A2 - Final

October 17, 2023

Jane Shen, 20841468

[1]:
```
# library imports
import numpy as np
import matplotlib.pyplot as plt
```

0.0.1 Exercise 1: MMD Classifier

Question 1

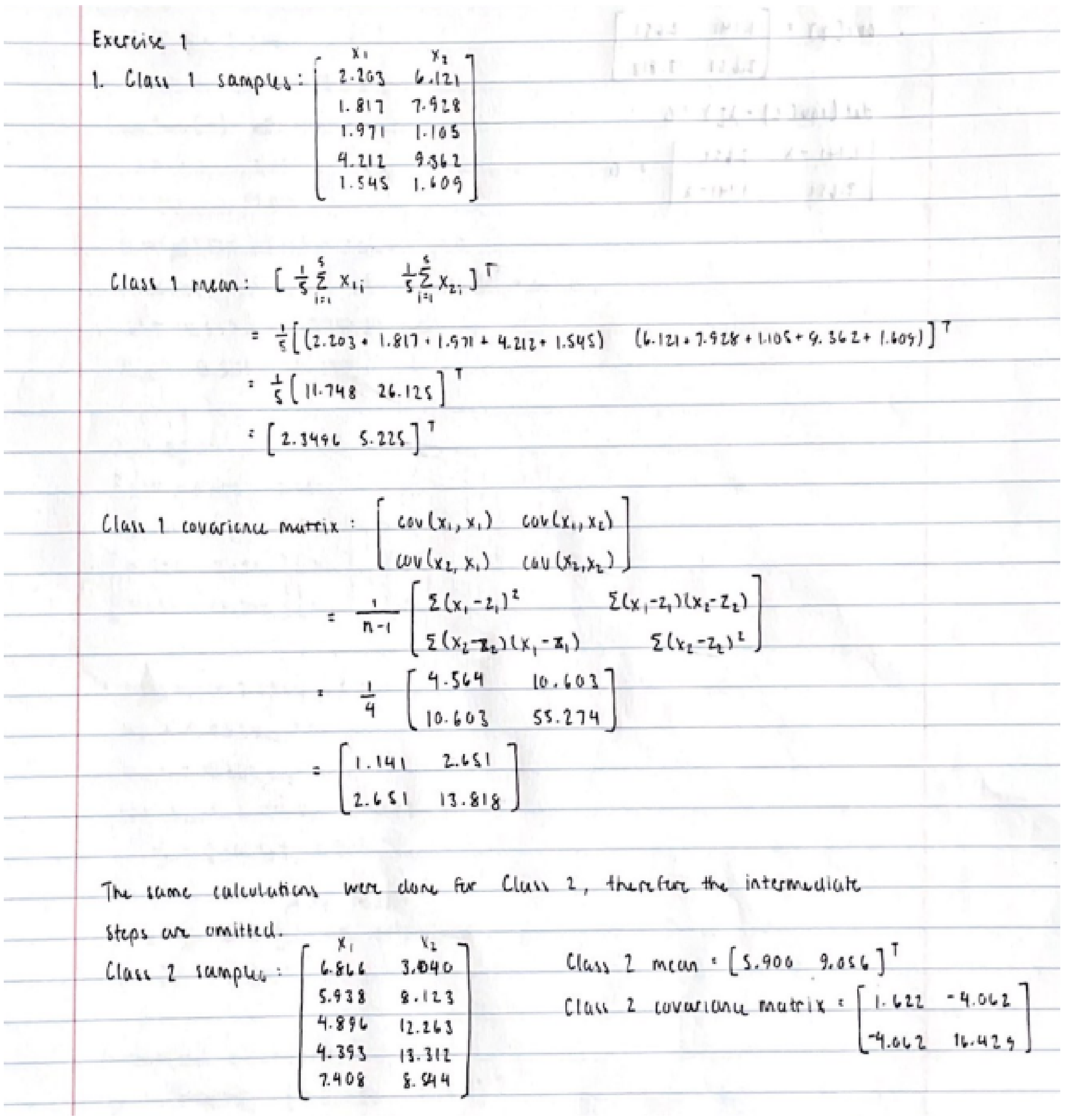
[2]:
```
class category:
    def __init__(self, mean, cov):
        self.mean = mean
        self.cov = cov
        self.samples = None
        self.sample_mean = None
        self.sample_cov = None
        self.eval = None
        self.evec = None
        self.cov_inverse = None

    def generate_samples(self, N):
        self.samples = np.random.multivariate_normal(self.mean, self.cov, N)

    def add_noise(self, mean, cov):
        for idx, _ in enumerate(self.samples):
            noise = np.random.multivariate_normal(mean, cov, 1)
            self.samples[idx] = self.samples[idx] + noise

    # for hand calculations
    def set_smean(self, smean):
        self.sample_mean = smean

    def set_scov(self, scov):
        self.sample_cov = scov

    def set_evals(self, evals):
        self.eval = evals
```

```python
    def set_evecs(self, evecs):
        self.evec = evecs # normalized by hand

    def set_cov_inverse(self, cov_inverse):
        self.cov_inverse = cov_inverse
```

[3]:
```python
# generating 5 random samples for two classes
c1 = category(np.array([1, 3]), np.array([[1, 0], [0, 15]]))
c2 = category(np.array([4, 7]), np.array([[3, 4], [4, 11]]))

cls_labels = ['c1', 'c2']
dataset = {'c1': c1, 'c2': c2}

np.random.seed(42)
for cls in cls_labels:
    dataset[cls].generate_samples(5)
    dataset[cls].add_noise(np.array([2, 2]), np.array([[2, 0], [0, 3]]))
```

[4]:
```python
# printing samples
for cls in cls_labels:
    print(f'{cls} samples:')
    print(dataset[cls].samples)
    print('\n')
```

```
c1 samples:
[[2.20309436 6.12110265]
 [1.81724299 7.92757787]
 [1.97066839 1.10548255]
 [4.21184757 9.36198997]
 [1.54526099 1.60898969]]



c2 samples:
[[ 6.86584925  3.04033067]
 [ 5.9377443   8.122781  ]
 [ 4.89585036 12.26318217]
 [ 4.39336291 13.31151622]
 [ 7.40846803  8.54434762]]
```



Exercise 1

1. Class 1 samples: $\begin{bmatrix} x_1 & x_2 \\ 2.203 & 6.121 \\ 1.817 & 7.928 \\ 1.971 & 1.105 \\ 4.212 & 9.362 \\ 1.545 & 1.609 \end{bmatrix}$

Class 1 mean: $\left[\frac{1}{5}\sum_{i=1}^{5} x_{1i} \quad \frac{1}{5}\sum_{i=1}^{5} x_{2i} \right]^T$

$= \frac{1}{5}\left[(2.203 + 1.817 + 1.971 + 4.212 + 1.545) \quad (6.121 + 7.928 + 1.105 + 9.362 + 1.609) \right]^T$

$= \frac{1}{5}\left[11.748 \quad 26.125 \right]^T$

$= \left[2.3496 \quad 5.225 \right]^T$

Class 1 covariance matrix: $\begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) \\ cov(x_2, x_1) & cov(x_2, x_2) \end{bmatrix}$

$= \frac{1}{n-1} \begin{bmatrix} \Sigma(x_1 - \bar{x}_1)^2 & \Sigma(x_1 - \bar{x}_1)(x_2 - \bar{x}_2) \\ \Sigma(x_2 - \bar{x}_2)(x_1 - \bar{x}_1) & \Sigma(x_2 - \bar{x}_2)^2 \end{bmatrix}$

$= \frac{1}{4} \begin{bmatrix} 4.564 & 10.603 \\ 10.603 & 55.274 \end{bmatrix}$

$= \begin{bmatrix} 1.141 & 2.651 \\ 2.651 & 13.818 \end{bmatrix}$

The same calculations were done for Class 2, therefore the intermediate steps are omitted.

Class 2 samples: $\begin{bmatrix} x_1 & x_2 \\ 6.866 & 3.040 \\ 5.938 & 8.123 \\ 4.896 & 12.263 \\ 4.393 & 13.312 \\ 7.408 & 8.544 \end{bmatrix}$

Class 2 mean $= \left[5.900 \quad 9.056 \right]^T$

Class 2 covariance matrix $= \begin{bmatrix} 1.622 & -4.062 \\ -4.062 & 16.429 \end{bmatrix}$

[5]:
```python
# setting sample means from hand calculations
c1.set_smean(np.array([2.3496, 5.225]))
c2.set_smean(np.array([5.900, 9.056]))

# setting sample covariance matrices from hand calculations
c1.set_scov(np.array([[1.141, 2.651], [2.651, 13.818]]))
c2.set_scov(np.array([[1.622, -4.062], [-4.062, 16.429]]))

# printing results
for cls in cls_labels:
    print(f'{cls} sample mean:')
    print(dataset[cls].sample_mean)
    print('\n')
    print(f'{cls} sample covariance matrix:')
    print(dataset[cls].sample_cov)
    print('\n')
```

```
c1 sample mean:
[2.3496 5.225 ]


c1 sample covariance matrix:
[[ 1.141  2.651]
 [ 2.651 13.818]]


c2 sample mean:
[5.9   9.056]


c2 sample covariance matrix:
[[ 1.622 -4.062]
 [-4.062 16.429]]
```

Question 2

2 Class 1:

$$\mathrm{cov}[\underline{x}] = \begin{bmatrix} 1.141 & 2.651 \\ 2.651 & 13.818 \end{bmatrix}$$

$$\det(\mathrm{cov}[\underline{x}] - \lambda I) = 0$$

$$\begin{vmatrix} 1.141-\lambda & 2.651 \\ 2.651 & 13.818-\lambda \end{vmatrix} = 0$$

$$(1.141-\lambda)(13.818-\lambda) - (2.651)^2 = 0$$

$$15.766338 - 14.959\lambda + \lambda^2 - 7.027801 = 0$$

$$\lambda^2 - 14.959\lambda + 8.738537 = 0$$

$$\lambda_{1,2} = 0.609,\ 14.350$$

$\lambda_1 = 0.609$:

$$\begin{bmatrix} 1.141-0.609 & 2.651 \\ 2.651 & 13.818-0.609 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0.532 & 2.651 \\ 2.651 & 13.209 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$0.532 v_{11} + 2.651 v_{12} = 0$$

$$v_{11} + 4.983 v_{12} = 0$$

$$v_{11} = -4.983 v_{12}$$

Let $v_{12} = 1 \rightarrow v_{11} = -4.983$

$\therefore \underline{v}_1 = [-4.983 \quad 1]^T$

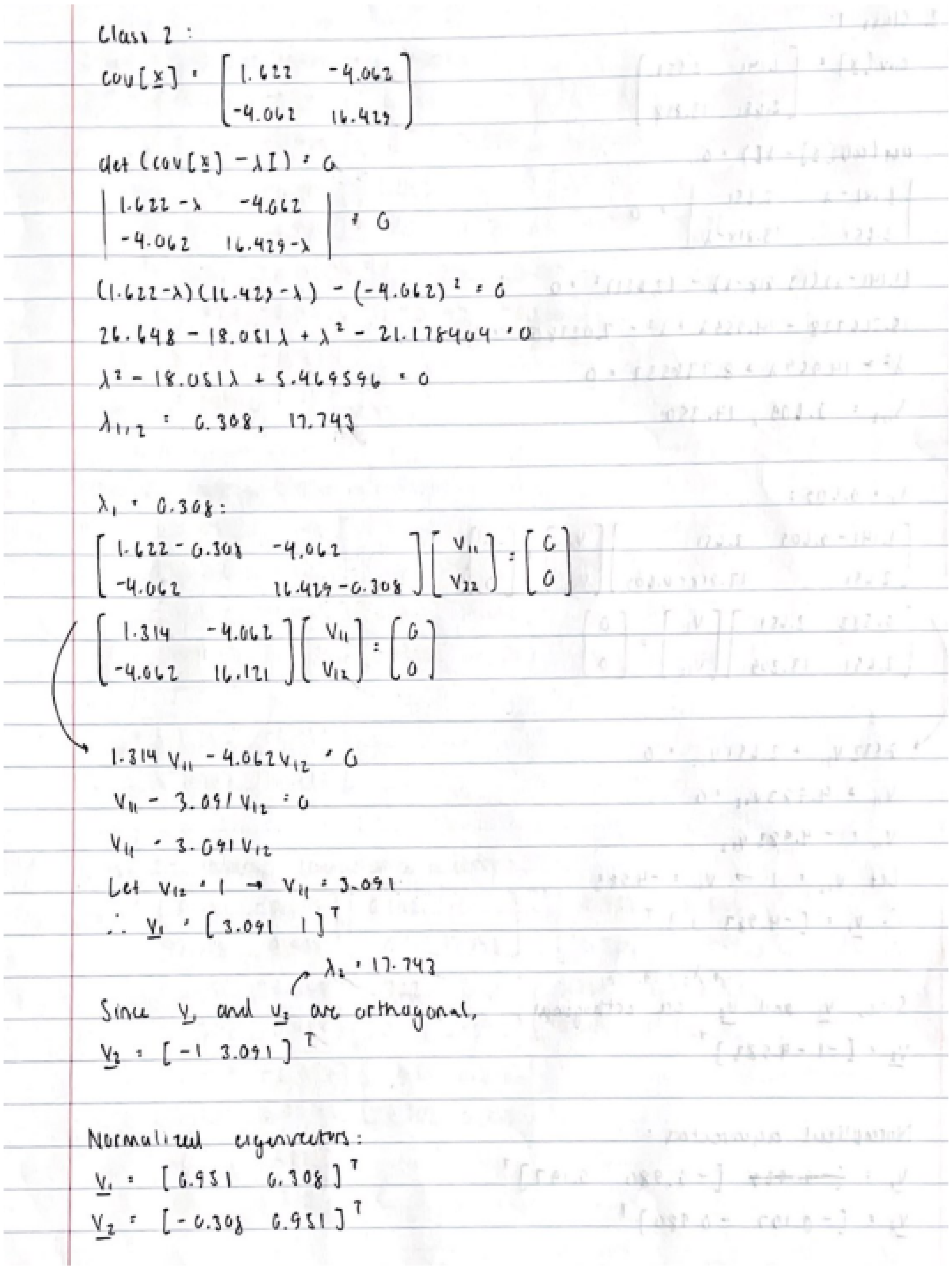
Since $\underline{v}_1$ and $\overset{\lambda_2 = 14.350}{\underline{v}_2}$ are orthogonal,

$\underline{v}_2 = [-1 \quad -4.983]^T$

Normalized eigenvectors:

$\underline{v}_1 =$ ~~[−0.1917~~ $[-0.980 \quad 0.197]^T$

$\underline{v}_2 = [-0.197 \quad -0.980]^T$


Class 2:

$$cov[\underline{x}] = \begin{bmatrix} 1.622 & -4.062 \\ -4.062 & 16.429 \end{bmatrix}$$

$$\det(cov[\underline{x}] - \lambda I) = 0$$

$$\begin{vmatrix} 1.622-\lambda & -4.062 \\ -4.062 & 16.429-\lambda \end{vmatrix} = 0$$

$$(1.622-\lambda)(16.429-\lambda) - (-4.062)^2 = 0$$

$$26.648 - 18.051\lambda + \lambda^2 - 21.178404 = 0$$

$$\lambda^2 - 18.051\lambda + 5.469596 = 0$$

$$\lambda_{1,2} = 0.308,\ 17.743$$

$\lambda_1 = 0.308$:

$$\begin{bmatrix} 1.622-0.308 & -4.062 \\ -4.062 & 16.429-0.308 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1.314 & -4.062 \\ -4.062 & 16.121 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$1.314 v_{11} - 4.062 v_{12} = 0$$

$$v_{11} - 3.091 v_{12} = 0$$

$$v_{11} = 3.091 v_{12}$$

Let $v_{12} = 1 \rightarrow v_{11} = 3.091$

$\therefore \underline{v}_1 = [3.091 \ \ 1]^T$

Since $\underline{v}_1$ and $\underline{v}_2$ are orthogonal, $\lambda_2 = 17.743$

$\underline{v}_2 = [-1 \ \ 3.091]^T$

Normalized eigenvectors:

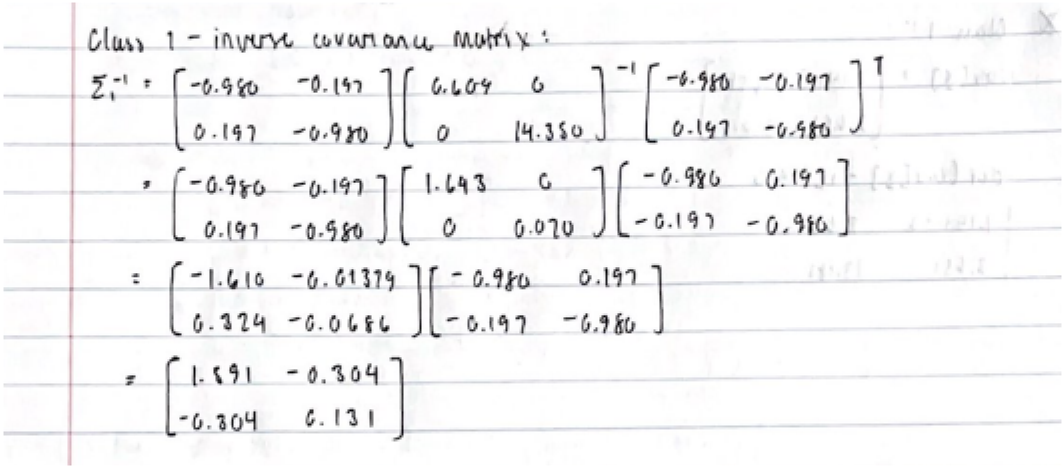
$\underline{v}_1 = [0.951 \ \ 0.308]^T$

$\underline{v}_2 = [-0.308 \ \ 0.951]^T$

[6]:
```python
# setting eigenvalues and normalized eigenvectors from hand calculations
c1.set_evals(np.array([0.609, 14.350]))
c1.set_evecs(np.array([[-0.980, 0.197], [-0.197, -0.980]]))

c2.set_evals(np.array([0.308, 17.743]))
c2.set_evecs(np.array([[0.951, 0.308], [-0.308, 0.951]]))
```

```python
# printing results
for cls in cls_labels:
    print(f'{cls} eigenvalues:')
    print(dataset[cls].eval)
    print('\n')
    print(f'{cls} eigenvectors:')
    print(dataset[cls].evec)
    print('\n')
```

```
c1 eigenvalues:
[ 0.609 14.35 ]


c1 eigenvectors:
[[-0.98   0.197]
 [-0.197 -0.98 ]]


c2 eigenvalues:
[ 0.308 17.743]


c2 eigenvectors:
[[ 0.951  0.308]
 [-0.308  0.951]]
```



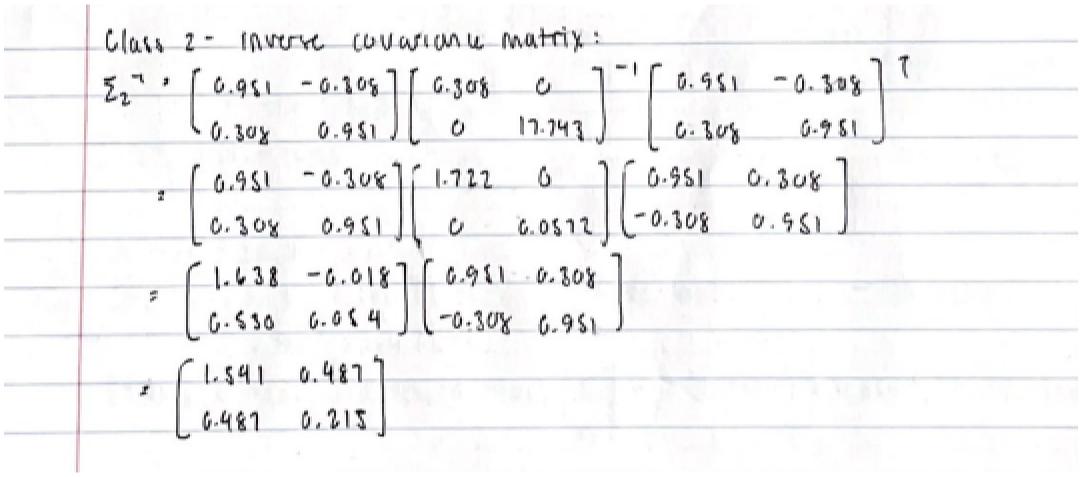
Class 1 - inverse covariance matrix:

$$\Sigma_1^{-1} = \begin{bmatrix} -0.980 & -0.197 \\ 0.197 & -0.980 \end{bmatrix} \begin{bmatrix} 0.609 & 0 \\ 0 & 14.350 \end{bmatrix}^{-1} \begin{bmatrix} -0.980 & -0.197 \\ 0.197 & -0.980 \end{bmatrix}^{T}$$

$$= \begin{bmatrix} -0.980 & -0.197 \\ 0.197 & -0.980 \end{bmatrix} \begin{bmatrix} 1.643 & 0 \\ 0 & 0.070 \end{bmatrix} \begin{bmatrix} -0.980 & 0.197 \\ -0.197 & -0.980 \end{bmatrix}$$

$$= \begin{bmatrix} -1.610 & -0.01379 \\ 0.324 & -0.0686 \end{bmatrix} \begin{bmatrix} -0.980 & 0.197 \\ -0.197 & -0.980 \end{bmatrix}$$

$$= \begin{bmatrix} 1.591 & -0.304 \\ -0.304 & 0.131 \end{bmatrix}$$



Class 2 - inverse covariance matrix:

$$\Sigma_2^{-1} = \begin{bmatrix} 0.951 & -0.308 \\ 0.308 & 0.951 \end{bmatrix} \begin{bmatrix} 0.308 & 0 \\ 0 & 17.743 \end{bmatrix}^{-1} \begin{bmatrix} 0.951 & -0.308 \\ 0.308 & 0.951 \end{bmatrix}^{T}$$

$$= \begin{bmatrix} 0.951 & -0.308 \\ 0.308 & 0.951 \end{bmatrix} \begin{bmatrix} 1.722 & 0 \\ 0 & 0.0572 \end{bmatrix} \begin{bmatrix} 0.951 & 0.308 \\ -0.308 & 0.951 \end{bmatrix}$$

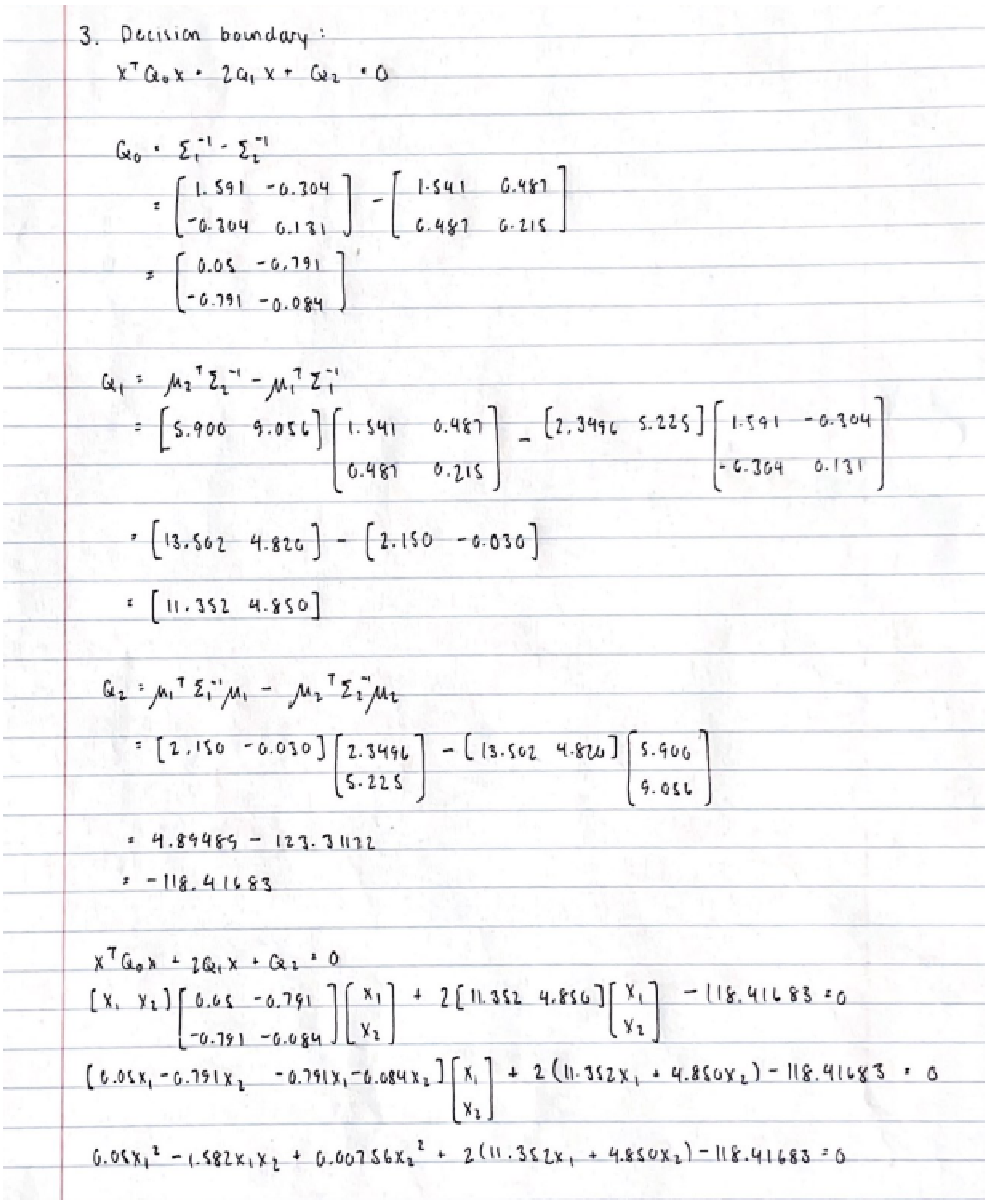
$$= \begin{bmatrix} 1.638 & -0.018 \\ 0.530 & 0.054 \end{bmatrix} \begin{bmatrix} 0.951 & 0.308 \\ -0.308 & 0.951 \end{bmatrix}$$

$$= \begin{bmatrix} 1.541 & 0.487 \\ 0.487 & 0.215 \end{bmatrix}$$

[7]:
```
# setting inverse covariance matrices from hand calculations
c1.set_cov_inverse(np.array([[1.591, -0.304], [-0.304, 0.131]]))
c2.set_cov_inverse(np.array([[1.541, 0.487], [0.487, 0.215]]))

# printing results
for cls in cls_labels:
    print(f'{cls} inverse covariance matrix:')
    print(dataset[cls].cov_inverse)
    print('\n')
```

```
c1 inverse covariance matrix:
[[ 1.591 -0.304]
 [-0.304  0.131]]


c2 inverse covariance matrix:
[[1.541 0.487]
 [0.487 0.215]]
```

Question 3



3. Decision boundary:

$x^T Q_0 x + 2Q_1 x + Q_2 = 0$

$Q_0 = \Sigma_1^{-1} - \Sigma_2^{-1}$

$= \begin{bmatrix} 1.591 & -0.304 \\ -0.304 & 0.131 \end{bmatrix} - \begin{bmatrix} 1.541 & 0.487 \\ 0.487 & 0.215 \end{bmatrix}$

$= \begin{bmatrix} 0.05 & -0.791 \\ -0.791 & -0.084 \end{bmatrix}$

$Q_1 = \mu_2^T \Sigma_2^{-1} - \mu_1^T \Sigma_1^{-1}$

$= \begin{bmatrix} 5.900 & 9.056 \end{bmatrix} \begin{bmatrix} 1.541 & 0.487 \\ 0.487 & 0.215 \end{bmatrix} - \begin{bmatrix} 2.3496 & 5.225 \end{bmatrix} \begin{bmatrix} 1.591 & -0.304 \\ -0.304 & 0.131 \end{bmatrix}$

$= \begin{bmatrix} 13.502 & 4.820 \end{bmatrix} - \begin{bmatrix} 2.150 & -0.030 \end{bmatrix}$

$= \begin{bmatrix} 11.352 & 4.850 \end{bmatrix}$

$Q_2 = \mu_1^T \Sigma_1^{-1} \mu_1 - \mu_2^T \Sigma_2^{-1} \mu_2$

$= \begin{bmatrix} 2.150 & -0.030 \end{bmatrix} \begin{bmatrix} 2.3496 \\ 5.225 \end{bmatrix} - \begin{bmatrix} 13.502 & 4.820 \end{bmatrix} \begin{bmatrix} 5.900 \\ 9.056 \end{bmatrix}$

$= 4.89485 - 123.31132$

$= -118.41683$

$x^T Q_0 x + 2Q_1 x + Q_2 = 0$

$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 0.05 & -0.791 \\ -0.791 & -0.084 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 2 \begin{bmatrix} 11.352 & 4.850 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 118.41683 = 0$

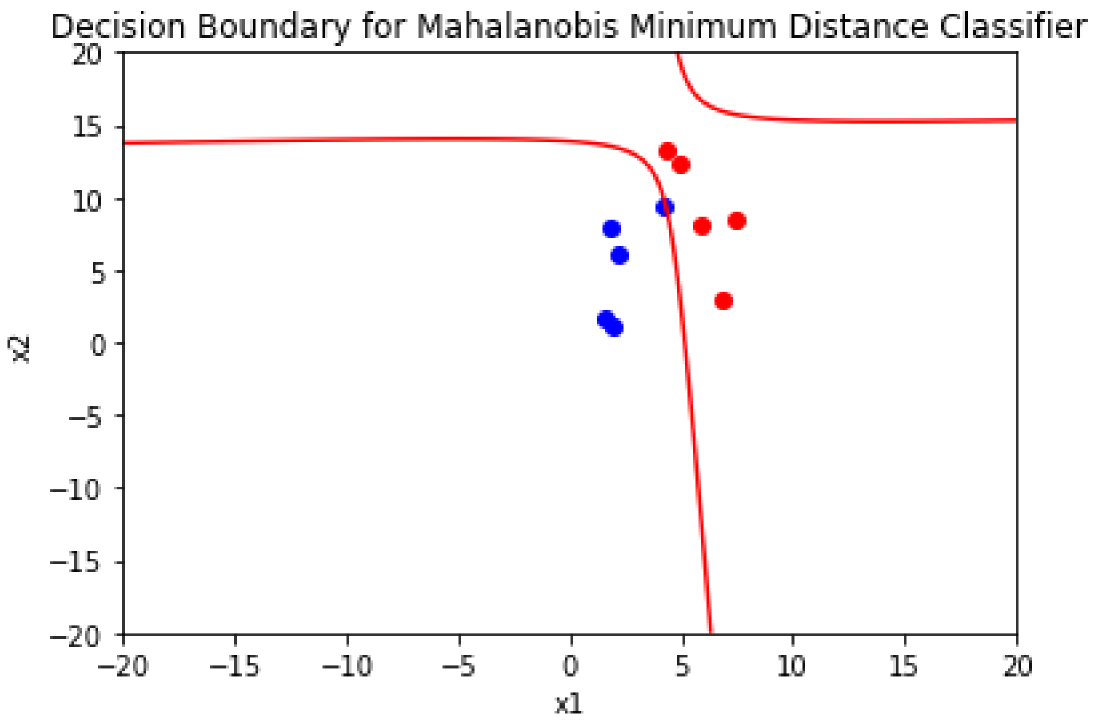
$\begin{bmatrix} 0.05x_1 - 0.791x_2 & -0.791x_1 - 0.084x_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 2(11.352x_1 + 4.850x_2) - 118.41683 = 0$

$0.05x_1^2 - 1.582x_1x_2 + 0.00756x_2^2 + 2(11.352x_1 + 4.850x_2) - 118.41683 = 0$

```
[8]: # calculating decision boundary from hand calculations
     q0 = np.array([[0.05, -0.791], [-0.791, -0.084]])
     q1 = np.array([11.352, 4.850])
     q2 = -118.41683

     # printing results
     print(f'q0 = {q0}')
     print(f'q1 = {q1}')
print(f'q2 = {q2}')
```

```
q0 = [[ 0.05  -0.791]
 [-0.791 -0.084]]
q1 = [11.352  4.85 ]
q2 = -118.41683
```

[9]:
```
def decision_boundary(x1, x2):
    return q0[0,0]*x1**2 + 2*q0[0,1]*x1*x2 + q0[1,1]*x2**2 + 2*(q1[0]*x1 +␣
 ↪q1[1]*x2) + q2

x1 = np.linspace(-20, 20, 400)
x2 = np.linspace(-20, 20, 400)
x1, x2 = np.meshgrid(x1, x2)
z = decision_boundary(x1, x2)

# plot decision boundary
plt.contour(x1, x2, z, levels=[0], colors='r')

# scatter data points
plt.scatter(c1.samples[:,0], c1.samples[:,1], color='blue')
plt.scatter(c2.samples[:,0], c2.samples[:,1], color='red')

# labels
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Decision Boundary for Mahalanobis Minimum Distance Classifier')

plt.show()
```



Question 4

```
[10]: class category:
          def __init__(self, mean, cov):
              self.mean = mean
              self.cov = cov
              self.samples = None
              self.sample_mean = None
              self.sample_cov = None
              self.eval = None
              self.evec = None
              self.cov_inverse = None

          def generate_samples(self, N):
              self.samples = np.random.multivariate_normal(self.mean, self.cov, N)

          def add_noise(self, mean, cov):
              for idx, _ in enumerate(self.samples):
                  noise = np.random.multivariate_normal(mean, cov, 1)
                  self.samples[idx] = self.samples[idx] + noise

          # for 100 samples
          def compute_smean(self):
              # sample mean
              self.sample_mean = np.array([np.mean(self.samples[:,0]), np.mean(self.
       ↪samples[:,1])])
```

```
    def compute_scov(self):
        # sample covariance
        self.sample_cov = np.cov(self.samples, rowvar=False)

    def compute_epairs(self):
        self.eval, self.evec = np.linalg.eig(self.sample_cov)

    def compute_cov_inverse(self):
        self.cov_inverse = np.linalg.inv(self.sample_cov)
```

[11]:
```
# generating 50 random samples for two classes
c1 = category(np.array([1, 3]), np.array([[1, 0], [0, 15]]))
c2 = category(np.array([4, 7]), np.array([[3, 4], [4, 11]]))

cls_labels = ['c1', 'c2']
dataset = {'c1': c1, 'c2': c2}

np.random.seed(42)
for cls in cls_labels:
    dataset[cls].generate_samples(50)
    dataset[cls].add_noise(np.array([2, 2]), np.array([[2, 0], [0, 3]]))
    dataset[cls].compute_smean()
    dataset[cls].compute_scov()
```

[12]:
```
# printing results
for cls in cls_labels:
    print(f'{cls} sample mean:')
    print(dataset[cls].sample_mean)
    print('\n')
    print(f'{cls} sample covariance matrix:')
    print(dataset[cls].sample_cov)
    print('\n')
```

```
c1 sample mean:
[3.12606025 4.30920131]


c1 sample covariance matrix:
[[ 2.3386168   0.21137213]
 [ 0.21137213 12.71368466]]


c2 sample mean:
[5.9453451  8.79747236]


c2 sample covariance matrix:
```

```
[[ 4.58895613  4.37160008]
 [ 4.37160008 17.43386725]]
```

[13]:
```
# computing eigenvalues and eigenvectors for each class
for cls in cls_labels:
    dataset[cls].compute_epairs()

# printing results
for cls in cls_labels:
    print(f'{cls} eigenvalues:')
    print(dataset[cls].eval)
    print('\n')
    print(f'{cls} eigenvectors:')
    print(dataset[cls].evec)
    print('\n')
```

```
c1 eigenvalues:
[ 2.33431229 12.71798918]


c1 eigenvectors:
[[-0.99979271 -0.02036041]
 [ 0.02036041 -0.99979271]]


c2 eigenvalues:
[ 3.24231762 18.78050576]


c2 eigenvectors:
[[-0.9556849  -0.29439154]
 [ 0.29439154 -0.9556849 ]]
```

[14]:
```
# computing inverse covariance using eigenvalues and eigenvectors
for cls in cls_labels:
    dataset[cls].compute_cov_inverse()

# printing results
for cls in cls_labels:
    print(f'{cls} inverse covariance matrix:')
    print(dataset[cls].cov_inverse)
    print('\n')
```

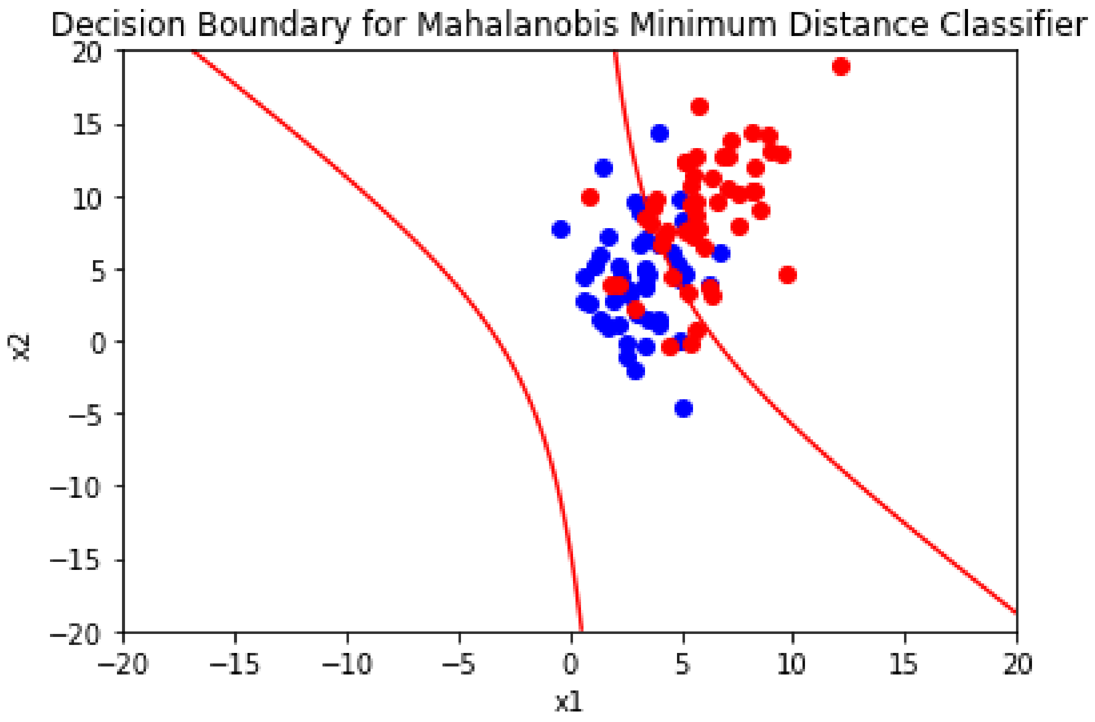
```
c1 inverse covariance matrix:
[[ 0.4282467  -0.00711984]
```

```
 [-0.00711984  0.07877378]]


c2 inverse covariance matrix:
[[ 0.28630629 -0.07179225]
 [-0.07179225  0.07536177]]
```

[15]:
```
# calculating decision boundary
q0 = c1.cov_inverse - c2.cov_inverse
q1 = (c2.sample_mean.T @ c2.cov_inverse) - (c1.sample_mean.T @ c1.cov_inverse)
q2 = (c1.sample_mean.T @ c1.cov_inverse) @ c1.sample_mean - (c2.sample_mean.T @␣
 ↪c2.cov_inverse) @ c2.sample_mean

# printing results
print(f'q0 = {q0}')
print(f'q1 = {q1}')
print(f'q2 = {q2}')
```

```
q0 = [[0.14194041 0.06467241]
 [0.06467241 0.00341201]]
q1 = [-0.23744481 -0.08103167]
q2 = -2.9868410040882827
```

[16]:
```
def decision_boundary(x1, x2):
    return q0[0,0]*x1**2 + 2*q0[0,1]*x1*x2 + q0[1,1]*x2**2 + 2*(q1[0]*x1 +␣
 ↪q1[1]*x2) + q2

x1 = np.linspace(-20, 20, 400)
x2 = np.linspace(-20, 20, 400)
x1, x2 = np.meshgrid(x1, x2)
z = decision_boundary(x1, x2)

# plot decision boundary
plt.contour(x1, x2, z, levels=[0], colors='r')

# scatter data points
plt.scatter(c1.samples[:,0], c1.samples[:,1], color='blue')
plt.scatter(c2.samples[:,0], c2.samples[:,1], color='red')

# labels
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Decision Boundary for Mahalanobis Minimum Distance Classifier')

plt.show()
```



Question 5

[17]:
```
# initializing two labelled test classes
t1 = category(np.array([1, 3]), np.array([[1, 0], [0, 15]]))
t2 = category(np.array([4, 7]), np.array([[3, 4], [4, 11]]))

cls_labels = ['t1', 't2']
dataset = {'t1': t1, 't2': t2}

N = 50
np.random.seed(24)
for cls in cls_labels:
    dataset[cls].generate_samples(N)
    dataset[cls].compute_smean()

    # create white noise with mean 0 and identity covariance matrix
    noise_mean = np.array([0, 0])
    noise_cov = np.identity(2)
    noise_matrix = np.random.multivariate_normal(noise_mean, noise_cov, 50)

    # add white noise to samples
    dataset[cls].samples = dataset[cls].samples + noise_matrix

# combine noisy samples for both classes
noisy_samples = np.vstack((t1.samples, t2.samples))
```

```
# labels for the noisy samples (0 for class c1, 1 for class c2)
labels = np.concatenate((np.zeros(N), np.ones(N))).astype(int)
```

```
[18]: # MMD classifier
def classify(sample, coefs):
    x1, x2 = sample
    q0, q1, q2 = coefs
    decision_boundary = q0[0,0]*x1**2 + 2*q0[0,1]*x1*x2 + q0[1,1]*x2**2 +␣
 ↪2*(q1[0]*x1 + q1[1]*x2) + q2

    # if inside hyperbola, c1, otherwise, c2
    if decision_boundary < 0:
        return 0
    else:
        return 1

# 10 sample dataset
q0 = np.array([[0.05, -0.791], [-0.791, -0.084]])
q1 = np.array([11.352, 4.850])
q2 = -118.41683

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample, [q0, q1, q2]))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"10-sample MMD: Accuracy = {accuracy * 100:.2f}%")

# 100 sample dataset
q0 = np.array([[0.14194041, 0.06467241], [0.06467241, 0.00341201]])
q1 = np.array([-0.23744481, -0.08103167])
q2 = -2.9868410040882827

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample, [q0, q1, q2]))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"100-sample MMD: Accuracy = {accuracy * 100:.2f}%")
```

```
10-sample MMD: Accuracy = 71.00%
100-sample MMD: Accuracy = 77.00%
```

Question 6

Which of the two classifiers is better, the one learned from 10 samples or the one with 100 samples? Why?

The classifier trained on 100 samples performs better than the one trained on 10 samples because larger sample size better represents the true distribution of the data, therefore the decision boundary generalizes better to unseen data.

Question 7

[19]:
```
# MED classifer
def classify(sample):
    boundary_vector = c2.sample_mean - c1.sample_mean
    y = (boundary_vector.T @ sample) + 0.5*((c1.sample_mean @ c1.sample_mean.T)␣
 ↪- (c2.sample_mean @ c2.sample_mean.T))

    if y > 0:
        return 1
    else:
        return 0

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"100-sample MED: Accuracy = {accuracy * 100:.2f}%")
```

```
100-sample MED: Accuracy = 74.00%
```

Is MMD better than MED? Explain.

10-sample MMD: Accuracy = 71.00% 100-sample MMD: Accuracy = 77.00% 100-sample MED: Accuracy = 74.00%

The performance of the 100-sample MED classifier is close to but slightly worse than the performance of the 100-sample MMD classifier, having a lower accuracy by 3%. This is reasonable given the nature of the two classes, which overlap. We could expect that the linear MED classifier decision boundary is not significantly different from a tangent line at points on the MMD curve close to the centroid of the whole dataset. Therefore, the accuracy is similar. In general, whether MMD is better than MED depends on the dataset. For example, MMD would perform better in a case where the two classes existed in two adjacent crescents. A hyperbolic decision boundary could fit better to the curve adjacent to both classes than a linear decision boundary.

0.0.2 Exercise 2: ML and MAP Classifiers

Question 1

[20]:
```
# generating 50 random samples for two classes
c1 = category(np.array([1, 3]), np.array([[1, 0], [0, 15]]))
c2 = category(np.array([4, 7]), np.array([[3, 4], [4, 11]]))
```

```
cls_labels = ['c1', 'c2']
dataset = {'c1': c1, 'c2': c2}

np.random.seed(42)
for cls in cls_labels:
    dataset[cls].generate_samples(50)
    dataset[cls].add_noise(np.array([2, 2]), np.array([[2, 0], [0, 3]]))
    dataset[cls].compute_smean()
    dataset[cls].compute_scov()
    dataset[cls].compute_cov_inverse()
```

[21]:
```
# initializing two labelled test classes
t1 = category(np.array([1, 3]), np.array([[1, 0], [0, 15]]))
t2 = category(np.array([4, 7]), np.array([[3, 4], [4, 11]]))

cls_labels = ['t1', 't2']
dataset = {'t1': t1, 't2': t2}

N = 50
np.random.seed(24)
for cls in cls_labels:
    dataset[cls].generate_samples(N)
    dataset[cls].compute_smean()

    # create white noise with mean 0 and identity covariance matrix
    noise_mean = np.array([0, 0])
    noise_cov = np.identity(2)
    noise_matrix = np.random.multivariate_normal(noise_mean, noise_cov, 50)

    # add white noise to samples
    dataset[cls].samples = dataset[cls].samples + noise_matrix

# combine noisy samples for both classes
noisy_samples = np.vstack((t1.samples, t2.samples))

# labels for the noisy samples (0 for class c1, 1 for class c2)
labels = np.concatenate((np.zeros(N), np.ones(N))).astype(int)
```

[22]:
```
# ML classifier

eta = 2*np.log(np.sqrt(np.linalg.det(c1.sample_cov))/np.sqrt(np.linalg.det(c2.
 ↪sample_cov)))
# print(eta)

def classify(sample):
    # calculate the likelihood ratio using multivariate Gaussian PDF
    mahalanobis_c1 = ((sample - c1.sample_mean).T @ c1.cov_inverse) @ (sample -␣
 ↪c1.sample_mean)
    mahalanobis_c2 = ((sample - c2.sample_mean).T @ c2.cov_inverse) @ (sample -␣
 ↪c2.sample_mean)

    decision = mahalanobis_c1 - mahalanobis_c2

    if decision < eta:
        return 0
    else:
        return 1

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"ML Accuracy = {accuracy * 100:.2f}%")
```

```
ML Accuracy = 81.00%
```

Question 2

```
[23]: # MAP classifier

p_c1 = 0.58
p_c2 = 0.42
eta = 2*np.log(np.sqrt(np.linalg.det(c1.sample_cov))/np.sqrt(np.linalg.det(c2.
 ↪sample_cov))) + 2*np.log(p_c1/p_c2)
# print(eta)

def classify(sample):
    # calculate the likelihood ratio using multivariate Gaussian PDF
    mahalanobis_c1 = ((sample - c1.sample_mean).T @ c1.cov_inverse) @ (sample -␣
 ↪c1.sample_mean)
    mahalanobis_c2 = ((sample - c2.sample_mean).T @ c2.cov_inverse) @ (sample -␣
 ↪c2.sample_mean)

    decision = mahalanobis_c1 - mahalanobis_c2

    if decision < eta:
        return 0
    else:
        return 1

# classify noisy samples
```

```
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"MAP Accuracy = {accuracy * 100:.2f}%")
```

```
MAP Accuracy = 78.00%
```

Question 3

Based on the results, do you think that assuming the probability distributions of the two classes as Gaussian was correct? What if you remove the extra step of adding noise in your training data (step 1 exercise 1)? Do you think that would make these classifiers better? Explain.

Given that the `np.random.multivariate_normal` function random draws samples from a normal distribution, it was appropriate to assume a Gaussian distribution for the ML and MAP classifiers. It cannot be confirmed from the classification accuracy alone, but the relatively high accuracy scores for ML and MAP suggest that the data may be Gaussian, since they are built on the assumption of Gaussian data, unlike the MMD and MED classifiers. If noise were not added to the data, I would expect the performance of the ML and MAP classifiers to improve slightly, as the addition of noise makes data less close to a true Gaussian distribution.

Question 4

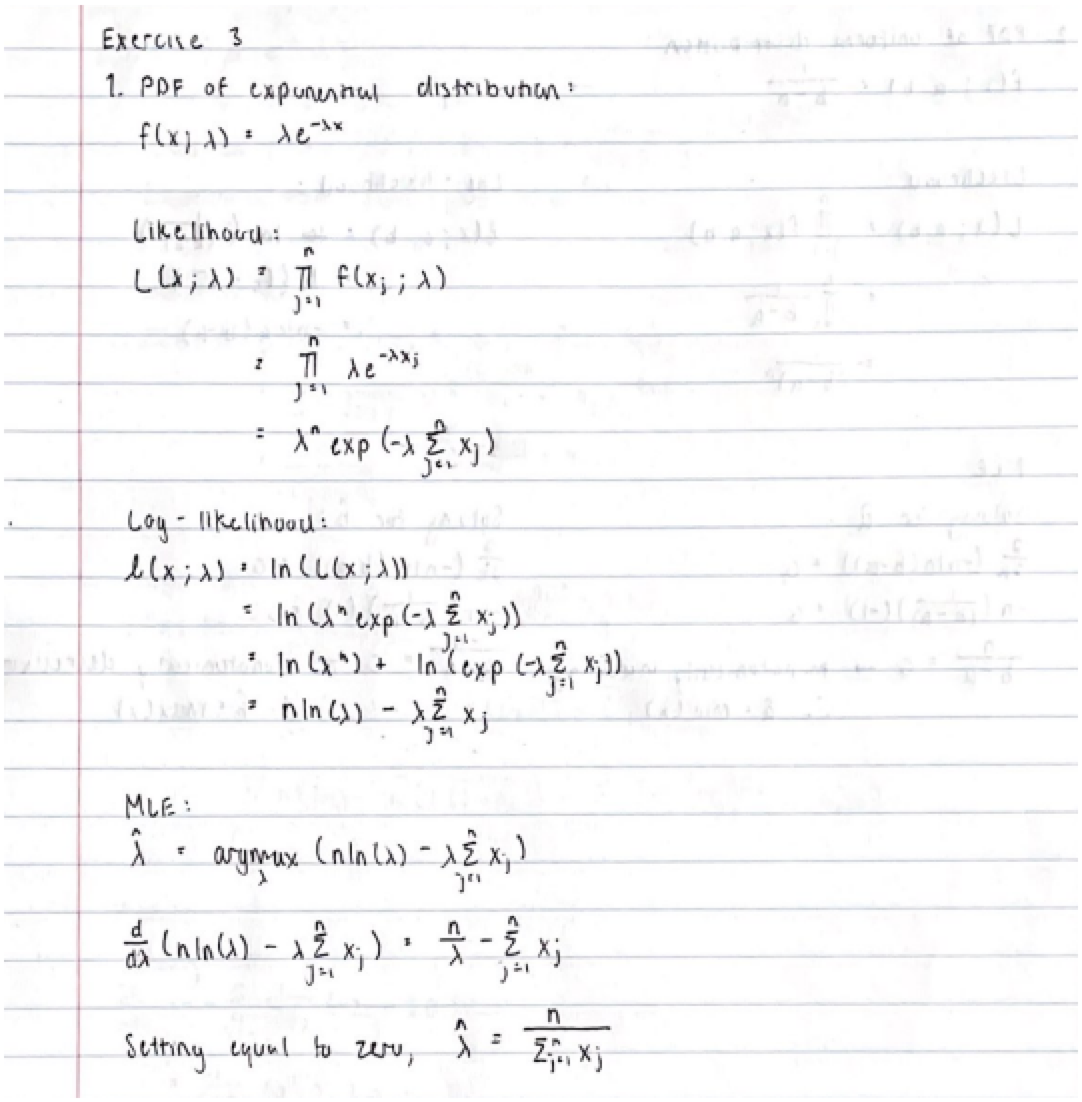
Compare the ML, MAP, MMD, and MED Classifiers based on classification accuracy. Which classifier is the best? Could the inferior classifiers be better for different data? Explain.

Summarizing results, MMD Accuracy = 77.00% MED Accuracy = 74.00% ML Accuracy = 81.00% MAP Accuracy = 78.00%

For this data, it appears that the ML classifer performs the best, while the MMD and MAP classifiers are close and comparable. The MED classifier has the lowest accuracy. The performances of these models are dependent on the nature of data being classified. For example, the MED classifier would perform better than the others in a case where the data is well separated and does not have significantly different covariances. The MMD, ML, and MAP classifiers assume more complex covariance structures, which might affect accuracy. Also, in such a case, the MED classifier would be the most efficient in terms of computation, which is also a factor to consider when comparing the models.

0.0.3 Exercise 3: Parametric Estimation

Question 1


Exercise 3

1. PDF of exponential distribution:

$f(x;\lambda) = \lambda e^{-\lambda x}$

Likelihood:

$L(x;\lambda) = \prod_{j=1}^{n} f(x_j;\lambda)$

$= \prod_{j=1}^{n} \lambda e^{-\lambda x_j}$

$= \lambda^n \exp(-\lambda \sum_{j=1}^{n} x_j)$

Log-likelihood:

$\ell(x;\lambda) = \ln(L(x;\lambda))$

$= \ln(\lambda^n \exp(-\lambda \sum_{j=1}^{n} x_j))$

$= \ln(\lambda^n) + \ln(\exp(-\lambda \sum_{j=1}^{n} x_j))$

$= n\ln(\lambda) - \lambda \sum_{j=1}^{n} x_j$

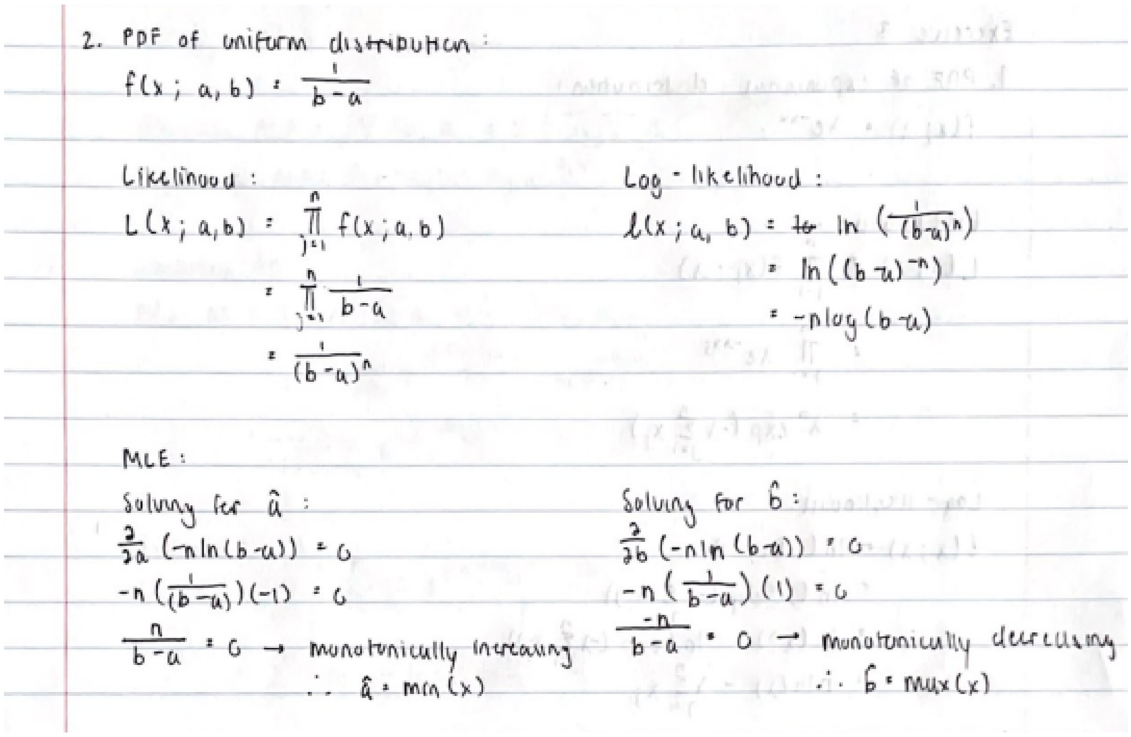
MLE:

$\hat{\lambda} = \underset{\lambda}{\operatorname{argmax}} (n\ln(\lambda) - \lambda \sum_{j=1}^{n} x_j)$

$\frac{d}{d\lambda}(n\ln(\lambda) - \lambda \sum_{j=1}^{n} x_j) = \frac{n}{\lambda} - \sum_{j=1}^{n} x_j$

Setting equal to zero, $\hat{\lambda} = \frac{n}{\sum_{j=1}^{n} x_j}$

Question 2



2. PDF of uniform distribution:

$f(x; a, b) = \frac{1}{b-a}$

Likelihood:

$L(x; a, b) = \prod_{j=1}^{n} f(x; a, b)$

$= \prod_{j=1}^{n} \frac{1}{b-a}$

$= \frac{1}{(b-a)^n}$

Log-likelihood:

$\ell(x; a, b) = \cancel{\text{lo}} \ \ln\left(\frac{1}{(b-a)^n}\right)$

$= \ln\left((b-a)^{-n}\right)$

$= -n\log(b-a)$

MLE:

Solving for $\hat{a}$:

$\frac{\partial}{\partial a}\left(-n\ln(b-a)\right) = 0$

$-n\left(\frac{1}{(b-a)}\right)(-1) = 0$

$\frac{n}{b-a} = 0 \rightarrow$ monotonically increasing

$\therefore \hat{a} = \min(x)$

Solving for $\hat{b}$:

$\frac{\partial}{\partial b}\left(-n\ln(b-a)\right) = 0$

$-n\left(\frac{1}{b-a}\right)(1) = 0$

$\frac{-n}{b-a} = 0 \rightarrow$ monotonically decreasing

$\therefore \hat{b} = \max(x)$

Question 3

3. $\theta = [\overset{\theta_1}{\overset{\uparrow}{\mu}} \ \overset{\theta_2}{\overset{\uparrow}{\sigma^2}} \ \overset{\theta_3}{\overset{\uparrow}{\lambda}}]$

Gaussian PDF : $\mathcal{N}(x; \theta_1, \theta_2) = \frac{1}{\sqrt{2\pi\theta_2}} e^{\frac{-1}{2}\frac{(x-\theta_1)^2}{\theta_2}}$

Exponential PDF : $f(x; \theta_3) = \theta_3 e^{-\theta_3 x}$

Combined PDF :

$$P(x;\theta) = \frac{1}{2} \cdot \mathcal{N}(x; \theta_1, \theta_2) \cdot f(x;\theta_3)$$

$$= \frac{1}{2} \cdot \frac{1}{\sqrt{2\pi\theta_2}} e^{\frac{-1}{2}\frac{(x-\theta_1)^2}{\theta_2}} \cdot \theta_3 e^{-\theta_3 x}$$

$$= \frac{\theta_3}{2\sqrt{2\pi\theta_2}} e^{\left(\frac{-1}{2}\frac{(x-\theta_1)^2}{\theta_2} - \theta_3 x\right)}$$

Likelihood :

$$L(x;\theta) = \prod_{j=1}^{n} \left(\frac{\theta_3}{2\sqrt{2\pi\theta_2}} e^{\left(\frac{-1}{2}\frac{(x_j-\theta_1)^2}{\theta_2} - \theta_3 x_j\right)} \right)$$

$$= \sum_{j=1}^{n} \left(\ln\lambda - \ln(2\sqrt{2\pi\theta_2}) + \left(\frac{-1}{2}\frac{(x_j-\theta_1)^2}{\theta_2} - \theta_3 x_j\right)\right)$$

$$= n(\ln\lambda - \ln(2\sqrt{2\pi\theta_2})) - \sum_{j=1}^{n} \left(\frac{1}{2}\frac{(x_j-\theta_1)^2}{\theta_2} + \theta_3 x_j\right)$$

MLE:

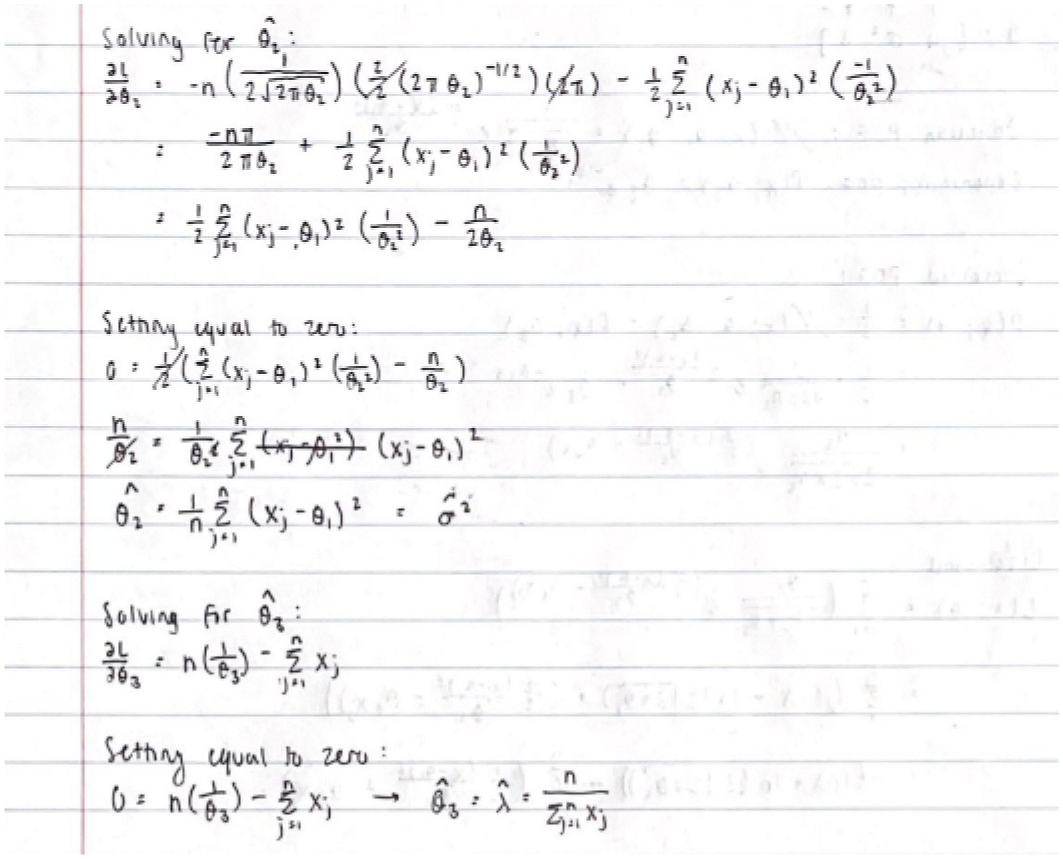
Solving for $\hat{\theta}_1$:

$$\frac{\partial L}{\partial \theta_1} = -\sum_{j=1}^{n} \frac{1}{2\theta_2}(-2x_j + 2\theta_1)$$

$$= \frac{\cancel{-1}}{\cancel{2}\theta_2} (\cancel{-2}) \sum_{j=1}^{n} (\theta_1 - x_j)$$

$$= \frac{1}{\theta_2}\left(n\theta_1 - \sum_{j=1}^{n} x_j\right)$$

Setting equal to zero, $\hat{\theta}_1 = \hat{\mu} = \frac{1}{n}\sum_{j=1}^{n} x_j$



Solving for $\hat{\theta}_2$:

$$\frac{\partial L}{\partial \theta_2} = -n\left(\frac{1}{2\sqrt{2\pi\theta_2}}\right)\left(\frac{\cancel{2}}{\cancel{2}}(2\pi\theta_2)^{-1/2}\right)(\cancel{2}\pi) - \frac{1}{2}\sum_{j=1}^{n}(x_j-\theta_1)^2\left(\frac{-1}{\theta_2^2}\right)$$

$$= \frac{-n\pi}{2\pi\theta_2} + \frac{1}{2}\sum_{j=1}^{n}(x_j-\theta_1)^2\left(\frac{1}{\theta_2^2}\right)$$

$$= \frac{1}{2}\sum_{j=1}^{n}(x_j-\theta_1)^2\left(\frac{1}{\theta_2^2}\right) - \frac{n}{2\theta_2}$$

Setting equal to zero:

$$0 = \cancel{\frac{1}{2}}\left(\sum_{j=1}^{n}(x_j-\theta_1)^2\left(\frac{1}{\theta_2^2}\right) - \frac{n}{\theta_2}\right)$$

$$\frac{n}{\cancel{\theta_2}} = \frac{1}{\theta_2^{\cancel{2}}}\sum_{j=1}^{n}\cancel{(x_j-\theta_1^2)}\,(x_j-\theta_1)^2$$

$$\hat{\theta}_2 = \frac{1}{n}\sum_{j=1}^{n}(x_j-\theta_1)^2 = \hat{\sigma}^2$$

Solving for $\hat{\theta}_3$:

$$\frac{\partial L}{\partial \theta_3} = n\left(\frac{1}{\theta_3}\right) - \sum_{j=1}^{n}x_j$$

Setting equal to zero:

$$0 = n\left(\frac{1}{\theta_3}\right) - \sum_{j=1}^{n}x_j \rightarrow \hat{\theta}_3 = \hat{\lambda} = \frac{n}{\sum_{j=1}^{n}x_j}$$

Question 4

[24]:
```python
# generate training data
N = 5
np.random.seed(42)
c1_training = np.random.normal(0.5, 1, N)
c2_training = np.random.normal(5, 3, N)

# generate testing data
cls_labels = ['c1', 'c2']
testing = {'c1': None, 'c2': None}

N = 50
np.random.seed(24)
testing['c1'] = np.random.normal(0.5, 1, N)
testing['c2'] = np.random.normal(5, 3, N)

for cls in cls_labels:
    # create white noise with mean 0 and identity covariance matrix
    noise_mean = 0
    noise_std = 1
    noise_matrix = np.random.normal(noise_mean, noise_std, N)
    # add white noise to samples
    testing[cls] = testing[cls] + noise_matrix

# combine noisy samples for both classes
noisy_samples = np.concatenate((testing['c1'], testing['c2']))

# labels for the noisy samples (0 for class 1, 1 for class 2)
labels = np.concatenate((np.zeros(N), np.ones(N))).astype(int)
```

[25]:
```
# exponential distribution

# compute estimators from training data
c1_lambd = 1/np.mean(c1_training)
c2_lambd = 1/np.mean(c2_training)

def classify(sample):
    c1_prob = c1_lambd*np.exp(-c1_lambd*sample)
    c2_prob = c2_lambd*np.exp(-c2_lambd*sample)

    if c1_prob > c2_prob:
        return 0
    else:
        return 1

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"Accuracy = {accuracy * 100:.2f}%")
```

```
Accuracy = 83.00%
```

[26]:
```
# uniform distribution

# compute estimators from training data
c1_a = min(c1_training)
c1_b = max(c1_training)
c2_a = min(c2_training)
c2_b = max(c2_training)

def classify(sample):
    c1_prob = 1/(c1_b - c1_a)
    c2_prob = 1/(c2_b - c2_a)

    if c1_prob > c2_prob:
        return 0
    else:
        return 1

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))

# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"Accuracy = {accuracy * 100:.2f}%")
```

```
Accuracy = 50.00%
```

[27]:
```
# gaussian x exponential distribution

# compute estimators from training data
c1_mean = np.mean(c1_training)
c1_lambd = 1/c1_mean
c1_var = 0
for _, x in enumerate(c1_training):
    c1_var += (x - c1_mean)**2
c1_var = c1_var/N

c2_mean = np.mean(c2_training)
c2_lambd = 1/c2_mean
c2_var = 0
for _, x in enumerate(c2_training):
    c2_var += (x - c2_mean)**2
c2_var = c2_var/N

def classify(sample):
    c1_prob = c1_lambd/(2*np.sqrt(2*np.pi*c1_var))*np.exp((-1/2)*((sample -␣
 ↪c1_mean)**2)/c1_var - c1_lambd*sample)
    c2_prob = c2_lambd/(2*np.sqrt(2*np.pi*c2_var))*np.exp((-1/2)*((sample -␣
 ↪c2_mean)**2)/c2_var - c2_lambd*sample)

    if c1_prob > c2_prob:
        return 0
    else:
        return 1

# classify noisy samples
predicted_labels = []
for sample in noisy_samples:
    predicted_labels.append(classify(sample))
```

```python
# calculate classification accuracy
accuracy = np.mean(labels == predicted_labels)
print(f"Accuracy = {accuracy * 100:.2f}%")
```

```
Accuracy = 80.00%
```

Which of the three ML classifiers is the best? Why? Of these three classifiers, the exponential distribution classifier performs the best, but the Gaussian x exponential distribution classifier is lower by only 3%. The uniform distribution classifier performs the most poorly, as it does not depend on the sample at all. The probabilities of the training classes are constants, and so the classifier always selects the class with the greater probability. Hence, its classification accuracy is 50%. I would have expected the Gaussian x exponential classifier to perform the best, fitting better to the Gaussian nature of the data. The discrepency may be attributed to the random variation of the dataset.