive_rate, true_positive_rate, label='Original')
plt.plot(false_positive_rate_incorrect, true_positive_rate_incorrect,␣
 ↪label='Incorrect')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend()
plt.show()
```



The model mismatch did negatively affect the ROC curve and minimum achieveable probability of error. However, at least with the threshold values used, it did not have an enormous impact. When comparing the ROC curves directly, the incorrect model produces a curve slightly worse since it has a smaller area under the curve. When looking at the minimum probabilities of error, the original model produces a value marginally smaller than the incorrect model, which is desirable. This marginal difference is likely due to the covariance matrices having the same diagonal entries.

p2

June 1, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal

%matplotlib widget
```

[2]:
```
# we're working with 3-dimensional data
# this 3-D random vector X takes values from a mixture of four Guassians
# the first gaussian is the class-conditional pdf for class 1, and antoher is␣
 ↪for class 2
# the remaining 2 gaussian components are the class-conditional pdfs for class␣
 ↪3 with equal weights
# the class priors are p1 = 0.3, p2 = 0.3, p3 = 0.4
# we want to set the distances between the means of pairs of gaussians to twice␣
 ↪the average standard deviation

# set up the parameters of the class-conditioned Gaussian pdfs
# the distances between the means of pairs of gaussians are set to twice the␣
 ↪average standard deviation
sigma_1 = np.eye(3)
sigma_2 = np.eye(3)
sigma_3 = np.eye(3)
sigma_4 = np.eye(3)

avg_std = np.sqrt(np.trace(sigma_1) + np.trace(sigma_2) + np.trace(sigma_3) +␣
 ↪np.trace(sigma_4)) / 4

mult = 1.5 * avg_std

mu_1 = np.array([mult, mult, mult])
mu_2 = np.array([0, mult, mult])
mu_3 = np.array([mult, 0, mult])
mu_4 = np.array([mult, mult, 0])

# set up the parameters of the class priors
p_1 = 0.3
```

```
p_2 = 0.3
p_3 = 0.4

# verify that the distances between the means of pairs of gaussians are set to␣
 ↪twice the average standard deviation
print("The distance between mu_1 and mu_2 is", np.linalg.norm(mu_1 - mu_2))
print("The distance between mu_2 and mu_3 is", np.linalg.norm(mu_2 - mu_3))
print("The distance between mu_3 and mu_4 is", np.linalg.norm(mu_3 - mu_4))
print("The distance between mu_4 and mu_1 is", np.linalg.norm(mu_4 - mu_1))
print("The distance between mu_1 and mu_3 is", np.linalg.norm(mu_1 - mu_3))
print("The distance between mu_2 and mu_4 is", np.linalg.norm(mu_2 - mu_4))
print("Twice the average standard deviation is", 2 * avg_std)
```

```
The distance between mu_1 and mu_2 is 1.299038105676658
The distance between mu_2 and mu_3 is 1.8371173070873836
The distance between mu_3 and mu_4 is 1.8371173070873836
The distance between mu_4 and mu_1 is 1.299038105676658
The distance between mu_1 and mu_3 is 1.299038105676658
The distance between mu_2 and mu_4 is 1.8371173070873836
Twice the average standard deviation is 1.7320508075688772
```

0.0.1 Part A

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

[3]:
```
# generate 10000 samples from this data distribution and keep track of the true␣
 ↪class labels
samples = 10000
gaussian_1 = np.random.multivariate_normal(mu_1, sigma_1, samples)
gaussian_2 = np.random.multivariate_normal(mu_2, sigma_2, samples)
gaussian_3 = np.random.multivariate_normal(mu_3, sigma_3, samples)
gaussian_4 = np.random.multivariate_normal(mu_4, sigma_4, samples)

data = []
labels = []

# use the class priors to generate the labels
for i in range(samples):
    # class 1
    if np.random.rand() < p_1:
        data.append(gaussian_1[i])
        labels.append(1)
    # class 2
    elif np.random.rand() < p_1 + p_2:
        data.append(gaussian_2[i])
        labels.append(2)
    # class 3
    elif np.random.rand() < p_1 + p_2 + p_3:
```

```
        # class 3 data originates from a mixture of gaussians 3 and 4 with␣
 ↪equal weights
        if np.random.rand() < 0.5:
            data.append(gaussian_3[i])
            labels.append(3)
        else:
            data.append(gaussian_4[i])
            labels.append(3)

# convert the data and labels to numpy arrays
data = np.array(data)
labels = np.array(labels)
```

2. Specify the decision rule that acheives minimum probability of error (use 0-1 loss). Implement this classifier with true data distribution knowledge. Classify the 10k samples and count the samples corresponding to each decision-label pair to empirically estimate the confusion matrix.

[4]:
```
# likelihood estimation to use in the decision rule
# the likelihood of a sample x is the probability of x given the class label
# the likelihood of x given class 1 is the pdf of the multivariate gaussian␣
 ↪with mean mu_1 and covariance sigma_1 evaluated at x
# the likelihood of x given class 2 is the pdf of the multivariate gaussian␣
 ↪with mean mu_2 and covariance sigma_2 evaluated at x
# the likelihood of x given class 3 is the pdf of the multivariate gaussians␣
 ↪with means mu_3, mu_4 and covariances sigma_3, sigma_4 equally weighted␣
 ↪evaluated at x
def likelihood(x: np.array, label: int) -> float:
    """
    Given a sample x and a class label, return the likelihood of x given the␣
 ↪class label

    Args:
        x (np.array): a sample
        label (int): a class label

    Raises:
        ValueError: if the label is not 1, 2, or 3

    Returns:
        float: the likelihood of x given the class label
    """
    if label == 1:
        return multivariate_normal.pdf(x, mu_1, sigma_1)
    elif label == 2:
        return multivariate_normal.pdf(x, mu_2, sigma_2)
    elif label == 3:
```

```
        return 0.5 * multivariate_normal.pdf(x, mu_3, sigma_3) + 0.5 *␣
↪multivariate_normal.pdf(x, mu_4, sigma_4)
    else:
        raise ValueError("Invalid label")
```

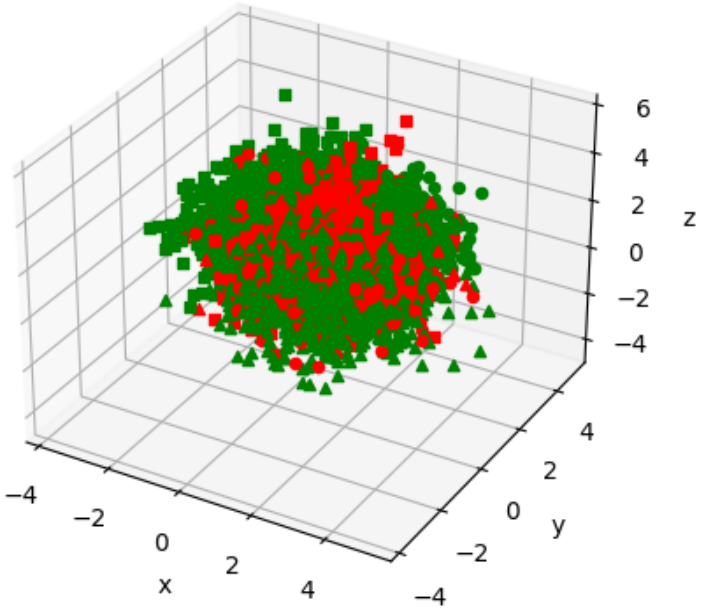
[5]:
```
# the decision rule is to choose the class label that maximizes the likelihood␣
↪of the sample
# the decision rule is implemented with true data distribution knowledge
# the decision rule is implemented with true class priors
def minimum_probability_error_rule(x: np.array, loss: np.array) -> int:
    """
    Given a sample x and a loss matrix, return the class label that minimizes␣
↪the probability of error

    Args:
        x (np.array): a sample
        loss (np.array): the loss matrix

    Returns:
        int: the class label that minimizes the probability of error
    """
    # compute the likelihoods of x given each class label
    likelihoods = [likelihood(x, 1), likelihood(x, 2), likelihood(x, 3)]

    # calculate the posterior probabilities of x given each class label
    # the posterior probability of x given class 1 is the likelihood of x given␣
↪class 1 times the class prior of class 1
    # the posterior probability of x given class 2 is the likelihood of x given␣
↪class 2 times the class prior of class 2
    # the posterior probability of x given class 3 is the likelihood of x given␣
↪class 3 times the class prior of class 3
    posterior_probs = [likelihoods[0] * p_1, likelihoods[1] * p_2,␣
↪likelihoods[2] * p_3]

    # calculate the expected loss of each class label
    loss_1 = posterior_probs[0] * loss[0][0] + posterior_probs[1] * loss[0][1]␣
↪+ posterior_probs[2] * loss[0][2]
    loss_2 = posterior_probs[0] * loss[1][0] + posterior_probs[1] * loss[1][1]␣
↪+ posterior_probs[2] * loss[1][2]
    loss_3 = posterior_probs[0] * loss[2][0] + posterior_probs[1] * loss[2][1]␣
↪+ posterior_probs[2] * loss[2][2]

    # return the class label that minimizes the expected loss
    return np.argmin([loss_1, loss_2, loss_3]) + 1
```

[6]:
```
# classify the 10k samples and count the samples corresponding to each␣
 ↪decision-label pair to empirically estimate the confusion matrix
# we're using 0-1 loss
loss_1 = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])

confusion_matrix = np.zeros((3, 3), dtype=int)

for i in range(samples):
    guess = minimum_probability_error_rule(data[i], loss_1)
    confusion_matrix[labels[i] - 1][guess - 1] += 1

# print the confusion matrix
print("The confusion matrix is")
print(confusion_matrix)
```

```
The confusion matrix is
[[1356  610 1066]
 [ 669 2703  782]
 [ 456  325 2033]]
```

3. Provide a vizualization of the data (scatter plot in 3 dimensions). For each sample, indicate the true class label with a different marker and if it was correctly classified with a different color

[8]:
```
# plot the data
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot the samples
for i in range(samples):
    # determine the color to use
    color = 'green' if labels[i] == minimum_probability_error_rule(data[i],␣
 ↪loss_1) else 'red'

    # determine the marker to use
    marker = 'o' if labels[i] == 1 else 's' if labels[i] == 2 else '^'

    # plot the sample
    ax.scatter(data[i][0], data[i][1], data[i][2], c=color, marker=marker)

# label the axes
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

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



0.0.2 Part B

Repeat the exercise for the ERM classificaiton rule with the following loss matrices. Using the same 10k samples, estimate the minimum expected risk that this optimal ERM classification rule will achieve.

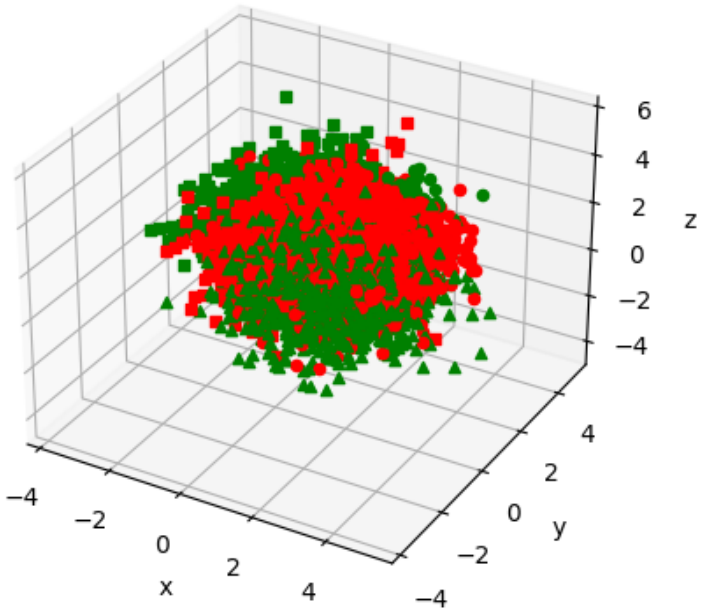
[9]:
```
loss_10 = np.array([[0, 1, 10],
                    [1, 0, 10],
                    [1, 1, 0]])

loss_100 = np.array([[0, 1, 100],
                     [1, 0, 100],
                     [1, 1, 0]])

confusion_matrix_10 = np.zeros((3, 3), dtype=int)

for i in range(samples):
    guess = minimum_probability_error_rule(data[i], loss_10)
    confusion_matrix_10[labels[i] - 1][guess - 1] += 1

# print the confusion matrix
print("The confusion matrix for caring 10x more is")
```

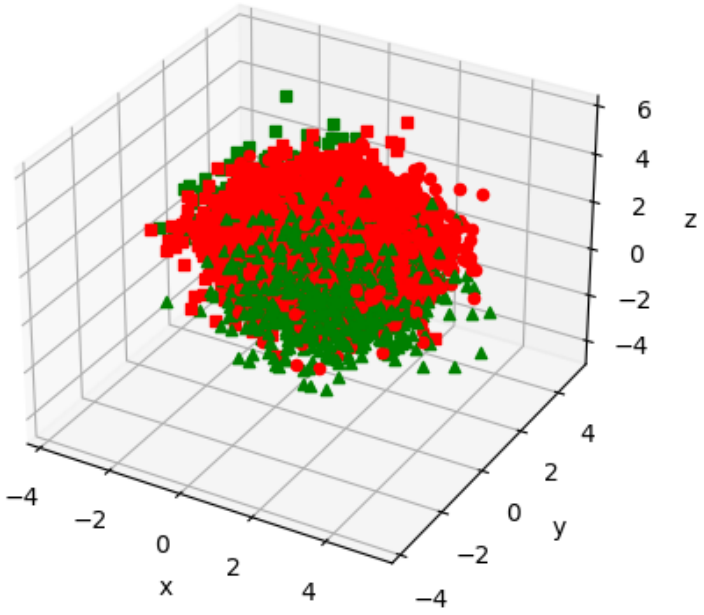
```
print(confusion_matrix_10)

confusion_matrix_100 = np.zeros((3, 3), dtype=int)

for i in range(samples):
    guess = minimum_probability_error_rule(data[i], loss_100)
    confusion_matrix_100[labels[i] - 1][guess - 1] += 1

# print the confusion matrix
print("The confusion matrix for caring 100x more is")
print(confusion_matrix_100)
```

```
The confusion matrix for caring 10x more is
[[  36  102 2894]
 [  29  909 3216]
 [   0   23 2791]]
The confusion matrix for caring 100x more is
[[   0    4 3028]
 [   0   66 4088]
 [   0    1 2813]]
```

[15]:
```
# plot the data for the 10x loss
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot the samples
for i in range(samples):
    # determine the color to use
    color = 'green' if labels[i] == minimum_probability_error_rule(data[i],␣
 ↪loss_10) else 'red'

    # determine the marker to use
    marker = 'o' if labels[i] == 1 else 's' if labels[i] == 2 else '^'

    # plot the sample
    ax.scatter(data[i][0], data[i][1], data[i][2], c=color, marker=marker)

# label the axes
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

# show the plot
plt.show()

# plot the data for the 100x loss
fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')

# plot the samples
for i in range(samples):
    # determine the color to use
    color = 'green' if labels[i] == minimum_probability_error_rule(data[i],␣
 ↪loss_100) else 'red'

    # determine the marker to use
    marker = 'o' if labels[i] == 1 else 's' if labels[i] == 2 else '^'

    # plot the sample
    ax.scatter(data[i][0], data[i][1], data[i][2], c=color, marker=marker)

# label the axes
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

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

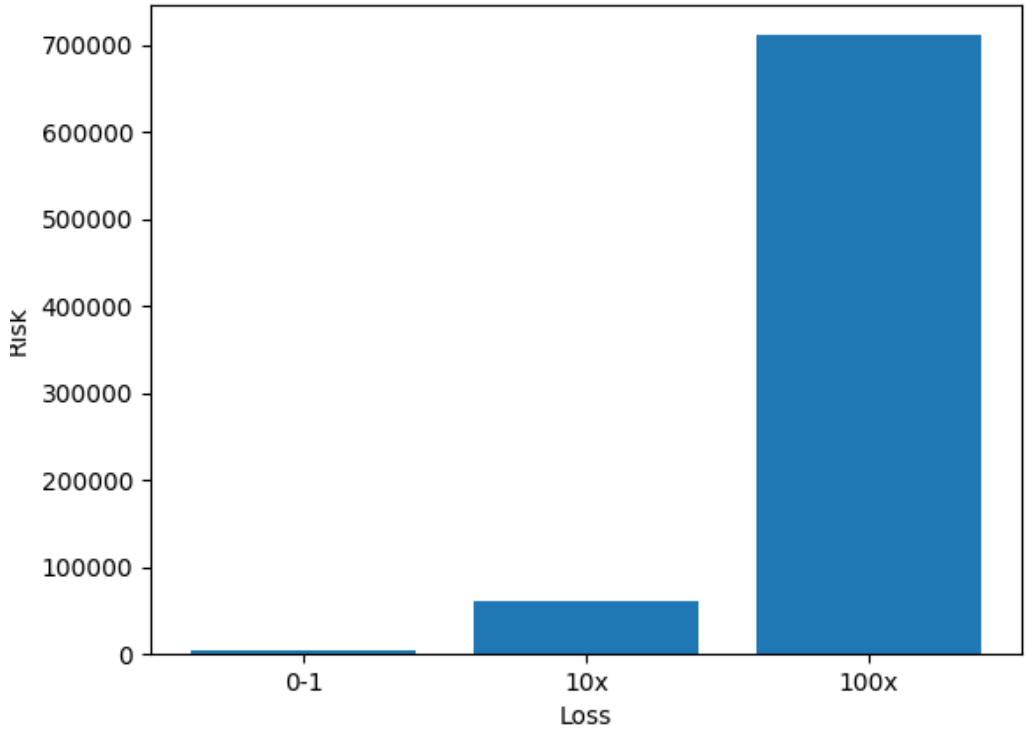




[14]:
```python
# estimate the minimum expected risk for each loss matrix
risk_1 = 0
risk_10 = 0
risk_100 = 0

# use the confusion matrices to estimate the minimum expected risk
for i in range(3):
    for j in range(3):
        risk_1 += confusion_matrix[i][j] * loss_1[i][j]
        risk_10 += confusion_matrix_10[i][j] * loss_10[i][j]
        risk_100 += confusion_matrix_100[i][j] * loss_100[i][j]

# normalize the risks
# risk_1 /= samples
# risk_10 /= samples
# risk_100 /= samples

# print the risks
print("The minimum expected risk for the 0-1 loss is", risk_1)
```

```
print("The minimum expected risk for the 10x loss is", risk_10)
print("The minimum expected risk for the 100x loss is", risk_100)

# plot the risks
fig = plt.figure()
plt.bar([1, 2, 3], [risk_1, risk_10, risk_100])
plt.xticks([1, 2, 3], ['0-1', '10x', '100x'])
plt.xlabel('Loss')
plt.ylabel('Risk')
plt.show()
```

```
The minimum expected risk for the 0-1 loss is 3908
The minimum expected risk for the 10x loss is 61254
The minimum expected risk for the 100x loss is 711605
```



We've calculated the minimum expected risk using each of the loss matrices. The notable distinction between these matrices is that they progressively care more about not making mistakes for Y = 3. As seen by the confusion matrix and the bar graph, this led to worse results as the loss matrix increased. For example, the confusion matrix for 100 has 0s all through the first column, meaning that for the true label Y = 1, there were no classifications for that label. This is because the loss matrix cared so much about not misclassifying label 3 that it would incorrectly bias towards

classifying as 3.

Another interesting note is that as the loss matrix increased, the number of correct classifications for label 3 increased. This came at a significant tradeoff, however, since it also meant that the number of incorrect classifications increased. Thus, it's important to consider all the effects of classification and not just focus on one number to determine success.

p3

June 1, 2023

[16]:
```
import numpy as np
from scipy.stats import multivariate_normal
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
```

[17]:
```
# load the wine data set
# characteristics: 11 features, 4898 samples, 11 classes (0-10)

# load the data set
wine = np.loadtxt('datasets/winequality/winequality-white.csv', delimiter=';',␣
 ↪skiprows=1)

# extract the data and labels
wine_data = []
wine_labels = []

for row in wine:
    wine_data.append(row[:-1])
    wine_labels.append(row[-1])

# convert to numpy arrays
wine_data = np.array(wine_data)
wine_labels = np.array(wine_labels)

wine_possible_labels = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

[18]:
```
# load the HAR data set
# characteristics: 561 features, 10299 samples, 6 classes (1-6)

# load the files
X_test = np.loadtxt('datasets/UCI HAR Dataset/test/X_test.txt')
y_test = np.loadtxt('datasets/UCI HAR Dataset/test/y_test.txt')
X_train = np.loadtxt('datasets/UCI HAR Dataset/train/X_train.txt')
y_train = np.loadtxt('datasets/UCI HAR Dataset/train/y_train.txt')

# format the data set so that all of this data is in one data set due to the␣
 ↪given charactersitics
```

```
har_data = np.concatenate((X_test, X_train), axis=0)
har_labels = np.concatenate((y_test, y_train), axis=0)

har_possible_labels = np.array([1, 2, 3, 4, 5, 6])
```

[19]:
```
# implement minimum-probability-of-error classifier, assuming the class␣
 ↪conditional pdfs are Gaussian
# using all available samples from a class, with sample averages, estimate mean␣
 ↪vectors and covariance matrices
# using sample counts, also estimate class priors

# calculate mean vectors, covariance matrices, and priors for each class
# also return the unique labels
def calculate_parameters(data: np.array, labels: np.array) -> tuple[np.array,␣
 ↪np.array, np.array, np.array]:
    """
    Calculates the mean vectors, covariance matrices, and priors for each class.

    Args:
        data (np.array): The data set.
        labels (np.array): The labels for the data set.

    Returns:
        tuple[np.array, np.array, np.array, np.array]: The unique labels, mean␣
 ↪vectors, covariance matrices, and priors.
    """
    # get the unique labels
    unique_labels = np.unique(labels)

    # calculate the mean vectors
    mean_vectors = []
    for label in unique_labels:
        mean_vectors.append(np.mean(data[labels == label], axis=0))

    # calculate the covariance matrices
    covariance_matrices = []
    for label in unique_labels:
        covariance_matrices.append(np.cov(data[labels == label].T))

    # calculate the priors
    priors = []
    for label in unique_labels:
        priors.append(np.sum(labels == label) / len(labels))

    # convert to numpy arrays
    mean_vectors = np.array(mean_vectors)
    covariance_matrices = np.array(covariance_matrices)
```

```
    priors = np.array(priors)

    return unique_labels, mean_vectors, covariance_matrices, priors
```

[20]:
```
# mean vector and covariance matrix for wine data set
wine_unique_labels, wine_mean_vectors, wine_covariance_matrices, wine_priors =␣
 ↪calculate_parameters(wine_data, wine_labels)

# mean vector and covariance matrix for HAR data set
har_unique_labels, har_mean_vectors, har_covariance_matrices, har_priors =␣
 ↪calculate_parameters(har_data, har_labels)
```

[21]:
```
# add a regularization term to the covariance matrices to ensure the␣
 ↪regularized covaraince matrix has all eigenvalues larger than 0
# this is done by adding a small value to the diagonal of the covariance matrix
# for now, we'll use a value on the order of arithmetic average of sample␣
 ↪covariance matrices
def regularize_covariance_matrices(covariance_matrices: np.array) -> np.array:
    """
    Regularizes the covariance matrices.

    Args:
        covariance_matrices (np.array): The covariance matrices.

    Returns:
        np.array: The regularized covariance matrices.
    """
    # calculate the average covariance matrix
    average_covariance_matrix = np.mean(covariance_matrices, axis=0)

    # calculate the regularization term
    regularization_term = np.mean(np.diag(average_covariance_matrix))

    print("The regularization term is: ", regularization_term)

    # add the regularization term to the covariance matrices
    for i in range(len(covariance_matrices)):
        covariance_matrices[i] += regularization_term * np.
 ↪eye(covariance_matrices[i].shape[0])

    return covariance_matrices
```

[22]:
```
# add the regularization term to the covariance matrices
wine_covariance_matrices =␣
 ↪regularize_covariance_matrices(wine_covariance_matrices)
har_covariance_matrices =␣
 ↪regularize_covariance_matrices(har_covariance_matrices)
```

```
The regularization term is:  353.1966224105839
The regularization term is:  0.03549356441569865
```

[23]:
```python
def minimum_probability_of_error_classifier(data: np.array, unique_labels: np.array, mean_vectors: np.array, covariance_matrices: np.array, priors: np.array) -> np.array:
    """
    Implements the minimum-probability-of-error classifier.

    Args:
        data (np.array): The data set to classify.
        unique_labels (np.array): The unique, used labels for the data set.
        mean_vectors (np.array): The mean vectors for each class.
        covariance_matrices (np.array): The covariance matrices for each class.
        priors (np.array): The priors for each class.

    Returns:
        np.array: The predicted labels for the data set.
    """
    # calculate the probabilities for all data points and classes
    probabilities = np.zeros((data.shape[0], len(unique_labels)))

    for i, label in enumerate(unique_labels):
        probabilities[:, i] = multivariate_normal.pdf(data, mean_vectors[i], covariance_matrices[i]) * priors[i]

    # find the index of the maximum probability for each data point
    max_probability_indices = np.argmax(probabilities, axis=1)

    # use the max_probability_indices to find the predicted labels
    predicted_labels = unique_labels[max_probability_indices]

    return predicted_labels
```

[24]:
```python
# implement a function that will count the errors, the error probability estimate, and the confusion matrix
def calculate_classification_metrics(predicted_labels: np.array, actual_labels: np.array, possible_labels: np.array) -> tuple[int, float, np.array]:
    """
    Calculates the number of errors, the error probability estimate, and the confusion matrix.

    Args:
        predicted_labels (np.array): The predicted labels.
        actual_labels (np.array): The actual labels.
        possible_labels (np.array): All the possible labels for the data set.
```

```
    Returns:
        tuple[int, float, np.array]: The number of errors, the error␣
↪probability estimate, and the confusion matrix.
    """
    # initialize the number of errors
    number_of_errors = 0

    # initialize the confusion matrix
    confusion_matrix = np.zeros((len(possible_labels), len(possible_labels)),␣
↪dtype=int)

    # iterate through the predicted labels
    for i in range(len(predicted_labels)):
        # check if the predicted label is correct
        if predicted_labels[i] != actual_labels[i]:
            # increment the number of errors
            number_of_errors += 1

        # increment the confusion matrix
        actual_index = np.where(possible_labels == actual_labels[i])[0][0]
        predicted_index = np.where(possible_labels == predicted_labels[i])[0][0]
        confusion_matrix[actual_index, predicted_index] += 1

    # calculate the error probability estimate
    error_probability_estimate = number_of_errors / len(predicted_labels)

    return number_of_errors, error_probability_estimate, confusion_matrix
```

[25]:
```
# classify the wine data set
wine_predicted_labels = minimum_probability_of_error_classifier(wine_data,␣
↪wine_unique_labels, wine_mean_vectors, wine_covariance_matrices, wine_priors)
```

[26]:
```
# calculate the classification metrics for the wine data set
wine_number_of_errors, wine_error_probability_estimate, wine_confusion_matrix =␣
↪calculate_classification_metrics(wine_predicted_labels, wine_labels,␣
↪wine_possible_labels)

# print the classification metrics for the wine data set
print("The number of errors for the wine data set is: ", wine_number_of_errors)
print("The error probability estimate for the wine data set is: ",␣
↪wine_error_probability_estimate)
print("The confusion matrix for the wine data set is:")
print(wine_confusion_matrix)
```

```
The number of errors for the wine data set is:  2683
The error probability estimate for the wine data set is:  0.5477746018783177
The confusion matrix for the wine data set is:
```

```
[[   0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    3    0    3   14    0    0    0    0]
 [   0    0    0    1    0    7  155    0    0    0    0]
 [   0    0    0    2    0  162 1293    0    0    0    0]
 [   0    0    0    0    0  148 2050    0    0    0    0]
 [   0    0    0    0    0   17  863    0    0    0    0]
 [   0    0    0    0    0    6  169    0    0    0    0]
 [   0    0    0    0    0    0    5    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0]]
```

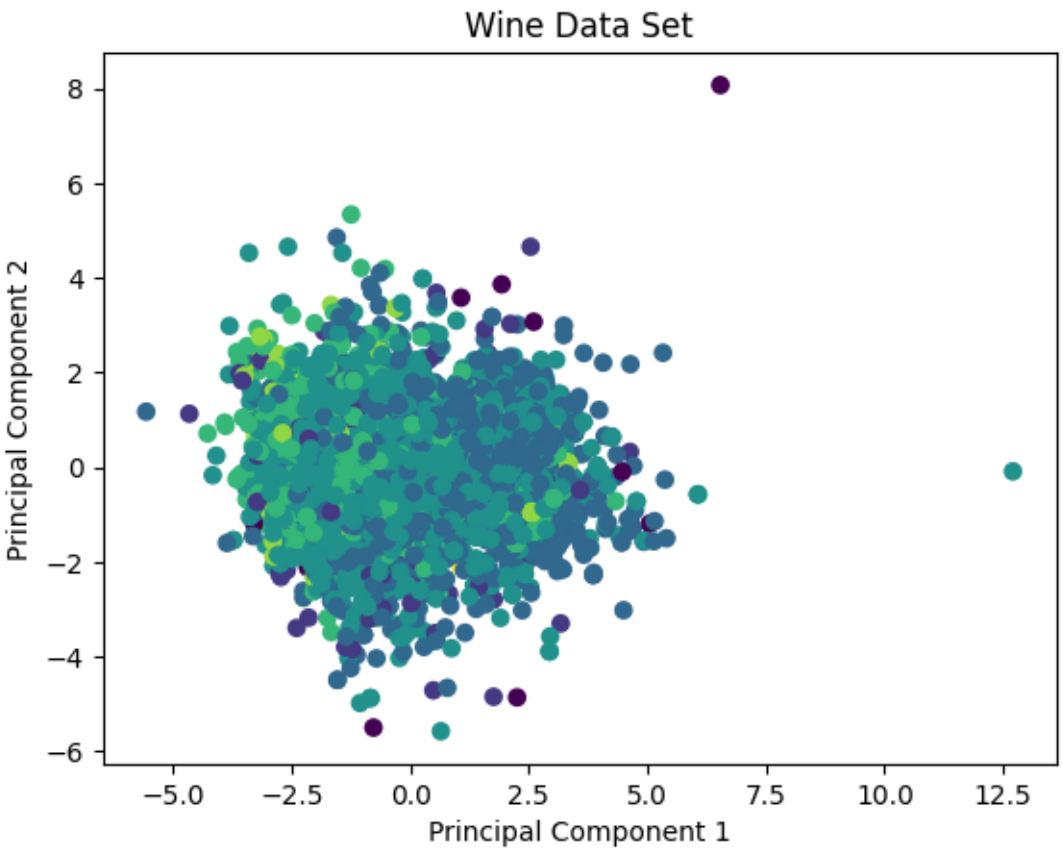
[27]:
```
# classify the HAR data set
har_predicted_labels = minimum_probability_of_error_classifier(har_data,␣
 ↪har_unique_labels, har_mean_vectors, har_covariance_matrices, har_priors)
```

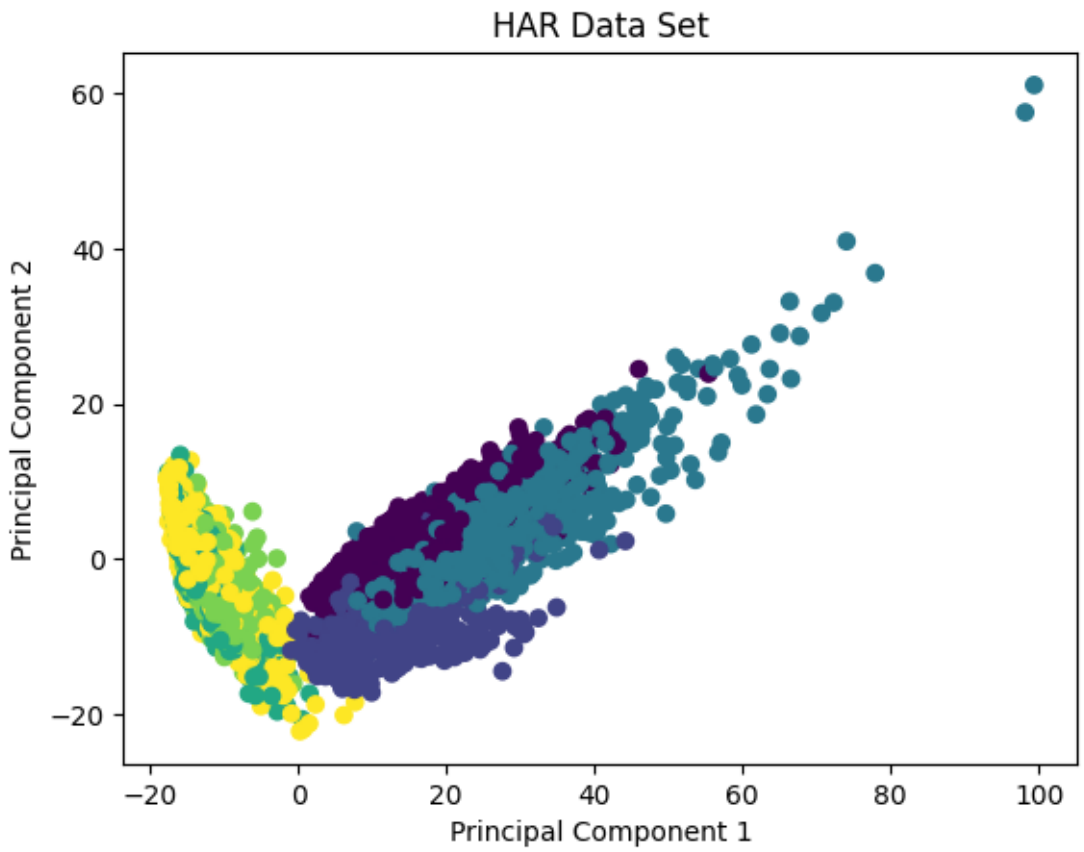
[28]:
```
# calculate the classification metrics for the HAR data set
har_number_of_errors, har_error_probability_estimate, har_confusion_matrix =␣
 ↪calculate_classification_metrics(har_predicted_labels, har_labels,␣
 ↪har_possible_labels)

# print the classification metrics for the HAR data set
print("The number of errors for the HAR data set is: ", har_number_of_errors)
print("The error probability estimate for the HAR data set is: ",␣
 ↪har_error_probability_estimate)
print("The confusion matrix for the HAR data set is:")
print(har_confusion_matrix)
```

```
The number of errors for the HAR data set is:  263
The error probability estimate for the HAR data set is:  0.02553645985047092
The confusion matrix for the HAR data set is:
[[1717    4    1    0    0    0]
 [   1 1543    0    0    0    0]
 [   2   53 1351    0    0    0]
 [   0    1    0 1584  192    0]
 [   0    0    0    9 1897    0]
 [   0    0    0    0    0 1944]]
```

[30]:
```
# use PCA to visualize the data sets
# scale the data sets
scaler = StandardScaler()
wine_scaled_data = scaler.fit_transform(wine_data)
har_scaled_data = scaler.fit_transform(har_data)

# create the PCA objects
wine_pca = PCA(n_components=2)
har_pca = PCA(n_components=2)
```

```
# fit the PCA objects
wine_pca.fit(wine_scaled_data)
har_pca.fit(har_scaled_data)

# transform the data sets
wine_pca_data = wine_pca.transform(wine_scaled_data)
har_pca_data = har_pca.transform(har_scaled_data)

# plot the data sets
# plot the wine data set
plt.figure()
plt.scatter(wine_pca_data[:, 0], wine_pca_data[:, 1], c=wine_labels)
plt.title("Wine Data Set")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()

# plot the HAR data set
plt.figure()
plt.scatter(har_pca_data[:, 0], har_pca_data[:, 1], c=har_labels)
plt.title("HAR Data Set")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```


Wine Data Set
Principal Component 2
Principal Component 1
8
6
4
2
0
−2
−4
−6
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5



Part of this exercise was making the assumption that the class conditional pdf of features for each class was Gaussian. One way of analyzing if this decision was appropriate for the data sets was to look at the confusion matrices. When looking at the confusion matrix for the wine data set, it's clear that the classifier is not a good fit. The error probability estimate for the set is over 50% and the confusion matrix has a significant amount of misclassified samples, especially for label 6. On the other hand, when looking at the HAR datat set, the confusion matrix yields much more promising results. The diagonal contains the majority of the values in the matrix, and the error probability estimate is around 3%.

The model choice has a direct impact on how the confusion matrix and probability of error. Since the wine data set was not a good fit with a Gaussian model, it led the classifier to have an incredibly high error probability and have a notable amount of misclassifications on the non-diagonal. For a data set that was much more Gaussian with the HAR set, the model performed much better.

When looking at the PCA visualization for the two data sets, the structure of the data under 2 principal components makes it a little more clear why the Gaussian assumption fit better for the HAR. For the wine data set, the data points are very clustered and have signficant overlap. For the HAR data set, there are clearer regions for each label. We look at the principal components to observe the most significant variations in the data. This is useful since it gives better insight into the separability of the data (or lackof in the wine data set instance).