EECE5644 2021 Fall – Assignment 2

Balaji Sundareshan

Question 1

- In this question, two-dimensional samples were generated from the probability density function(PDF): p(x) = P(L = 0)p(x|L = 0) + P(L = 1)p(x|L = 1) where P(L = 0) and P(L = 1) are class priors and p(x|L = 0) and p(x|L = 1) are class conditional PDFs.

- The class-conditional pdfs are p(x|L = 0) = w1 * g(x|m01 , C01) + w2 * g(x|m02, C02) and p(x|L = 1) = g(x|m1 , C1), where g(x|m, C) is a multivariate Gaussian probability density function with mean vector m and covariance matrix C.

$$m_{01} = \begin{bmatrix}5\\0\end{bmatrix} \; C_{01} = \begin{bmatrix}4 & 0\\0 & 2\end{bmatrix} \, m_{02} = \begin{bmatrix}0\\4\end{bmatrix} \; C_{02} = \begin{bmatrix}1 & 0\\0 & 3\end{bmatrix} \, m_1 = \begin{bmatrix}3\\2\end{bmatrix} \; C_1 = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}$$

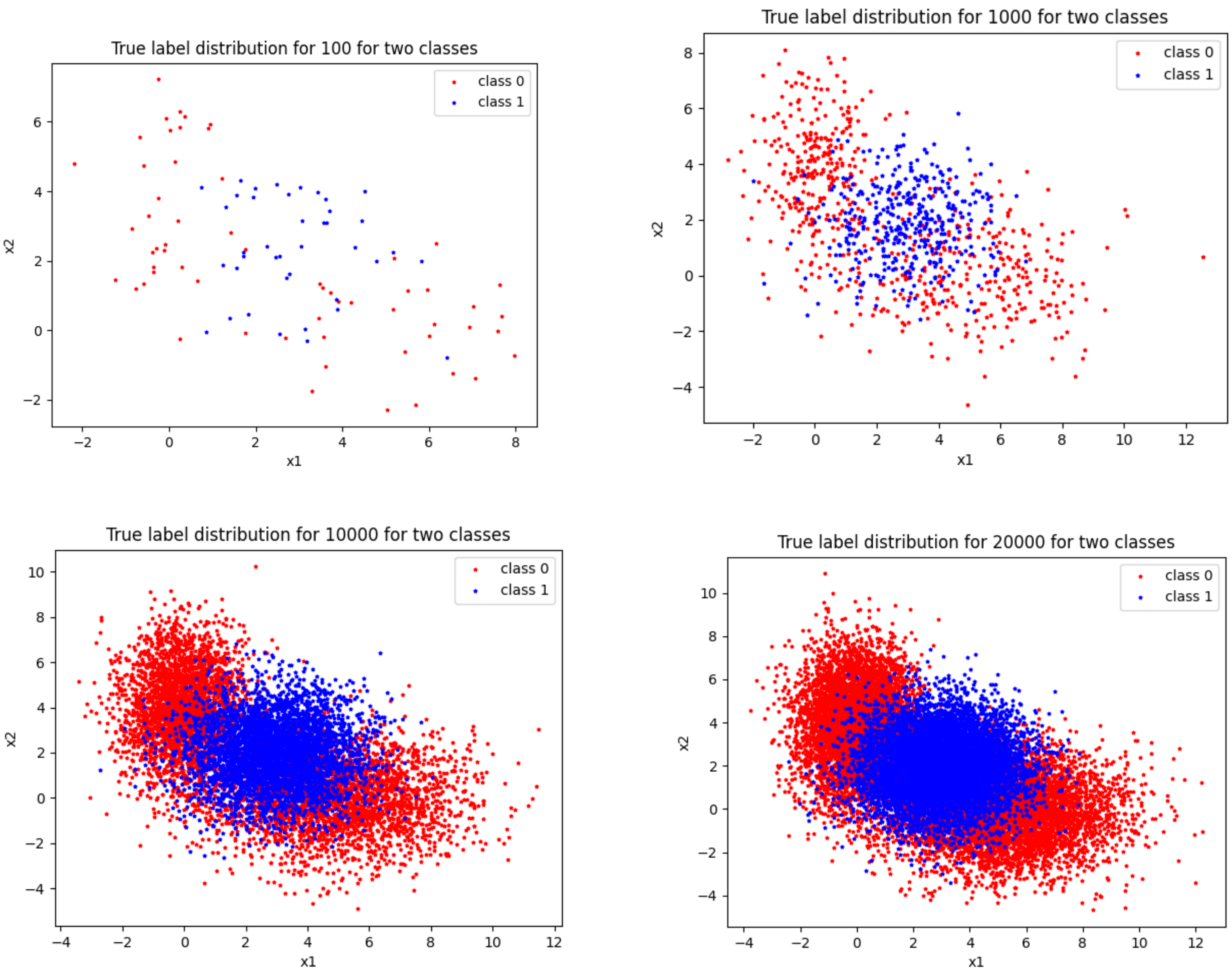
$$P(L=0) = 0.6 \; P(L=1) = 0.4$$

$$w_1 = 0.5 \; w_2 = 0.5$$

- For the following parts 4 sets of data were generated:

  - D^{100}_{train} consists of 100 data samples and their labels for training
  - D^{1000}_{train} consists of 1000 data samples and their labels for training
  - D^{10000}_{train} consists of 10000 data samples and their labels for training
  - $D^{20000}_{validate}$ consists of 20000 data samples and their labels for validation

- Plots for the datasets used are shown below


True label distribution for 100 for two classes
True label distribution for 1000 for two classes
True label distribution for 10000 for two classes
True label distribution for 20000 for two classes
class 0
class 1
x1
x2

Part A: Theoretically Optimal Classifier with Known Parameters

- For this part, knowledge of the true pdf was used to determine the theoretically optimal classifier. Minimum expected risk classification rule in the form of a likelihood-ratio test

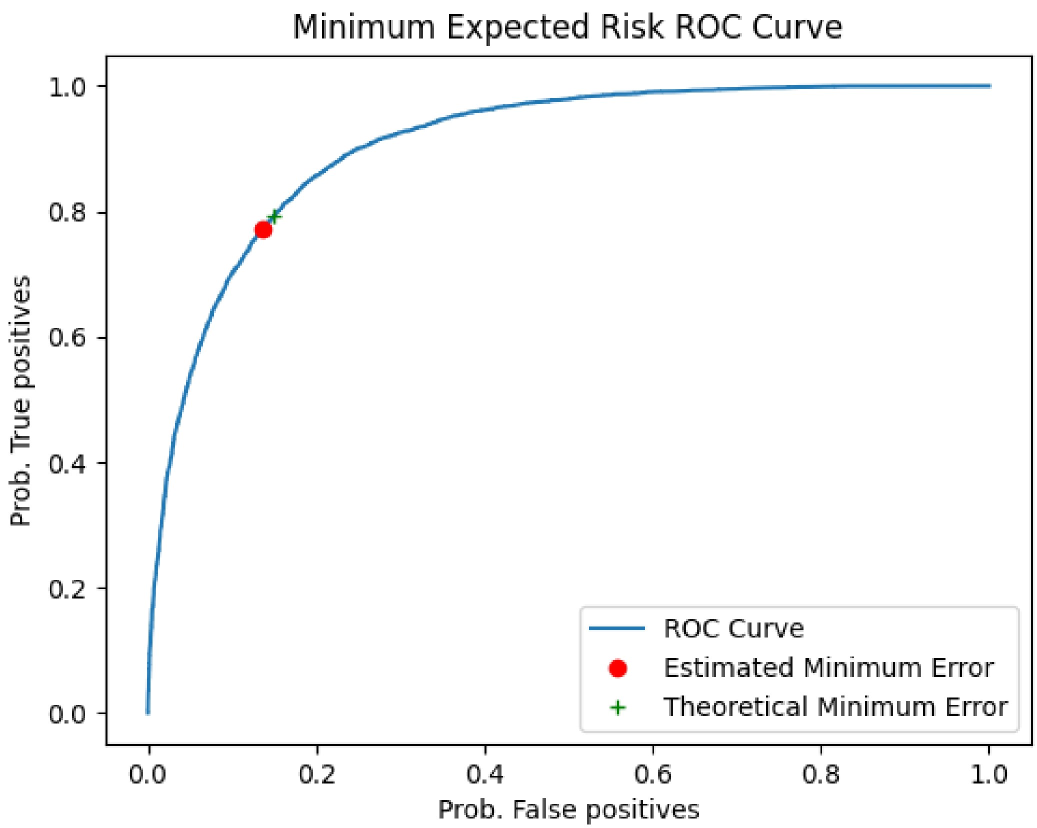
$$\frac{P(X|L=1)}{P(X|L=0)} \underset{D(x)=0}{\overset{D(x)=1}{\gtrless}} \frac{(\lambda_{10}-\lambda_{00})P(L=0)}{(\lambda_{01}-\lambda_{11})P(L=1)}$$

- In order to reduce the probability of misclassifications, the cost of incorrect classification should be 1 and the cost of correct classifications should be 0. For this case, the likelihood-ratio test is shown below

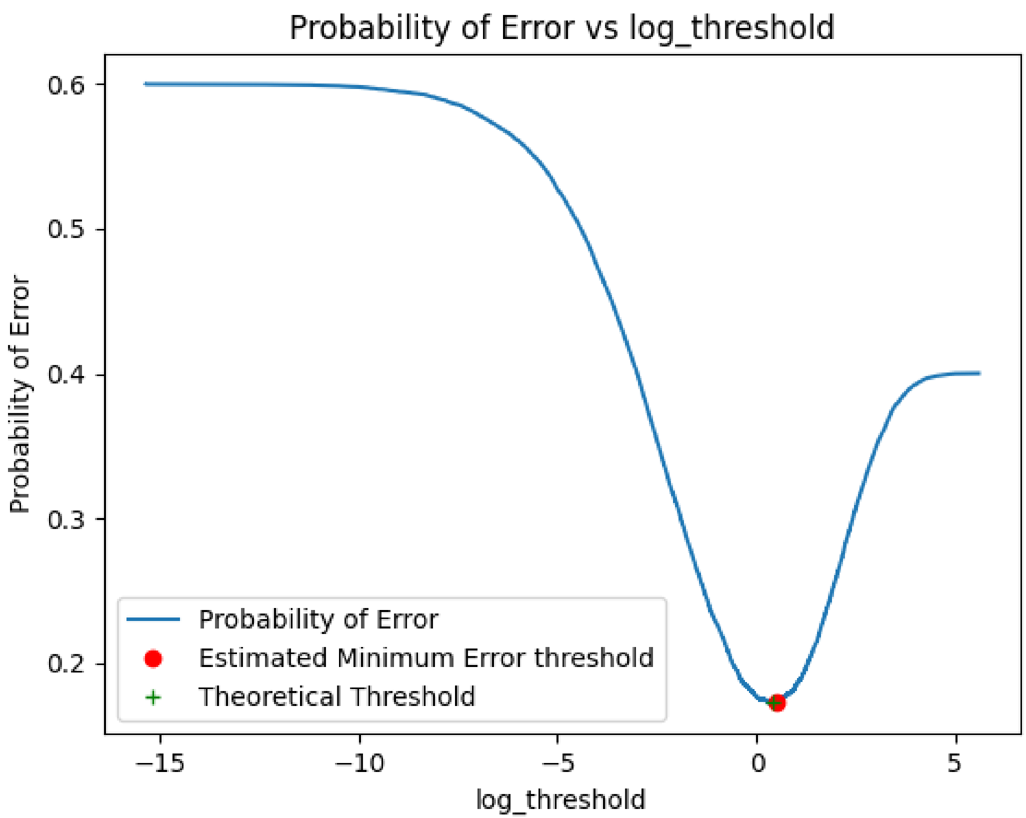
$$\frac{P(X|L=1)}{P(X|L=0)} \underset{D(x)=0}{\overset{D(x)=1}{\gtrless}} \frac{(1-0)*0.6}{(1-0)*0.4} = 1.5 = \gamma$$

- Parametric sweep was done for a range of threshold values and the classifier was found for each of these threshold values. True positives and False positives were calculated for each of these threshold values and the ROC curve was plotted. The plot is shown below.

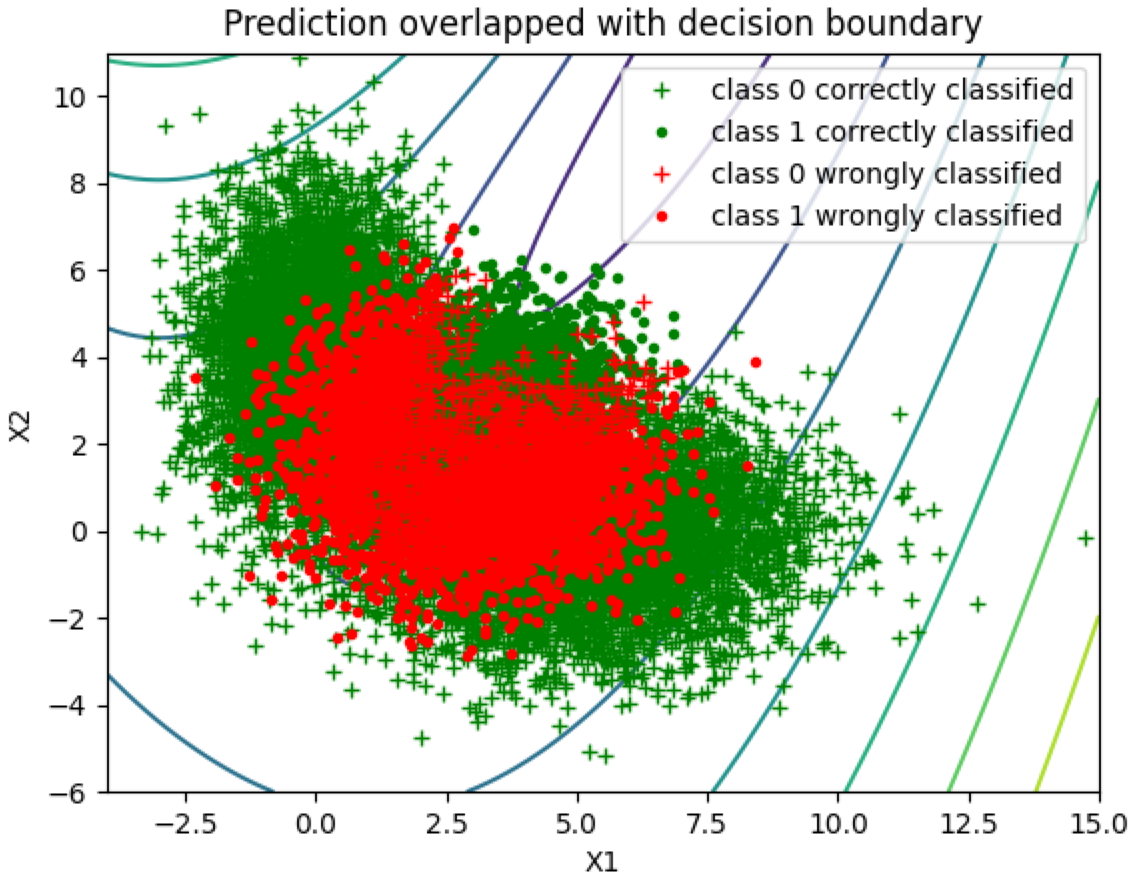
	Threshold value (γ)	Minimum probability of Error
Theoretical	1.5	17.28%
Estimated from Data	1.68	17.22%



- Minimum probability of error (min_PE) for the theoretically optimal threshold value was calculated and the operating point of this error was superimposed on the ROC curve above.



- The below plot shows the decision boundary for each distribution along with equi-level contours of the discriminant function.



Prediction overlapped with decision boundary
class 0 correctly classified
class 1 correctly classified
class 0 wrongly classified
class 1 wrongly classified
X2
X1
10
8
6
4
2
0
−2
−4
−6
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0

Part B: Classifier with Estimated Parameters

- For this part, classification was performed with estimated knowledge of the underlying distributions of the data. Class 0 was modeled as a Gaussian Mixture Model with 2 components and Class 1 was modeled as a single Gaussian. Parameters were estimated using each of the 3 training datasets containing 100, 1000, 10000 samples and then these parameter estimates were used to classify a dataset containing 20,000 samples.
- The likelihood function for a Gaussian Model is defined as

$$\begin{aligned}\mathcal{L} = p(\mathbf{X} \mid \theta) &= \mathcal{N}(\mathbf{X} \mid \theta) \\ &= \mathcal{N}(\mathbf{X} \mid \mu, \Sigma)\end{aligned}$$

where X is the dataset, μ, Σ are the mean and covariance of the distribution.

- To get a good MLE of model, we need good estimates of mean and covariance. They can be calculated as follows,

$$\mu_{MLE} = \text{argmax}_{\mu}\mathcal{N}(\mathbf{X} \mid \mu, \Sigma)$$

$$\Sigma_{MLE} = \text{argmax}_{\Sigma}\mathcal{N}(\mathbf{X} \mid \mu, \Sigma)$$

- Parameter estimation for Class 0 and Class 1 was performed using the built-in function sklearn.mixture.GaussianMixture in python. The iterative numerical optimization method used here is **Expectation-Maximization**(EM) algorithm. This algorithm maximizes the expected value of the log likelihood function of θ as shown below.

$$Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}) = \mathrm{E}_{\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{(t)}}\left[\log L(\boldsymbol{\theta}; \mathbf{X}, \mathbf{Z})\right]$$

$$\boldsymbol{\theta}^{(t+1)} = \underset{\boldsymbol{\theta}}{\arg\max}\, Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)})$$

- Below shows the estimated means, covariance and weights obtained from training on **100** data samples.

$$m_{01} = \begin{bmatrix} 5.09 \\ 0.12 \end{bmatrix} \quad C_{01} = \begin{bmatrix} 3.17 & 0.39 \\ 0.39 & 1.27 \end{bmatrix} \quad m_{02} = \begin{bmatrix} -0.13 \\ 4.13 \end{bmatrix} \quad C_{02} = \begin{bmatrix} 0.87 & 0.06 \\ 0.06 & 2.33 \end{bmatrix}$$

$$m_1 = \begin{bmatrix} 3.15 \\ 1.85 \end{bmatrix} \quad C_1 = \begin{bmatrix} 1.98 & 0.04 \\ 0.04 & 0.99 \end{bmatrix} \quad w1 = 0.47 \ w2 = 0.52$$

- Below shows the estimated means, covariance and weights obtained from training on **1000** data samples.

$$m_{01} = \begin{bmatrix} 5.1 \\ -0.17 \end{bmatrix} \quad C_{01} = \begin{bmatrix} 3.87 & 0.18 \\ 0.18 & 2.11 \end{bmatrix} \quad m_{02} = \begin{bmatrix} 0.1 \\ 3.79 \end{bmatrix} \quad C_{02} = \begin{bmatrix} 1.1 & -0.08 \\ -0.08 & 3.04 \end{bmatrix}$$

$$m_1 = \begin{bmatrix} 3.01 \\ 2.1 \end{bmatrix} \quad C_1 = \begin{bmatrix} 2.18 & -0.07 \\ -0.07 & 1.9 \end{bmatrix} \quad w1 = 0.50 \ w2 = 0.49$$

- Below shows the estimated means, covariance and weights obtained from training on **10,000** data samples.

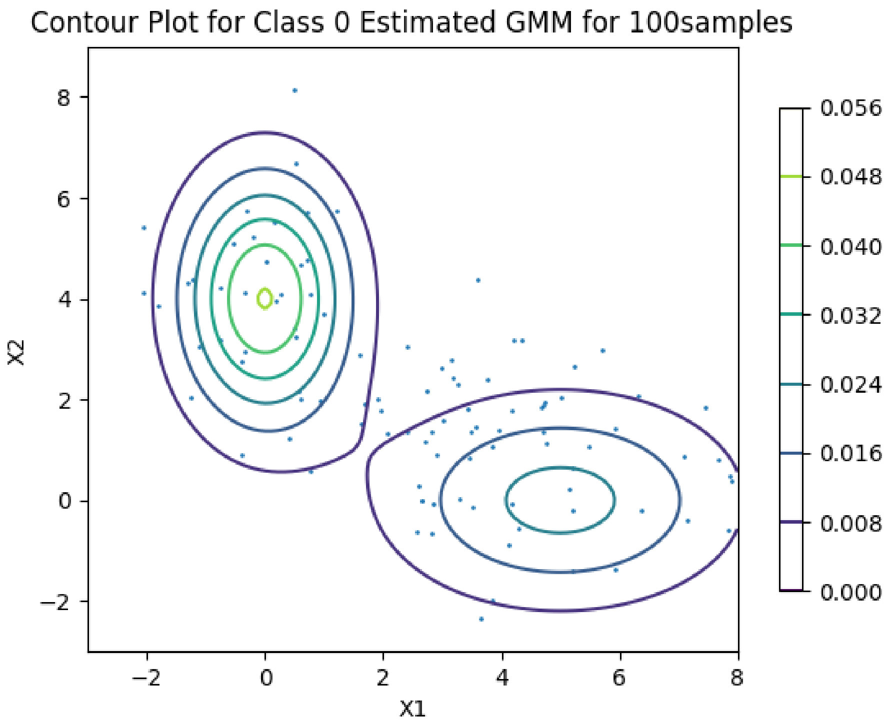
$$m_{01} = \begin{bmatrix} 5.0 \\ -0.01 \end{bmatrix} \quad C_{01} = \begin{bmatrix} 3.88 & 0.01 \\ 0.01 & 1.99 \end{bmatrix} \quad m_{02} = \begin{bmatrix} 0.0 \\ 3.9 \end{bmatrix} \quad C_{02} = \begin{bmatrix} 0.99 & -0.01 \\ -0.01 & 3.07 \end{bmatrix}$$

$$m_1 = \begin{bmatrix} 3.01 \\ 1.98 \end{bmatrix} \quad C_1 = \begin{bmatrix} 2.07 & 0.00 \\ 0.00 & 2.02 \end{bmatrix} \quad w1 = 0.50 \ w2 = 0.49$$

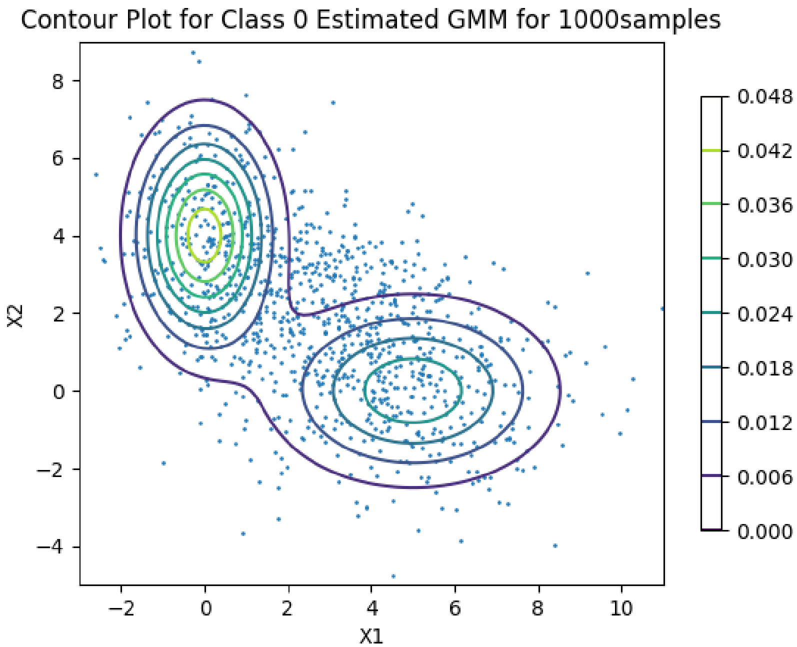
- Below table shows the sample class priors for each of the training data.

	D^{100}_{train}	D^{1000}_{train}	D^{10000}_{train}
P(L=0)	0.66	0.63	0.61
P(L=1)	0.34	0.37	0.39

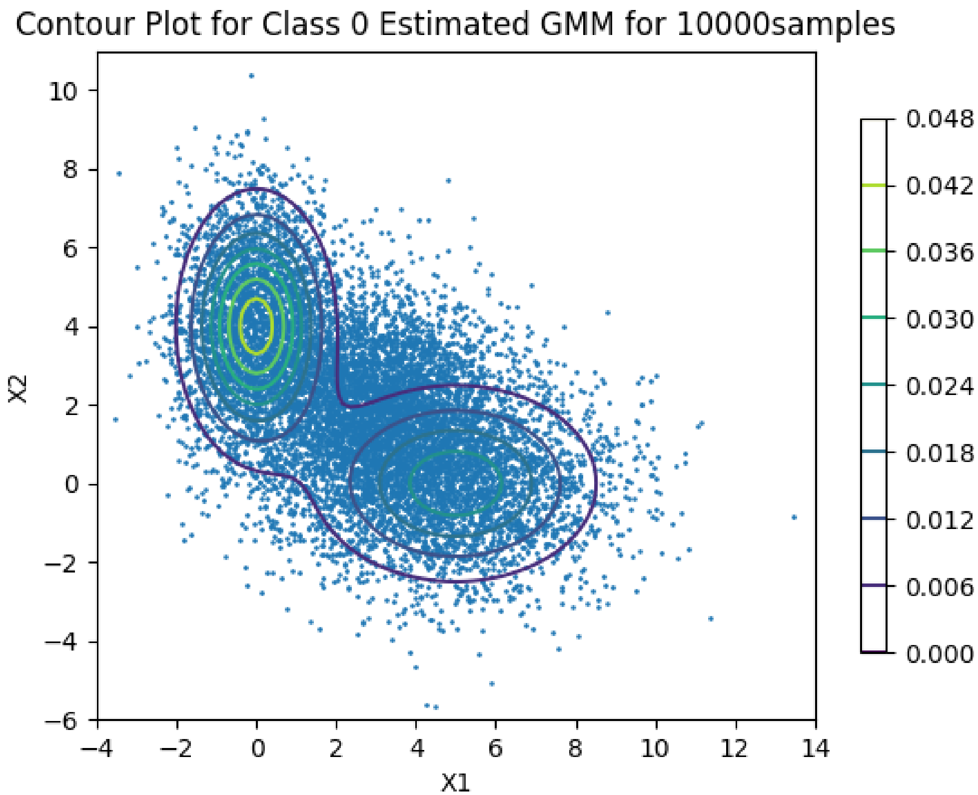
- Below plot shows contour of estimated distributions for Class 0 Gaussian Mixture Model(GMM) for **100** training samples.



- Below plot shows contour of estimated distributions for Class 0 GMM for **1000** training samples.



- Below plot shows contour of estimated distributions for Class 0 GMM for **10,000** training samples.



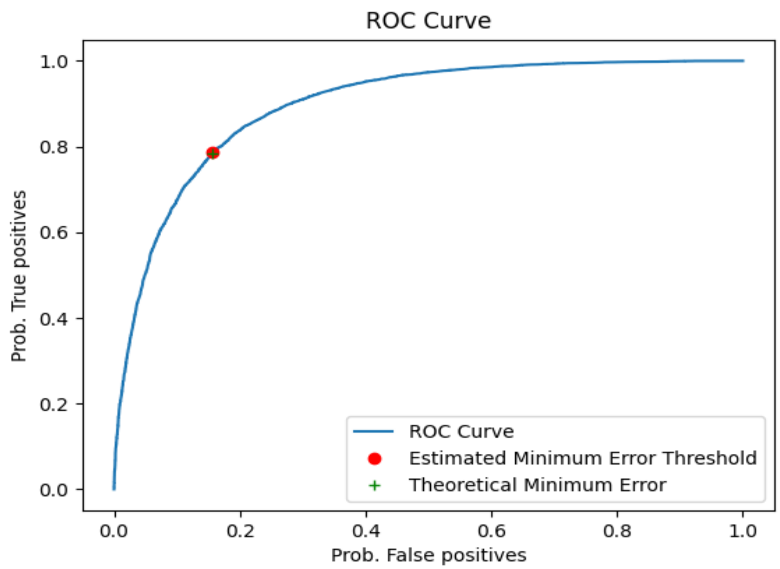
- A summary of the minimum estimated probability of errors associated with the parameters estimated from the three training datasets and validated on **20,000** data samples is shown below.

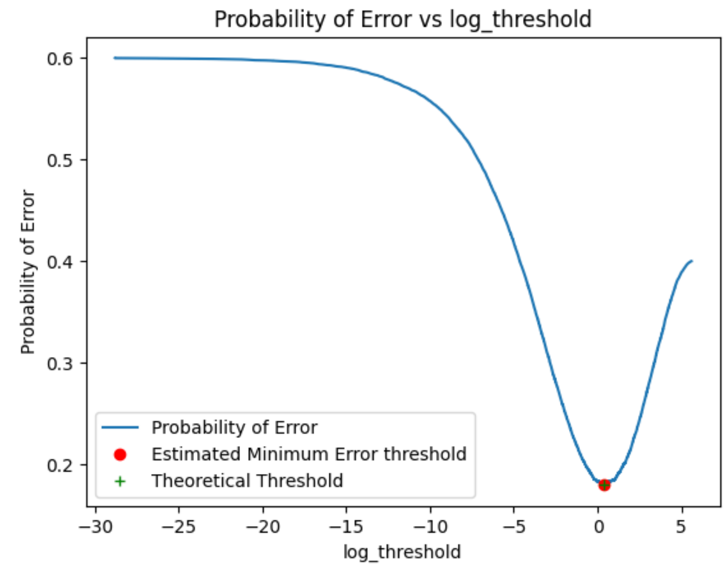
Training samples	Min Error threshold value (γ)	Probability of Error
100	1.46	17.9%
1000	1.26	17.6%
10000	1.61	17.5%
Known PDF (Part 1)	1.68	17.2%

Observation:

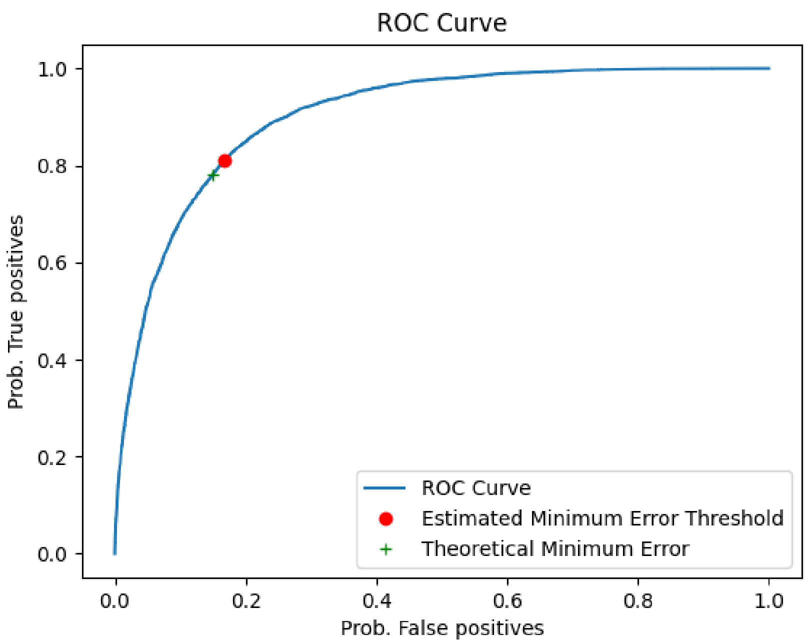
- With the increase in the number of training samples, the parameter estimates (mean, covariance and weights) are closer to the actual value.
- With the increase in the number of training samples, the probability of error on the validation dataset reduces.
- The probability of error for a model trained on 10,000 samples is close to the error calculated using the known data(part 1).

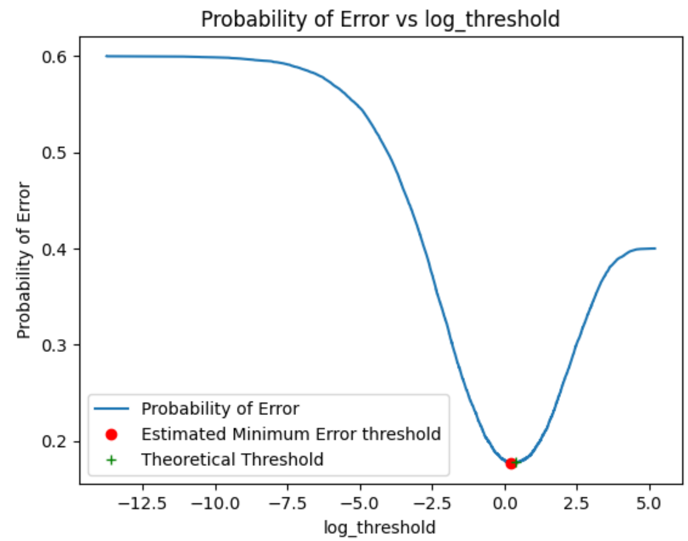
- ROC Curve and Min. Probability of Error plot for training data: **D100** validate data: **D20000**



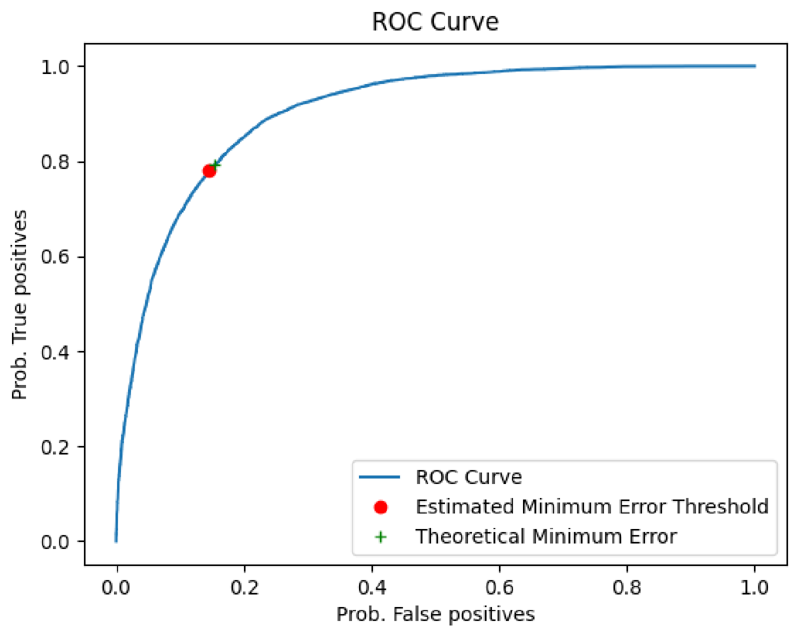


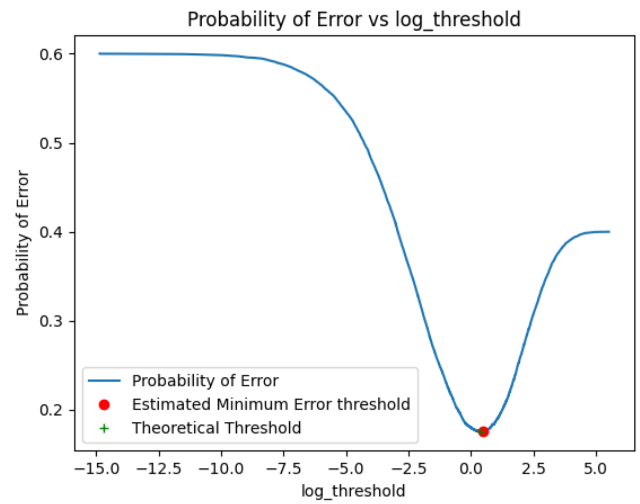
- ROC Curve and Min. Probability of Error plot for training data: **D1000** validate data: **D20000**





- ROC Curve and Min. Probability of Error plot for training data: **D10000** validate data: **D20000**





Part C: Classifier using Logistic Function

- In this part, maximum likelihood parameter estimation techniques were used to train logistic linear and logistic quadratic based approximation of class label posterior functions on a given dataset.
- Training was done on 3 separate training datasets containing 100, 1000, 10000 samples and was used to validate on a dataset containing 20,000 samples.
- The logistic function is shown below:

$$h(x, w) = \frac{1}{1 + e^{w^T z(x)}}$$

- For linear logistic function $z(x) = [1\ x_1\ x_2]^T$
- For quadratic logistic function $z(x) = [1\ x_1\ x_2\ x_1^2\ x_1x_2\ x_2^2]^T$
- The built-in function *scipy.optimize.minimize* was used for numerical optimization of both the functions.

- The learnable vector(w) is estimated using numerical optimization techniques with the cost function as shown below

$$\hat{\theta}_{ML} = -\frac{1}{N}\sum_{1}^{N} l_n \ln\big(h(x_n,\theta)\big) + (1-l_n)\ln\big(1-h(x_n,\theta)\big)$$

- The minimum expected risk classification criteria is

$$(l_n = 1) \qquad \hat{w}^T z(x) \gtrless 0 \qquad (l_n = 0)$$

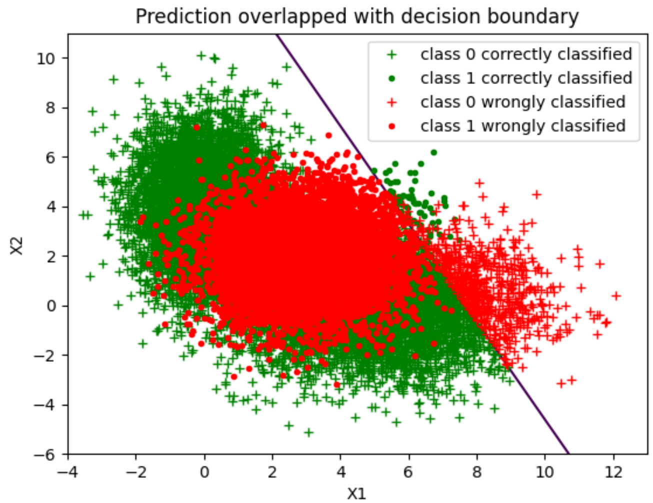
- Below table contains the summary of probability of error(PE) for classifiers trained on the three datasets and validated on a dataset containing 20,000 samples for both linear logistic fit and quadratic logistic fit.
- As the number of training samples increases, the probability of error reduces. As the classifiers are limited by the approximation capability of their functional form, the reduction in the probability of error is less.
- Due to the increase in the complexity in quadratic logistic function, the probability of error is significantly less compared to the linear logistic fit. This function trained on 10000 samples even approached the theoretical optimal probability of error of 17.2% obtained in Part 1.

	PE D^{100}_{train}	PE D^{1000}_{train}	PE D^{10000}_{train}
Linear Logistic Fit	42.7%	42.6%	41.2%
Quadratic Logistic Fit	20.0%	17.4%	17.2%

Data and Classifier Decision on True Label for Linear Logistic Fit

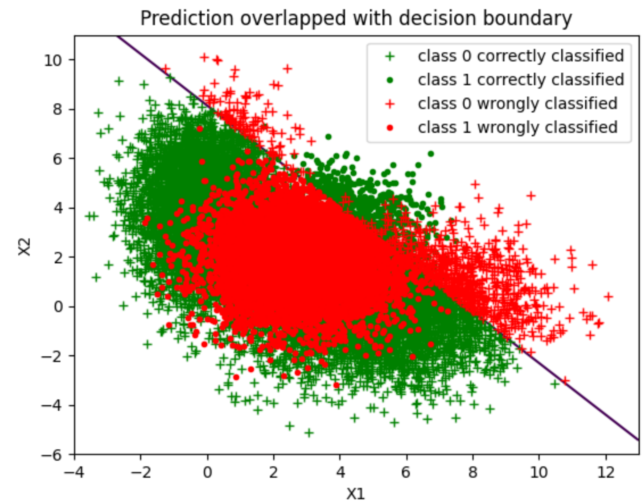
Train Data: **100** Validate Data: **20000**

Probability of Error = **42.7%**



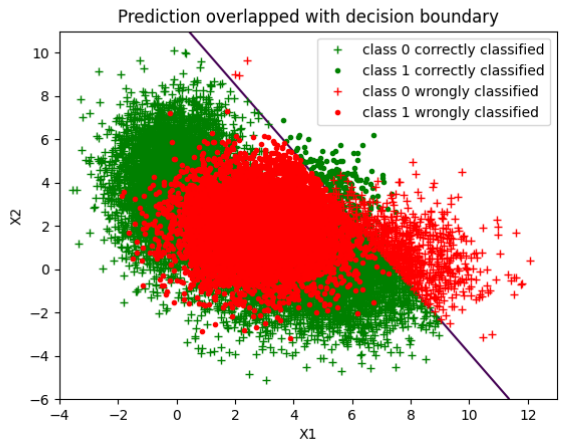
Train Data: **1000** Validate Data: **20000**

Probability of Error = **42.6%**



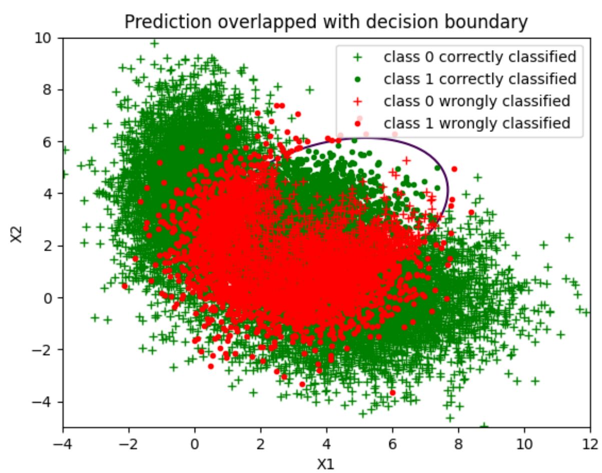
Train Data: **10000** Validate Data: **20000**

Probability of Error = **41.2%**



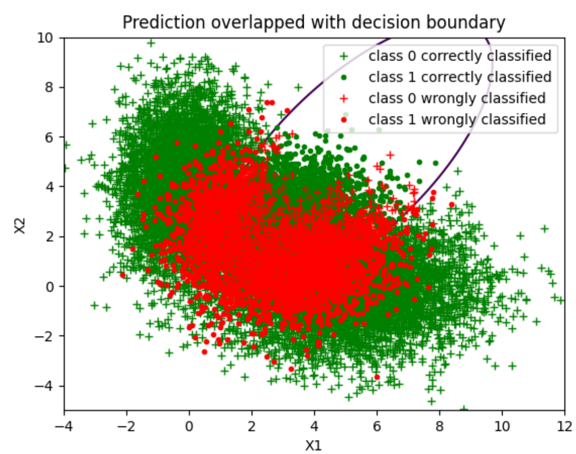
Data and Classifier Decision on True Label for Quadratic Logistic Fit

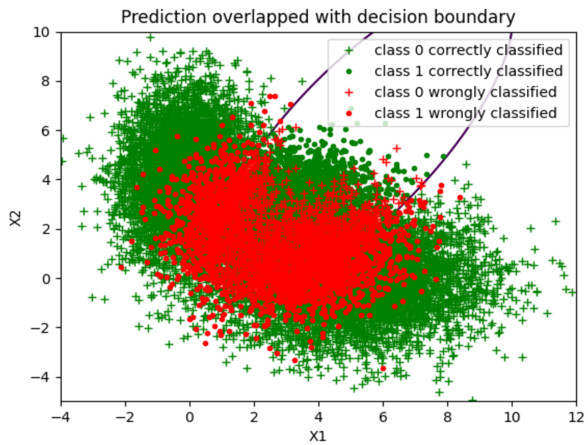
Train Data: **100** Validate Data: **20000**
Probability of Error = **20.0%**



Train Data: **10000** Validate Data: **20000**
Probability of Error = **17.2%**

Train Data: **1000** Validate Data: **20000**
Probability of Error = **17.4%**





Question 2

The objective is to find the [x,y] T coordinate position with the highest probability given the prior distribution as well as the range measurements from each of the K reference coordinates.

$$\begin{pmatrix}\hat{x}\\ y\end{pmatrix}_{MAP} = \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; p\left(\begin{bmatrix}x\\y\end{bmatrix} \middle| \{r_1 \ldots r_k\}\right)$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; \ln p\left(\begin{bmatrix}x\\y\end{bmatrix} \middle| \{r_1 \ldots r_k\}\right)$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; \ln p\left(\{r_1 \ldots r_k\} \middle| \begin{bmatrix}x\\y\end{bmatrix}\right) + \ln p\left(\begin{bmatrix}x\\y\end{bmatrix}\right)$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; \sum_{i=1}^{K} \ln p\left(r_i \middle| \begin{bmatrix}x\\y\end{bmatrix}\right) + \ln p\left(\begin{bmatrix}x\\y\end{bmatrix}\right)$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; \sum_{i=1}^{K} \ln N(n_i|0,\sigma_i^2) + \ln\left((2\pi\sigma_x\sigma_y)^{-1}\, e^{-\frac{1}{2}[x\ y]\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}}\right)$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}}\; \sum_{i=1}^{K} \ln N(n_i|0,\sigma_i^2) + \underbrace{\cancel{\ln\left((2\pi\sigma_x\sigma_y)^{-1}\right)}}_{\to 0\ \text{(constant)}} + \ln\left(e^{-\frac{1}{2}[x\ y]\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}}\right).$$

$$\ldots\ \frac{-((r_i-d_i)-0)^2}{\ldots} \ldots -\frac{1}{2}[x\ y]\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$\begin{bmatrix}x\\y\end{bmatrix} \quad i=1 \qquad \infty$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}} \sum_{i=1}^{k} \ln\left(\left(2\pi\sigma_i^2\right)^{-1/2} e^{-\frac{((r_i-d_i)-0)^2}{2\sigma_i^2}}\right) - \frac{1}{2}\begin{bmatrix}x & y\end{bmatrix}\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}} \sum_{i=1}^{k} \cancel{\ln\left(\left(2\pi\sigma_i^2\right)^{-1/2}\right)} + \ln\left(e^{-\frac{(r_i-d_i)^2}{2\sigma_i^2}}\right) - \frac{1}{2}\begin{bmatrix}x & y\end{bmatrix}\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

0

(constant w.r.t x, y)

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmax}} \sum_{i=1}^{k} -\frac{(r_i-d_i)^2}{2\sigma_i^2} - \frac{1}{2}\begin{bmatrix}x & y\end{bmatrix}\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}.$$

multiply (-2).

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmin}} \begin{bmatrix}x & y\end{bmatrix}\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \sum_{i=1}^{k} \frac{(r_i-d_i)^2}{2\sigma_i^2}.$$

$$= \underset{\begin{bmatrix}x\\y\end{bmatrix}}{\operatorname{argmin}} \begin{bmatrix}x & y\end{bmatrix}\begin{bmatrix}\sigma_x^2 & 0\\ 0 & \sigma_y^2\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \sum_{i=1}^{k} \frac{\left(r_i - \left\|\begin{bmatrix}x\\y\end{bmatrix} - \begin{bmatrix}x_i\\y_i\end{bmatrix}\right\|\right)^2}{\sigma_i^2}.$$

Code implementation:

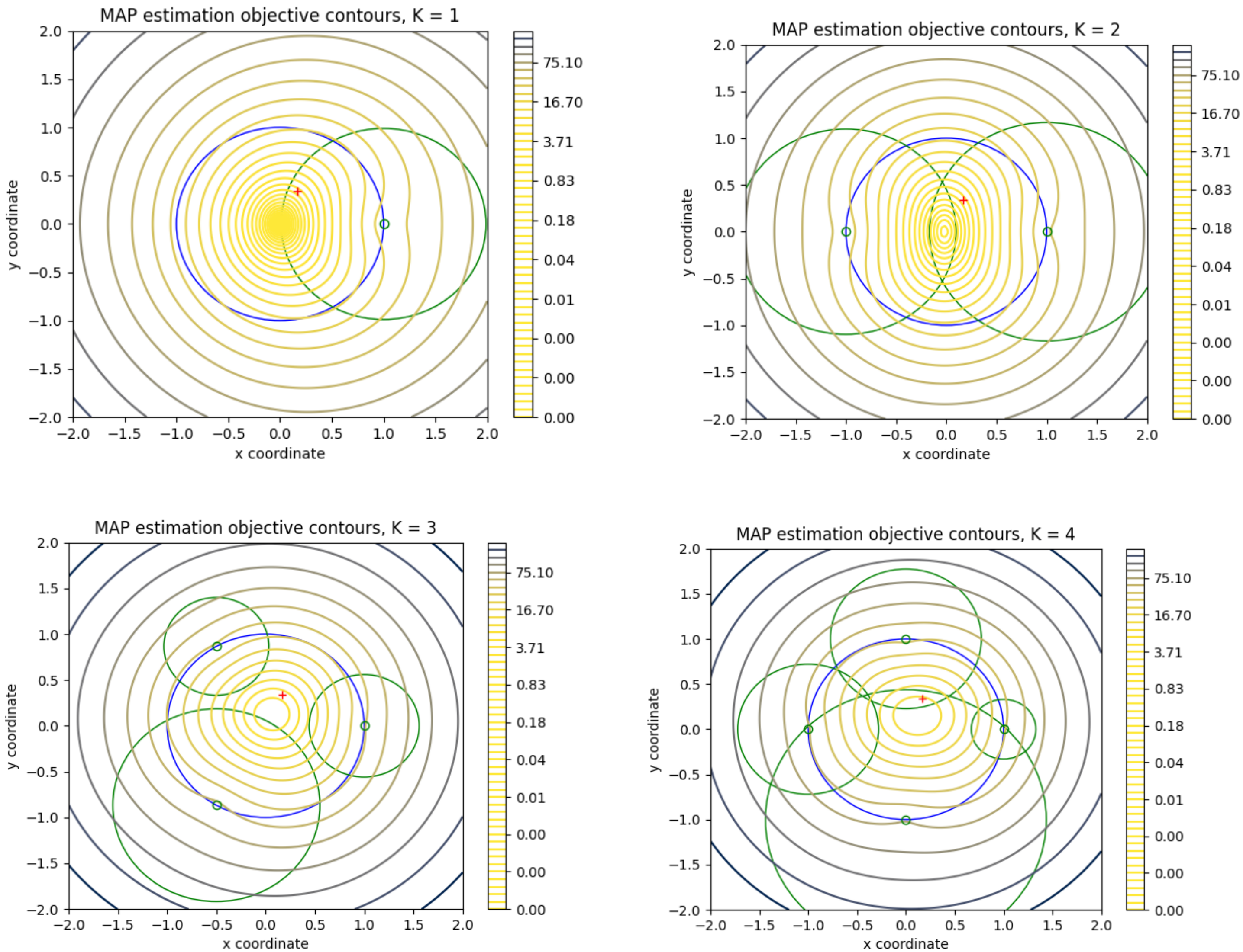
The code was implemented as per the instructions provided in the question.
$\sigma x = \sigma y = 0.25$ and $\sigma i = 0.3$.

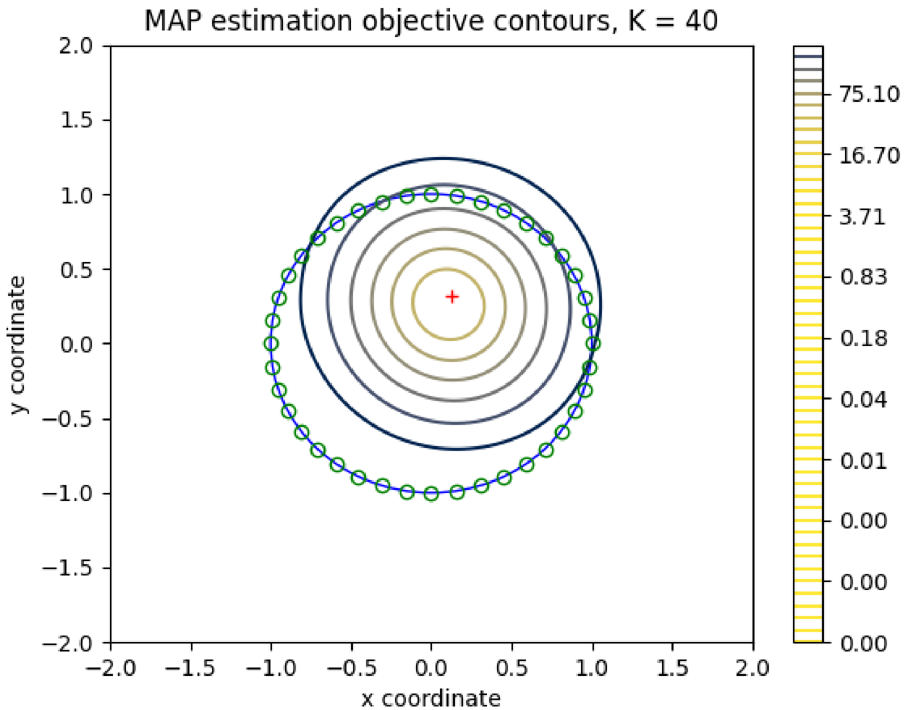
Code Overflow:

- The true position of the vehicle is randomly chosen such that it lies inside a circle of radius=1 whose center is at origin.
- K landmark points are chosen such that they lie on this circle and are equidistant to each other. Euclidean distance was calculated from these landmark points to the true position of the vehicle. Then additive noise was added to these measurements.
- A meshgrid whose x coordinates and y coordinates lie in the range of -2 and 2 was generated. This mesh was divided into 128*128 bins and the MAP objective function(above derived) was used to create the equi level contours.

Observation:

- MAP equi level contour plots for K values between 1 and 4 are shown below.
- Below the unit circle (shown in blue), the true location (shown as a red '+'), the landmark locations (shown as small green circles), and the ranges reported by each landmark (shown as large green circles around their respective landmarks) are marked. As we have to minimize the MAP objective function, we focus on the contour with minimum MAP function value(yellow contours - center of the innermost contour).
- The MAP estimate of position for K = 1 and K = 2 are not accurate and the minimum MAP estimate is symmetric about the origin. However, for K = 3 and K = 4, the estimator is much more accurate. This can be seen on the contour graph, in which the true location lies within only two and one contour levels away from the central estimate contour, respectively.
- Generally, the MAP estimate overlaps more with the true position as K increases. While it is not true in the transition from K = 1 → 2, it becomes a stronger trend as K becomes very large.
- Based on the distance from the true location to the point with the lowest contour (center of the innermost contour), the estimator's accuracy can be determined. The certainty of the estimator increases as we increase the number of landmark points.
- The certainty of the estimator can be visualized on the contour graphs by a shrinkage of the area of locations with a high probability. This behaviour was clearly seen for K=40(below figure) where the accuracy and certainty of the MAP estimate is higher.


MAP estimation objective contours, K = 1
MAP estimation objective contours, K = 2
MAP estimation objective contours, K = 3
MAP estimation objective contours, K = 4
x coordinate
y coordinate
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
75.10
16.70
3.71
0.83
0.18
0.04
0.01
0.00
0.00
0.00



Question 3:

The proof and solution for this question is attached below for your reference.

$$R = \Lambda \, P(L|X)$$

where R is the risk, Λ is the loss matrix

and $P(L/X)$ is the posterior probability.

For c classes,

$$\begin{bmatrix} R(D=1|x) \\ \vdots \\ R(D=c|X) \end{bmatrix} = \Lambda \begin{bmatrix} P(L=1|x) \\ \vdots \\ P(L=c|X) \end{bmatrix}$$

As λ_s - substitution error,

$$\Lambda = \begin{bmatrix} 0 & \lambda_s & \cdots & \lambda_s \\ \lambda_s & 0 & \cdots & \lambda_s \\ \lambda_s & \cdots & \lambda_s & 0 \end{bmatrix}$$

where non-diagonal elements are λ_s.

Equating element wise,

$$R(D = w_i | x) = \sum_{\substack{j=1 \\ j \neq i}} \lambda_s P(L = w_j | x)$$

$$= \lambda_s \sum_{\substack{j=1 \\ j \neq i}} P(L = w_j | x).$$

$$= \lambda_s (1 - P(L = w_i | x)).$$

We choose w_i when

$$\lambda_s (1 - P(L = w_i | x)) < \lambda_s (1 - P(L = w_j | x))$$

$$\boxed{P(L = w_i | x) > P(L = w_j | x) \quad \text{for all } j \in [1, c] \; j \neq i}$$

To choose w_i instead of rejection,

$$\lambda_s (1 - P(L = w_i | x)) < \lambda_r$$

$$(1 - P(L = w_i | x)) < \frac{\lambda_r}{\lambda_s}$$

$$\boxed{P(L = w_i | x) > 1 - \frac{\lambda_r}{\lambda_s}}$$

when $\lambda_r = 0$, the rejection condition becomes

$$P(L = w_i | x) > 1$$

Since $0 \leq P(L = w_i | x) \leq 1$, w_i won't be choosen over rejection. So, all samples would be rejected.

when $\lambda_r > \lambda_s$, the condition becomes

$$P(L = w_i | x) > 1 - \frac{\lambda_r}{\lambda_s}.$$

when $\frac{\lambda_r}{\lambda_s} > 1$ $\quad 1 - \frac{\lambda_r}{\lambda_s} < 0.$

Since $0 \leq P(L = w_i | x) \leq 1$, substitution will always be chosen over rejection. So, no samples woulbe rejected.

Code:
Question 1:

```
import numpy as np
import scipy.stats
import random
import matplotlib.pyplot as plt
import sys
from sklearn.mixture import GaussianMixture
from scipy.optimize import minimize
from matplotlib.colors import LogNorm
np.set_printoptions(suppress=True)

def calc_pxl(data, mean, cov):

    return scipy.stats.multivariate_normal.pdf(data, mean=mean, cov=cov)

def calc_prob_threshs(sample_type, log_score, log_thresh_range):

    tps, tns, fps, fns, fs = [], [], [], [], []

    num_samples = sample_type[0]
    N0, N1 = sample_type[1]
    data_wt_labels = sample_type[2]
    labels = data_wt_labels[2,:]

    for log_thresh in log_thresh_range:

        tp, tn, fp, fn, f = calc_prob_thresh(log_score, log_thresh, labels, N0, N1)
        tps.append(tp); fps.append(fp)
        tns.append(tn); fns.append(fn)
        fs.append(f)

    tps = np.array(tps); tns = np.array(tns)
    fps = np.array(fps); fns = np.array(fns)
```

```
    fs = np.array(fs)

    sample_type[3] = [tps, tns, fps, fns, fs]

    return sample_type

def calc_prob_thresh(log_score, log_thresh, labels, N0, N1):

    decisions = (log_score>log_thresh).astype('int')
    #print('decisions ',decisions)

    tp = np.sum(np.multiply(labels == 1, decisions==1).astype('int'))/N1
    fp = np.sum(np.multiply(labels == 0, decisions==1).astype('int'))/N0
    tn = np.sum(np.multiply(labels == 0, decisions==0).astype('int'))/N0
    fn = np.sum(np.multiply(labels == 1, decisions==0).astype('int'))/N1
    f = (fp*N0 + fn*N1)/(N0 + N1)

    return tp, tn, fp, fn, f

def erm(sample_type, means, covs):

    #data_wt_labels (3, N)
    print('***** erm *****')
    m0, m1 = means
    C0, C1 = covs

    data_wt_labels = sample_type[2]
    pts = data_wt_labels[:2,:].T ##(N, 2)
    labels = data_wt_labels[2,:]

    px0_0 = scipy.stats.multivariate_normal.pdf(pts, mean=m0[0,:], cov=C0[0,:,:])
    px0_1 = scipy.stats.multivariate_normal.pdf(pts, mean=m0[1,:], cov=C0[1,:,:])

    px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
    px1 = scipy.stats.multivariate_normal.pdf(pts, mean=m1, cov=C1) ##(N, 1)
```

```
score = np.divide(px1, px0)
log_score = np.log(score)
sort_log_score = np.sort(log_score)  ##(N, 1)

eps = 1e-3
log_thresh_range = np.append(sort_log_score[0] - eps, sort_log_score + eps)
sample_type = calc_prob_threshs(sample_type, log_score, log_thresh_range)

# theoretical
log_thresh_t = np.log(pL[0]/pL[1])
N0, N1 = sample_type[1]
tp_t, tn_t, fp_t, fn_t, f_t = calc_prob_thresh(log_score, log_thresh_t, labels, N0, N1)

# min PE thresh from data
tps, tns, fps, fns, fs = sample_type[3]
min_poe = np.min(fs)
min_poe_ids = np.where(fs==min_poe)[0]

# get closest thresh to theoretical
min_dist, min_id = sys.maxsize, 0
for id in min_poe_ids:
    dist = log_thresh_range[id] - log_thresh_t
    if dist<min_dist:
        min_dist = dist
        min_id = id

print('min_poe_t ',f_t)
print('min_poe_a ',fs[min_id])
print('min_poe_thresh ',np.exp(log_thresh_range[min_id]))
print('thresh_t ',np.exp(log_thresh_t))

#ROC curve
plt.plot(fps, tps, label='ROC Curve')
plt.plot(fps[min_id], tps[min_id], 'ro', label='Estimated Minimum Error')
plt.plot(fp_t, tp_t, 'g+', label='Theoretical Minimum Error')
plt.title('Minimum Expected Risk ROC Curve')
```

```
plt.xlabel('Prob. False positives')
plt.ylabel('Prob. True positives')
plt.legend()
plt.show()

# Probability of Error
plt.plot(log_thresh_range, fs, label='Probability of Error')
plt.plot(log_thresh_range[min_id], fs[min_id], 'ro', label='Estimated Minimum Error threshold')
plt.plot(log_thresh_t, f_t, 'g+', label='Theoretical Threshold')
plt.title('Probability of Error vs log_threshold')
plt.xlabel('log_threshold')
plt.ylabel('Probability of Error')
plt.legend()
plt.show()

# Decision boundary
log_score = np.log(score)
decisions = (log_score>log_thresh_t).astype('int')
pts = pts.T
plot_boundary(pts, labels, decisions)
hgrid = np.linspace(np.floor(min(pts[0,:])),np.ceil(max(pts[0,:])),100)
vgrid = np.linspace(np.floor(min(pts[1,:])),np.ceil(max(pts[1,:])),100)
dsg = np.zeros((100,100))
mat = np.array(np.meshgrid(hgrid, vgrid))

for i in range(100):
    for j in range(100):
        px0_0 = scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j], mat[1][i][j]]), mean=m0[0,:], cov=C0[0,:,:])
        px0_1 = scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j], mat[1][i][j]]), mean=m0[1,:], cov=C0[1,:,:])
        px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
        px1 = scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j], mat[1][i][j]]), mean=m1, cov=C1) ##(N, 1)
        dsg[i][j] = np.log(px0) - np.log(px1) - np.log(pL[0]/pL[1])

plt.contour(mat[0], mat[1], dsg)
plt.show()
```

```
def split_data(data_wt_labels):

    l0_ids = np.where(data_wt_labels[2,:]==0)[0]
    l1_ids = np.where(data_wt_labels[2,:]==1)[0]

    data0 = data_wt_labels[:,l0_ids]
    data1 = data_wt_labels[:,l1_ids]

    return data0, data1

def print_gmm_params(gmm_l0, gmm_l1):

    print('GMM params L0 ',gmm_l0.get_params())
    print('GMM params L1 ',gmm_l1.get_params())

    if gmm_l0.converged_:print('Label 0 converged')
    else:print('Label 0 not converged')

    if gmm_l1.converged_:print('Label 1 converged')
    else:print('Label 1 not converged')

    print('Label 0 weights ',gmm_l0.weights_, gmm_l0.weights_.shape)
    print('Label 1 weights ',gmm_l1.weights_, gmm_l1.weights_.shape)

    print('Label 0 means ',gmm_l0.means_.shape)
    print('Label 0 covariances ',gmm_l0.covariances_.shape)

    print('Label 1 means ',gmm_l1.means_.shape)
    print('Label 1 covariances ',gmm_l1.covariances_.shape)


def mle_gmm(train_sample_type, val_sample_type):

    data_wt_labels = train_sample_type[2]
    data0, data1 = split_data(data_wt_labels)
    data0, data1 = data0[:2, :].T, data1[:2, :].T
```

```
gmm_l0 = GaussianMixture(2, covariance_type='full',
            random_state=0).fit(data0)

gmm_l1 = GaussianMixture(1, covariance_type='full',
            random_state=0).fit(data1)

#print_gmm_params(gmm_l0, gmm_l1)

m01 = gmm_l0.means_[0,:]
m02 = gmm_l0.means_[1,:]
C01 = gmm_l0.covariances_[0,:]
C02 = gmm_l0.covariances_[1,:]
gmm_weights0 = gmm_l0.weights_

w1 = gmm_weights0[0]; w2 = gmm_weights0[1]

m1 = gmm_l1.means_[0,:]
C1 = gmm_l1.covariances_[0,:]
gmm_weights1 = gmm_l1.weights_

print('C01: ', C01)
print('C02: ', C02)
print('C1: ', C1)

print('m01: ', m01)
print('m02: ', m02)
print('m1: ', m1)

print('w1 ',w1)
print('w2 ',w2)

data_wt_labels = val_sample_type[2]
pts = data_wt_labels[:2,:].T ##(N, 2)
labels = data_wt_labels[2,:]
```

```
px0_0 = scipy.stats.multivariate_normal.pdf(pts, mean=m01, cov=C01)
px0_1 = scipy.stats.multivariate_normal.pdf(pts, mean=m02, cov=C02)

px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
px1 = scipy.stats.multivariate_normal.pdf(pts, mean=m1, cov=C1) ##(N, 1)

score = np.divide(px1, px0)
log_score = np.log(score)
sort_log_score = np.sort(log_score)  ##(N, 1)

eps = 1e-3
log_thresh_range = np.append(sort_log_score[0] - eps, sort_log_score + eps)
val_sample_type = calc_prob_threshs(val_sample_type, log_score, log_thresh_range)

# theoretical
log_thresh_t = np.log(pL[0]/pL[1])
N0, N1 = val_sample_type[1]
tp_t, tn_t, fp_t, fn_t, f_t = calc_prob_thresh(log_score, log_thresh_t, labels, N0, N1)

# min PE thresh from data
tps, tns, fps, fns, fs = val_sample_type[3]
min_poe = np.min(fs)
min_poe_ids = np.where(fs==min_poe)[0]

# get closest thresh to theoretical
min_dist, min_id = sys.maxsize, 0
for id in min_poe_ids:
    dist = log_thresh_range[id] - log_thresh_t
    if dist<min_dist:
        min_dist = dist
        min_id = id

print('min_poe_t ',f_t)
print('min_poe_a ',fs[min_id])
print('min_poe_thresh ',np.exp(log_thresh_range[min_id]))
print('thresh_t ',np.exp(log_thresh_t))
```

```
#ROC curve
plt.plot(fps, tps, label='ROC Curve')
plt.plot(fps[min_id], tps[min_id], 'ro', label='Estimated Minimum Error Threshold')
plt.plot(fp_t, tp_t, 'g+', label='Theoretical Minimum Error')
plt.title('ROC Curve')
plt.xlabel('Prob. False positives')
plt.ylabel('Prob. True positives')
plt.legend()
plt.show()

# Probability of Error
plt.plot(log_thresh_range, fs, label='Probability of Error')
plt.plot(log_thresh_range[min_id], fs[min_id], 'ro', label='Estimated Minimum Error threshold')
plt.plot(log_thresh_t, f_t, 'g+', label='Theoretical Threshold')
plt.title('Probability of Error vs log_threshold')
plt.xlabel('log_threshold')
plt.ylabel('Probability of Error')
plt.legend()
plt.show()

# GMM contour for Class 0
data_wt_labels = train_sample_type[2]
pts = data_wt_labels[:2,:] ##(2, N)
hgrid = np.linspace(np.floor(min(pts[0,:])),np.ceil(max(pts[0,:])),100)
vgrid = np.linspace(np.floor(min(pts[1,:])),np.ceil(max(pts[1,:])),100)
dsg = np.zeros((100,100))
mat = np.array(np.meshgrid(hgrid, vgrid))

for i in range(100):
    for j in range(100):
        px0_0 = scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j], mat[1][i][j]]), mean=m0[0,:], cov=C0[0,:,:])
        px0_1 = scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j], mat[1][i][j]]), mean=m0[1,:], cov=C0[1,:,:])
        dsg[i][j] = w1*px0_0 + w2*px0_1 ##(N, 1)

CS = plt.contour(mat[0], mat[1], dsg)
```

```
    CB = plt.colorbar(CS, shrink=0.8, extend='both')
    plt.scatter(pts[0,:], pts[1,:], .8)
    plt.title('Contour Plot for Class 0 Estimated GMM for ' + str(pts.shape[1]) + 'samples')
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.show()

def calc_cost(x, data, labels):

    w = x

    h = 1 / (1+ np.exp(-(np.dot(w.T,data)))) ##(N, )
    loss = labels * np.log(h) + (1 - labels) * np.log(1 - h)  ##(N, )

    sum = np.sum(loss)
    scale = -(1.0 / data.shape[1])
    cost = scale * sum

    return cost

def predict(w, data, thresh=0.5):

    h = 1 / (1+ np.exp(-(np.dot(w.T,data))))

    h[h>=thresh] = 1
    h[h<thresh] = 0

    return h

def mle_opt_lin(train_sample_type, test_sample_type):

    train_data_wt_labels = train_sample_type[2]
    train_data = train_data_wt_labels[:2, :]
    train_labels = train_data_wt_labels[2,:]
    train_ones = np.ones(train_data_wt_labels.shape[1]).reshape((1, -1))
    train_data = np.concatenate((train_ones, train_data), axis=0)  ##(3, N)
```

```
    print('training.. ')
    w_init = np.zeros((3, 1), dtype='float')
    result = minimize(calc_cost, w_init, args=(train_data, train_labels))
    w_trained = result.x
    print('training completed!')

    print('w_trained ',w_trained)
    test_data_wt_labels = test_sample_type[2]
    test_data = test_data_wt_labels[:2, :]
    test_labels = test_data_wt_labels[2,:]
    test_ones = np.ones(test_data_wt_labels.shape[1]).reshape((1, -1))
    test_data = np.concatenate((test_ones, test_data), axis=0)  ##(3, N)

    decisions = predict(w_trained, test_data)
    acc = calc_poe(decisions, test_labels)
    print('acc ',acc)

    plot_boundary(test_data_wt_labels[:2, :], test_labels, decisions)
    mat = get_mesh_grid(test_data_wt_labels[:2, :])
    boundary = np.zeros((100, 100))
    for i in range(100):
        for j in range(100):
            x1 = mat[0][i][j]
            x2 = mat[1][i][j]
            z = np.c_[1, x1, x2].T
            boundary[i][j] = np.sum(np.dot(w_trained.T, z))
    plt.contour(mat[0], mat[1], boundary, levels = [0])
    plt.show()

def gen_quad_data(data_wt_labels):

    num_samples = data_wt_labels.shape[1]
    input_data = data_wt_labels[:2, :] # (x1, x2)
    data = np.zeros((6, num_samples), dtype='float') # (1, x1, x2, x1**2, x1x2, x2**2)
```

```
    data[0] = np.ones(num_samples).reshape((1, -1)) # 1
    data[1:3] = input_data #x1, x2
    data[3] = np.square(input_data[0, :]) # x1**2
    data[4] = np.multiply(input_data[0, :], input_data[1, :]) # x1x2
    data[5] = np.square(input_data[1, :]) # x2**2

    return data

def mle_opt_quad(train_sample_type, test_sample_type):

    train_data_wt_labels = train_sample_type[2]
    train_labels = train_data_wt_labels[2,:]
    train_data = gen_quad_data(train_data_wt_labels)

    print('training.. ')
    w_init = np.zeros((6, 1), dtype='float')
    result = minimize(calc_cost, w_init, args=(train_data, train_labels))
    w_trained = result.x
    print('training completed!')

    print('w_trained ',w_trained)
    test_data_wt_labels = test_sample_type[2]
    test_data = gen_quad_data(test_data_wt_labels)
    test_labels = test_data_wt_labels[2,:]

    decisions = predict(w_trained, test_data)
    acc = calc_poe(decisions, test_labels)
    print('acc ',acc)

    plot_boundary(test_data_wt_labels[:2, :], test_labels, decisions)
    mat = get_mesh_grid(test_data_wt_labels[:2, :])
    boundary = np.zeros((100, 100))
    for i in range(100):
        for j in range(100):
            x1 = mat[0][i][j]
            x2 = mat[1][i][j]
```

```
            z = np.c_[1, x1, x2, x1**2, x1*x2, x2**2].T
            boundary[i][j] = np.sum(np.dot(w_trained.T, z))
    plt.contour(mat[0], mat[1], boundary, levels = [0])
    plt.show()

def calc_poe(decisions, labels):

    N0 = np.sum((labels == 0).astype('int'))
    N1 = np.sum((labels == 1).astype('int'))

    tp = np.sum(np.multiply(labels == 1, decisions==1).astype('int'))/N1
    fp = np.sum(np.multiply(labels == 0, decisions==1).astype('int'))/N0
    tn = np.sum(np.multiply(labels == 0, decisions==0).astype('int'))/N0
    fn = np.sum(np.multiply(labels == 1, decisions==0).astype('int'))/N1
    f = (fp*N0 + fn*N1)/(N0 + N1)

    return (tp*N1 + tn*N0)/(N0 + N1)

def mle_opt(train_sample_type, test_sample_type, part=1):

    if part==1:
        mle_opt_lin(train_sample_type, test_sample_type)
    else:
        mle_opt_quad(train_sample_type, test_sample_type)

def plot_boundary(data, labels, decisions):

    tp = np.multiply(labels == 1, decisions == 1).astype('int')
    tn = np.multiply(labels == 0, decisions == 0).astype('int')
    fp = np.multiply(labels == 0, decisions == 1).astype('int')
    fn = np.multiply(labels == 1, decisions == 0).astype('int')

    tp_ids = np.where(tp == 1)[0]
    tn_ids = np.where(tn == 1)[0]
    fp_ids = np.where(fp == 1)[0]
    fn_ids = np.where(fn == 1)[0]
```

```
    plt.plot(data[0, tn_ids], data[1, tn_ids], '+', color ='g', markersize = 6)
    plt.plot(data[0, tp_ids], data[1, tp_ids], '.', color ='g', markersize = 6)
    plt.plot(data[0, fp_ids], data[1, fp_ids], '+', color ='r', markersize = 6)
    plt.plot(data[0, fn_ids], data[1, fn_ids], '.', color ='r', markersize = 6)

    plt.legend(["class 0 correctly classified", 'class 1 correctly classified','class 0 wrongly classified', 'class 1 wrongly classified'])
    plt.title('Prediction overlapped with decision boundary')
    plt.xlabel("X1")
    plt.ylabel("X2")

def get_mesh_grid(data, num_grid=100):

    hgrid = np.linspace(np.floor(min(data[0,:])), np.ceil(max(data[0,:])), num_grid)
    vgrid = np.linspace(np.floor(min(data[1,:])), np.ceil(max(data[1,:])), num_grid)
    mat = np.array(np.meshgrid(hgrid, vgrid))

    return mat

def plot_dist(data, label_names):

    tname, xname, yname = label_names

    print('***** plot *****')
    data0, data1 = split_data(data)

    plt.scatter(data0[0, :], data0[1, :], s=5, color = 'red', label = 'class 0',marker='*')
    plt.scatter(data1[0, :], data1[1, :], s=5, color = 'blue', label = 'class 1', marker='*')

    plt.title(tname)
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.legend()
    plt.show()

def generate_data_pxgl(prior, means, covs, num_samples):
```

```
    m0, m1 = means
    C0, C1 = covs

    N0 = int(prior[0]*num_samples)
    N1 = num_samples - N0
    print('N0, N1 ',N0, N1)

    # generate L0
    N00 = int(w1*N0)
    N01 = N0 - N00
    wt_dist = [0]*N00 + [1]*N01
    for i in range(10):
        random.shuffle(wt_dist)
    wt_dist = np.array(wt_dist)

    dist0 = np.random.multivariate_normal(m0[0,:], C0[0,:,:], N0).T
    dist1 = np.random.multivariate_normal(m0[1,:], C0[1,:,:], N0).T
    pxgl0 = np.multiply(1-wt_dist, dist0) + np.multiply(wt_dist, dist1)

    # generate L1
    pxgl1 = np.random.multivariate_normal(m1, C1, N1).T

    ## combine data and label
    labels = [0]*N0 + [1]*N1
    labels = np.reshape(labels, (1, -1))

    pxgl = np.concatenate((pxgl0, pxgl1), axis=1)
    data = np.concatenate((pxgl, labels), axis=0)

    return data, N0, N1

def generate_data_pxgl_samples(samples_type):

    for i, key in enumerate(samples_type.keys()):
```

```
        sample_type = samples_type[key]
        num_samples = int(sample_type[0][0])

        data_wt_labels, N0, N1 = generate_data_pxgl(pL, [m0, m1], [C0, C1], num_samples)

        sample_type[1] = [N0, N1]
        sample_type[2] = data_wt_labels

        label_names = ["True label distribution for " + str(num_samples) + " for two classes", "x1", "x2"]
        plot_dist(data_wt_labels, label_names)

    return samples_type

if __name__ == "__main__":

    dim = 2

    #priors
    pL = [0.6, 0.4]

    #means
    m0 = np.array([[5, 0], [0, 4]])
    m1 = [3, 2]

    #covariance
    C0 = np.zeros((2,2,2), dtype=int)
    C0[0,:,:] = np.array([[4, 0], [0, 2]])
    C0[1,:,:] = np.array([[1, 0], [0, 3]])
    C1 = np.array([[2, 0], [0, 2]])

    ## gaus weight
    w1 = 0.5; w2 = 0.5

    # data
    ## num_samples, [N0, N1], data_wt_labels, [tps, tns, fps, fns, fs]
    samples_type = {
```

```
    'D100': [[100], [], [], []],
    'D1k': [[1000], [], [], []],
    'D10k': [[10000], [], [], []],
    'D20k': [[20000], [], [], []],
}

## generate data for all samples
samples_type = generate_data_pxgl_samples(samples_type)

erm_ = 0
gmm_ = 0
opt_ = 1

## erm
if erm_:
    print('Part1')
    erm(samples_type['D20k'], [m0, m1], [C0, C1])

## mle_gmm
if gmm_:
    print('Part2')
    for i, key in enumerate(list(samples_type.keys())[:-1]):

        print('***********************************')
        print('train: ',key,' val: D20k')
        mle_gmm(samples_type[key], samples_type['D20k'])
        print('***********************************')

        # if i==0:break

if opt_:
    print('Part3')
    for i, key in enumerate(list(samples_type.keys())[:-1]):

        print('***********************************')
        print('train: ',key,' val: D20k')
```

```
        mle_opt(samples_type[key], samples_type['D20k'], part=1)
        mle_opt(samples_type[key], samples_type['D20k'], part=2)
        print('*********************************')
```

Question 2:

```
import numpy as np
import random
import matplotlib.pyplot as plt

CONTOUR_LEVELS = np.geomspace(0.0001, 250, 50)

def get_true_pos(center, radius):

    rand_r = random.uniform(0.0, 1.0)
    rand_t = random.uniform(0.0, 1.0)

    r = radius * np.sqrt(rand_r)
    theta = 2 * np.pi * rand_t
    x = center[0] + r * np.cos(theta)
    y = center[1] + r * np.sin(theta)

    return np.array([x, y])

def get_equidist_points(center, radius, num_points):

    equ_pts = []
    for i in range(num_points):
        theta = 2.0 * np.pi * i/ num_points
        x = center[0] + radius * np.cos(theta)
        y = center[1] + radius * np.sin(theta)
        equ_pts.append([x, y])

    return np.array(equ_pts)
```

```
def get_measurements(equ_pts, true_pos):

    measurements = []
    for equ_pt in equ_pts:

        dist = np.linalg.norm(true_pos - equ_pt)
        measurement = dist + np.random.normal(0, sigi)

        while measurement<=0:
            measurement = dist + np.random.normal(0, sigi)

        measurements.append(measurement)

    return np.array(measurements)

def get_MAP_contour(equ_pts, measurements, quad_range, num_grid_pts):

    min_, max_ = quad_range[0], quad_range[1]
    xgrid = np.linspace(min_, max_, num_grid_pts)
    ygrid = np.linspace(min_, max_, num_grid_pts)
    mat = np.array(np.meshgrid(xgrid, ygrid))

    contours = np.zeros((num_grid_pts, num_grid_pts), dtype='float')
    for i in range(num_grid_pts):
        for j in range(num_grid_pts):
            x1 = mat[0][i][j]
            x2 = mat[1][i][j]
            pt = np.array([x1, x2])
            contours[i][j] = get_MAP_obj(pt, equ_pts, measurements, num_pts)

    return contours, mat

def plot_equilevel_contours(equ_pts, measurements, quad_range, num_grid_pts):

    contours, grid = get_MAP_contour(equ_pts, measurements, quad_range, num_grid_pts)
```

```
    ax = plt.gca()

    unit_circle = plt.Circle((0, 0), 1, color='blue', fill=False)
    ax.add_artist(unit_circle)

    plt.contour(grid[0], grid[1], contours, cmap='cividis_r', levels=CONTOUR_LEVELS)

    for (pt_i, r_i) in zip(equ_pts, measurements):

        print('r i ',r_i)
        x, y = pt_i[0], pt_i[1]
        plt.plot((x), (y), 'o', color='g', markerfacecolor='none')
        range_circle = plt.Circle((x, y), r_i, color='g', fill=False)
        ax.add_artist(range_circle)

    ax.set_xlabel("x coordinate")
    ax.set_ylabel("y coordinate")
    ax.set_title("MAP estimation objective contours, K = " + str(len(measurements)))

    ax.set_xlim((-2, 2))
    ax.set_ylim((-2, 2))
    ax.plot([true_pos[0]], [true_pos[1]], '+', color='r')
    plt.colorbar();
    plt.show()

def get_MAP_obj(pt, equ_pts, measurements, num_pts): ##(1, 2)

    sigma_mat = np.array([[sigx**2, 0], [0, sigy**2]])

    prior = np.matmul(pt, np.linalg.inv(sigma_mat))
    prior = np.matmul(prior, pt.T)

    measure_sum = 0
    for equ_pt, r_i in zip(equ_pts, measurements):
        d_i = np.linalg.norm(pt - equ_pt)
```

```
        measure = (r_i - d_i)**2/sigi**2
        measure_sum += measure

    return prior + measure_sum

if __name__ == "__main__":

    sigx, sigy = 0.25, 0.25
    sigi = 0.3
    num_points = [1, 2, 3, 4]
    center = [0, 0]
    radius = 1
    quad_range = (-2, 2)
    num_grid_pts = 128

    true_pos = get_true_pos(center, radius)

    for num_pts in num_points:
        equ_pts = get_equidist_points(center, radius, num_pts)
        measurements = get_measurements(equ_pts, true_pos)
        print('measurements ',measurements)
        plot_equilevel_contours(equ_pts, measurements, quad_range, num_grid_pts)
```

References:

1. http://jrmeyer.github.io/machinelearning/2017/08/18/mle.html
2. https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html
3. https://numpy.org/
4. Referenced solutions of practise questions provided in the class.