Kashish Hetal Shah
002818647
Assignment - 2
ECE5644 – Introduction to Machine Learning and Pattern Recognition
ECE5644-Assignments/Assignment-2 at main · KashS28/ECE5644-Assignments (github.com)

QUESTION 1

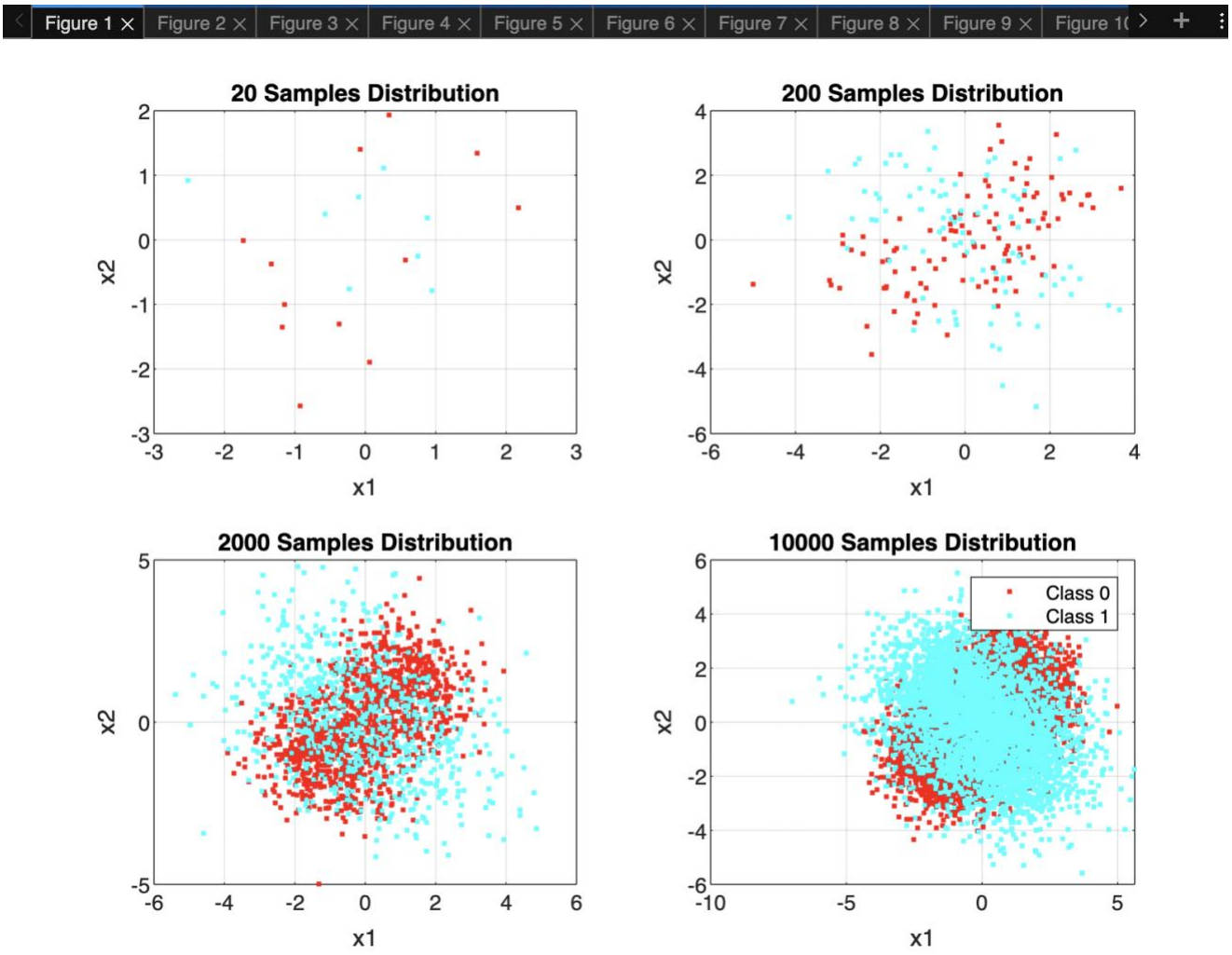


Fig.1.1: Datasets

$$(D=1) = \frac{P(x|L1)}{P(x|L0)} \geq \frac{\lambda_{10} - \lambda_{00}}{\lambda_{01} - \lambda_{11}} * \frac{P(L0)}{P(L1)} = \gamma \quad (D=0)$$

To minimize probability of misclassifications the cost for incorrect classifications should be 1 and the cost for correct classifications should be 0 which results in:

$$(D=1) = \frac{P(x|L1)}{P(x|L0)} \geq \frac{1-0}{1-0} \times \frac{0.6}{0.4} = 1.5 = \gamma \ (D=0)$$

Plots of the ROC with the calculated ideal minimum error point as well as the minimum error point estimated from the generated validation data is shown in Fig 1.2

The probability of errors versus Gamma with the calculated and estimate minimum error points marked are shown in Fig 1.3.

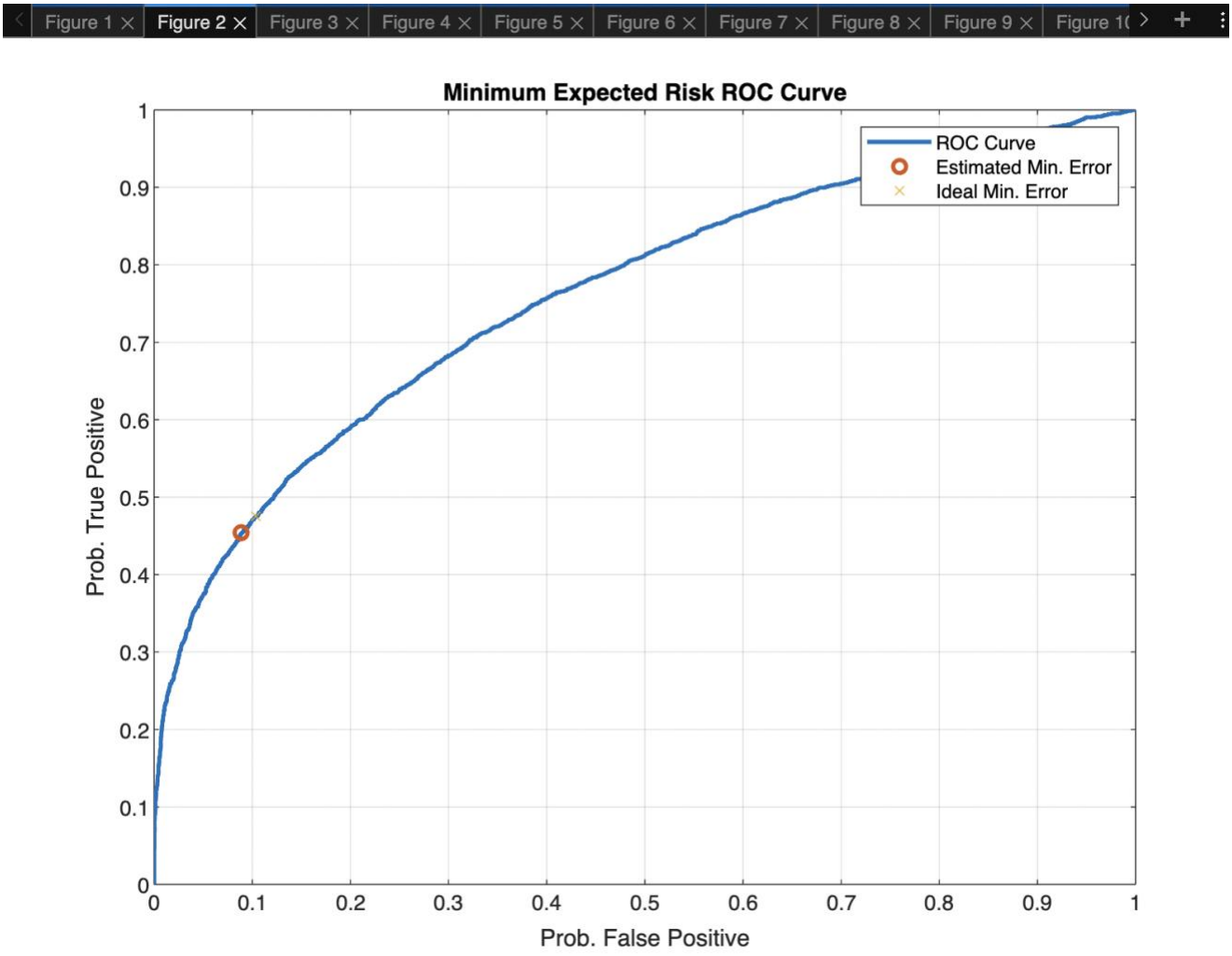


Fig 1.2: ROC Curve for Known Ideal Classification Case

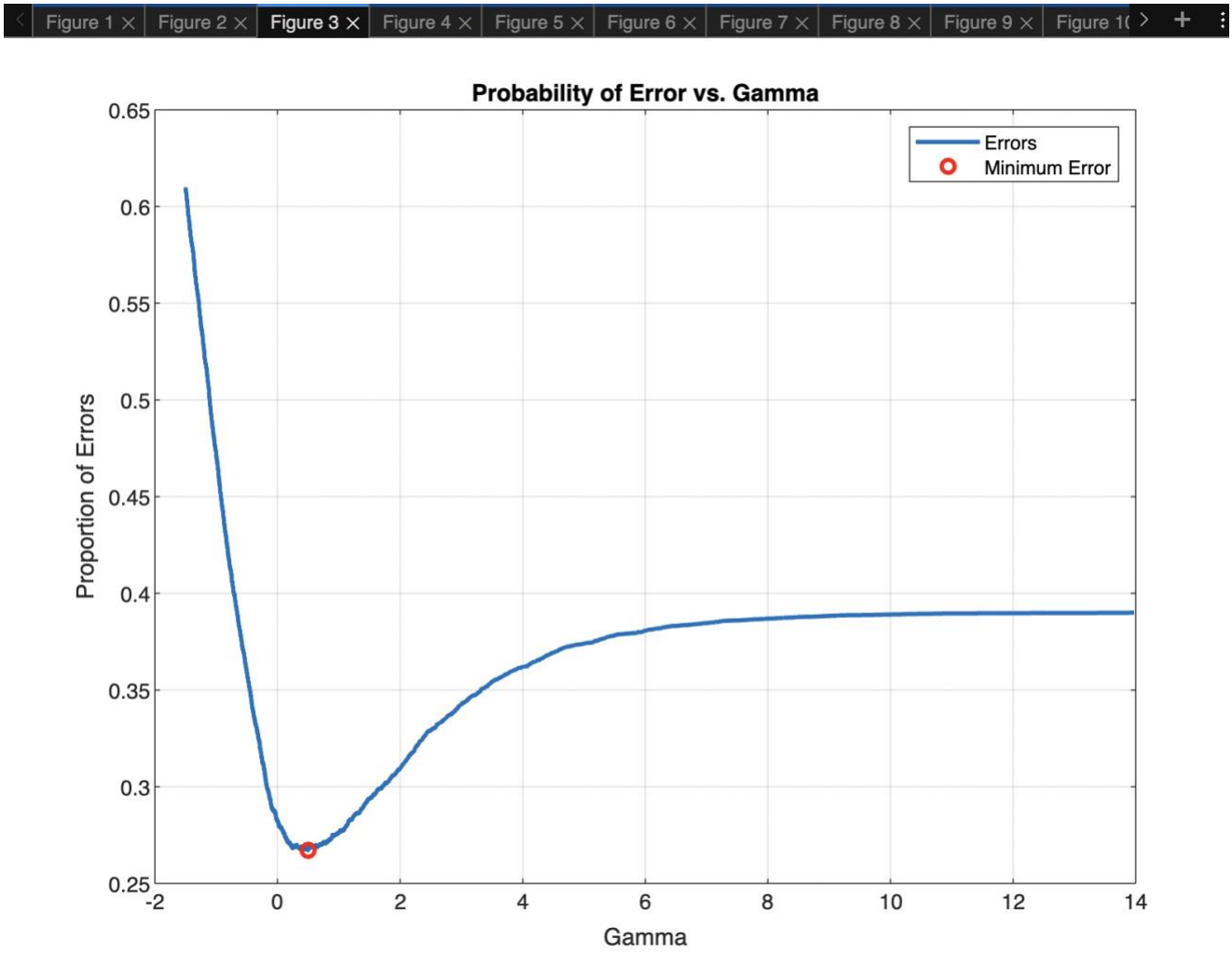


Fig 1.3: Probability of Error Curve for Ideal Classification

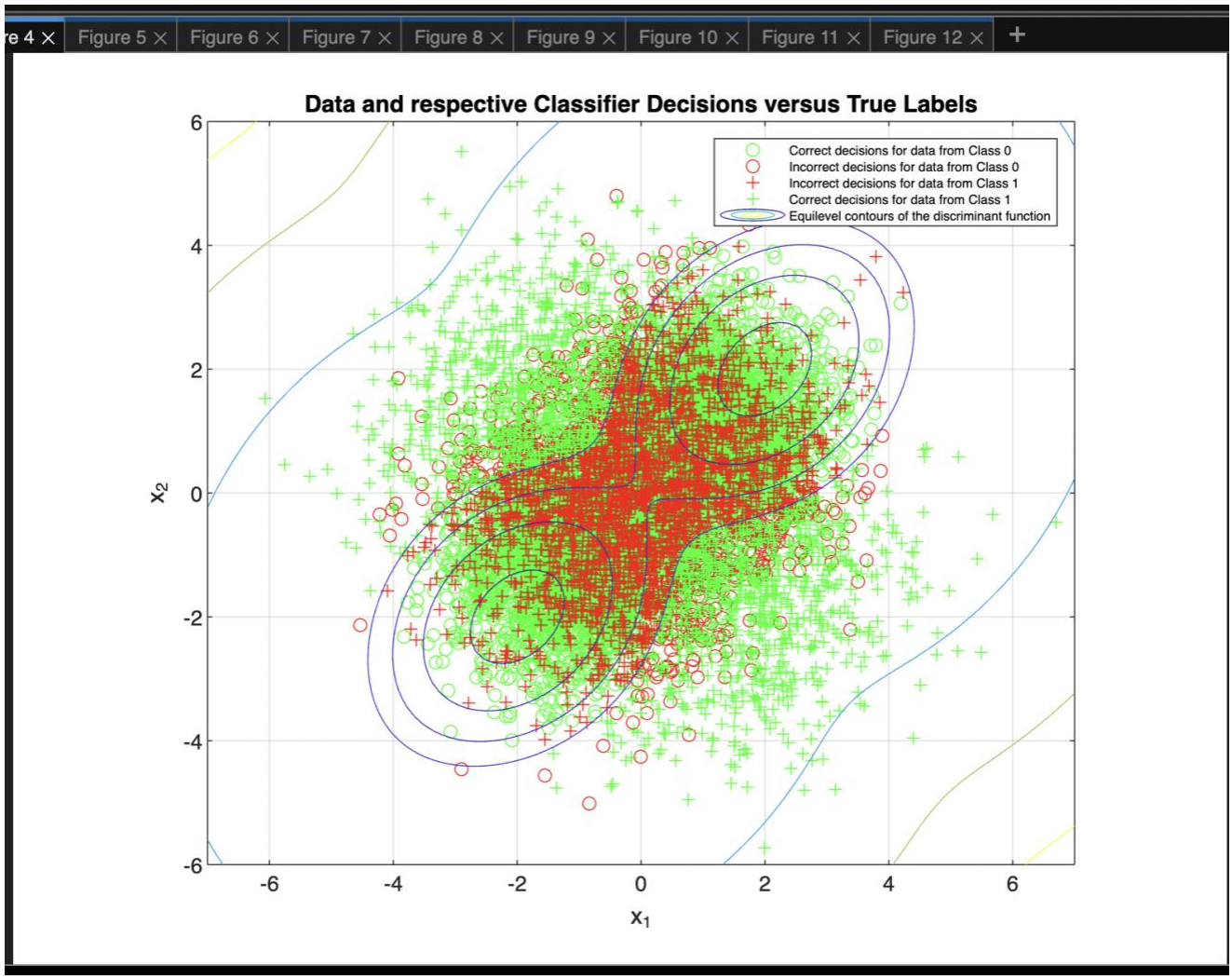


Fig 1.4: Decision Boundary of Ideal Classifier

The maximum likelihood parameter estimation techniques were employed to train logistic linear and logistic quadratic approximations for class label posterior functions using three distinct training datasets comprising 20, 200, and 2000 samples, respectively. These trained models were then utilized to classify samples from a validation dataset containing 10000 samples.

The logistic function is defined as follows:

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

Linear logistic function $z(x) = [1\ x_1\ x_2]^T$

Quadratic logistic function $z(x) = [1\ x_1\ x_2\ x_1^2\ x_1 x_2\ x_2^2]^T$

w vectors are estimated using numerical optimization techniques with the cost functions.

$$\Theta_{ML} = -\frac{1}{N}\sum_{1}^{N} t_n \log(h(x_n,\Theta)) + (1-t_n)\log(1-h(x_n,\Theta))$$

The minimum expected risk classification criteria are then.

$(t_n = 1)$ $w^T z(x) \geq 0$ $(t_n = 0)$

Table 1.1 presents a summary of the resulting probability of errors obtained by classifying the 10000-sample validation dataset using each of the three training datasets. The data illustrates a consistent trend: as the number of samples in the training datasets increases, the probabilities of error decrease for both the linear and quadratic estimation functions. Moreover, it's evident that the quadratic logistic function notably outperformed the linear logistic function across all cases.

Training Dataset	Linear	Quadratic
20	0.507	0.321
200	0.393	0.282
2000	0.40	0.279

Table 1.1: Logistic Function Probabilities of Error

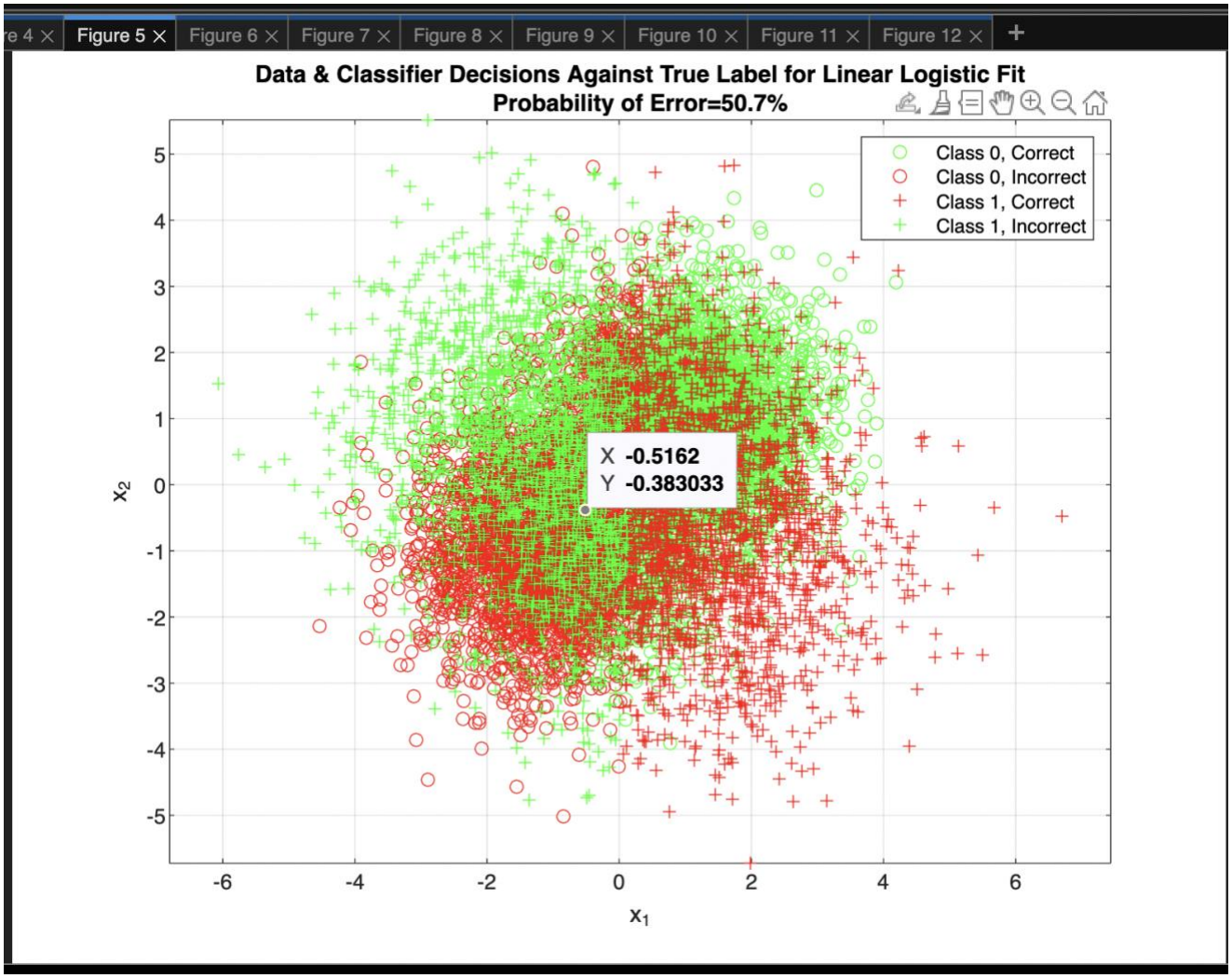


Fig 1.5.1 D20 Training Data - Linear Logistic Fit

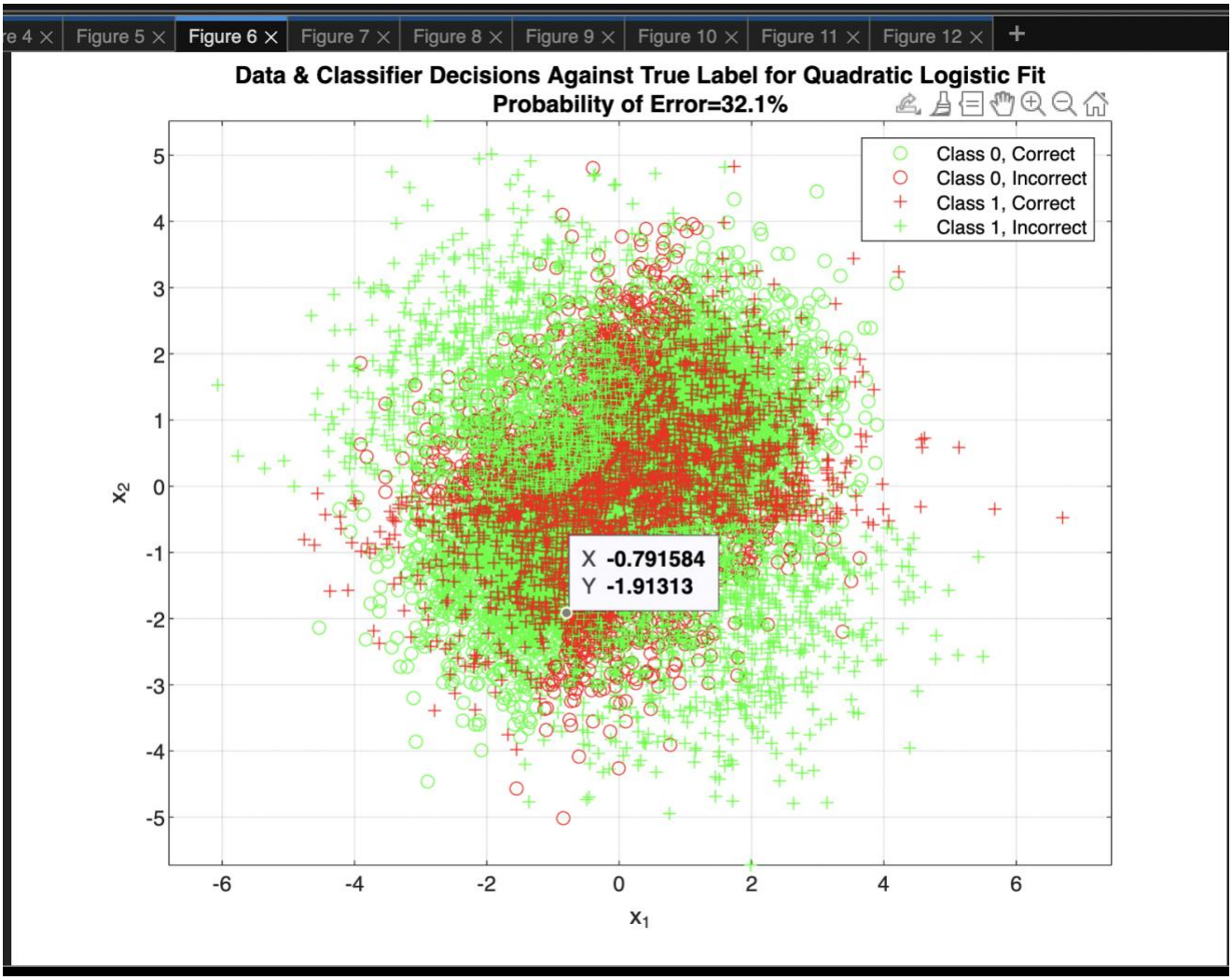


Fig 1.5.2 D20 Training Data - Quadratic Logistic Fit

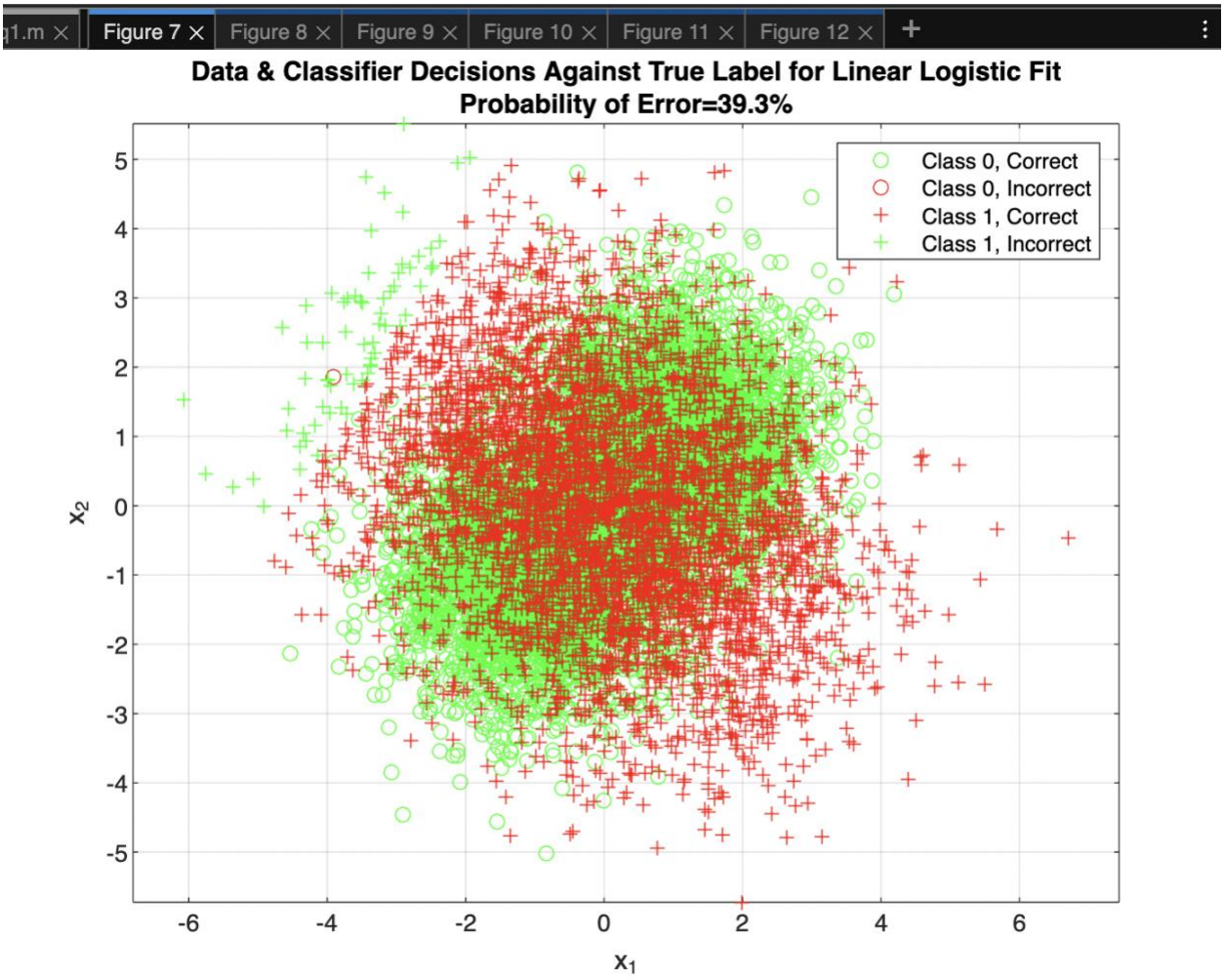


Fig 1.5.3 D200 Training Data - Linear Logistic Fit

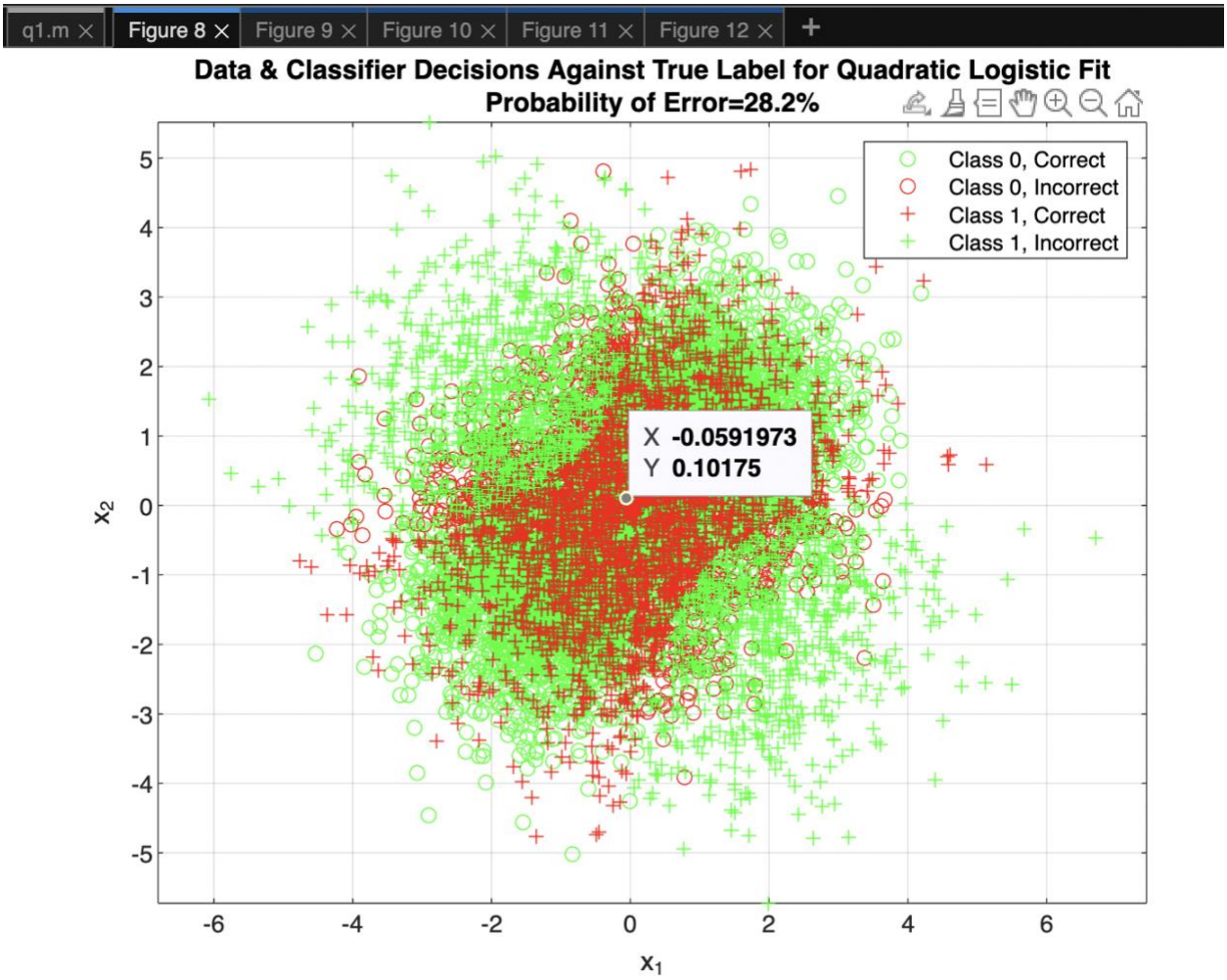


Fig 1.5.4 D200 Training Data - Quadratic Logistic Fit

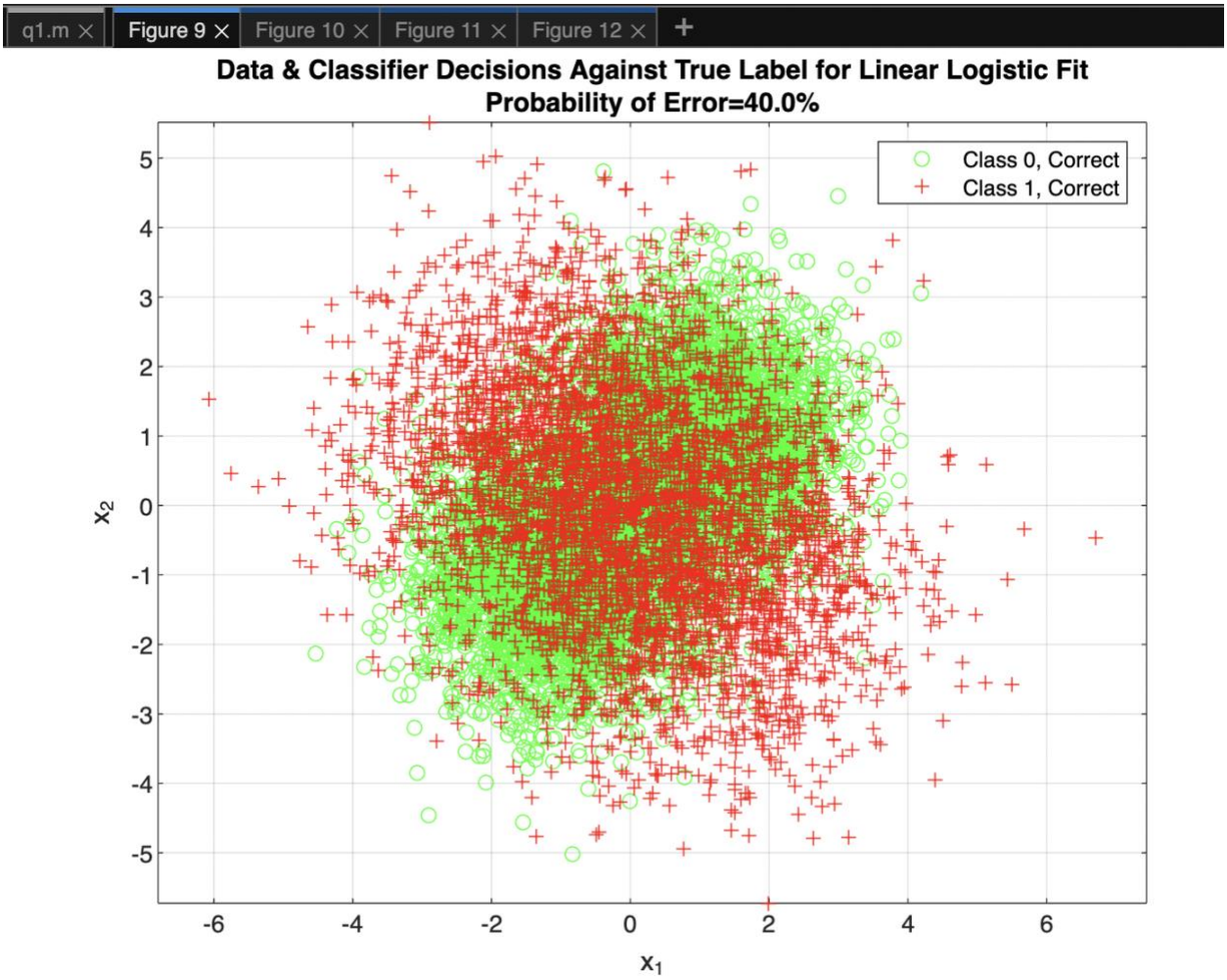


Fig 1.5.5 D2000 Training Data - Linear Logistic Fit

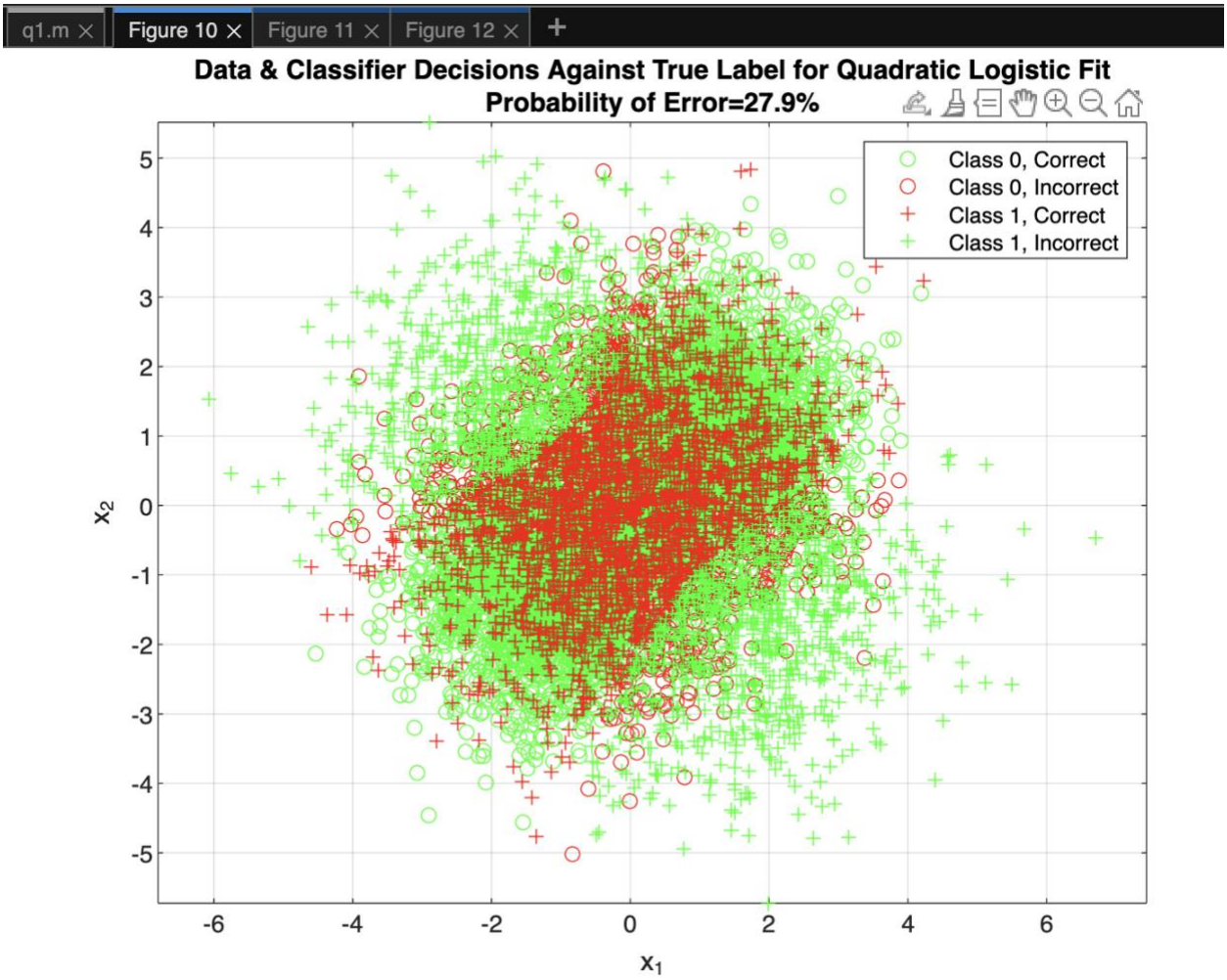
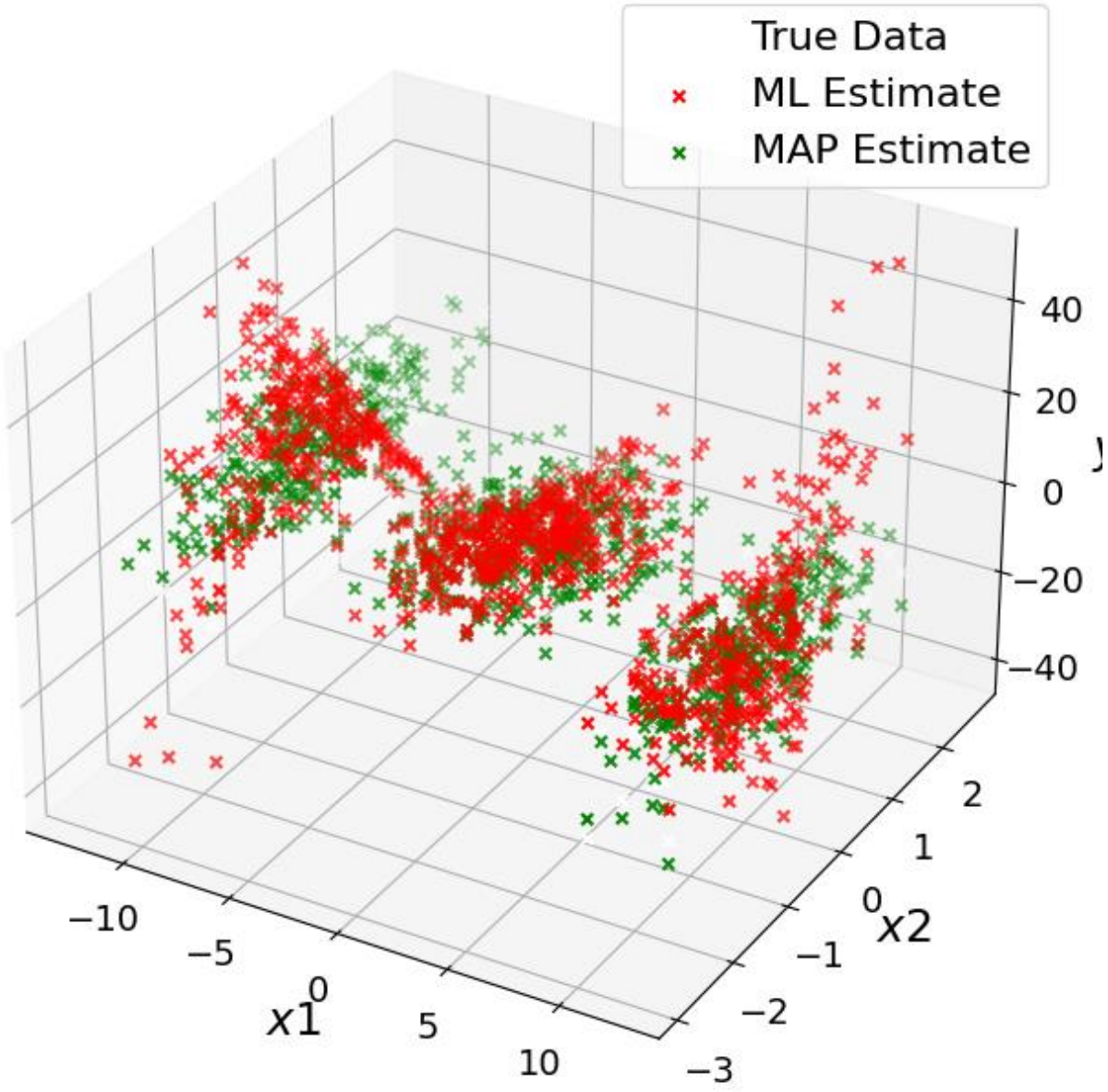
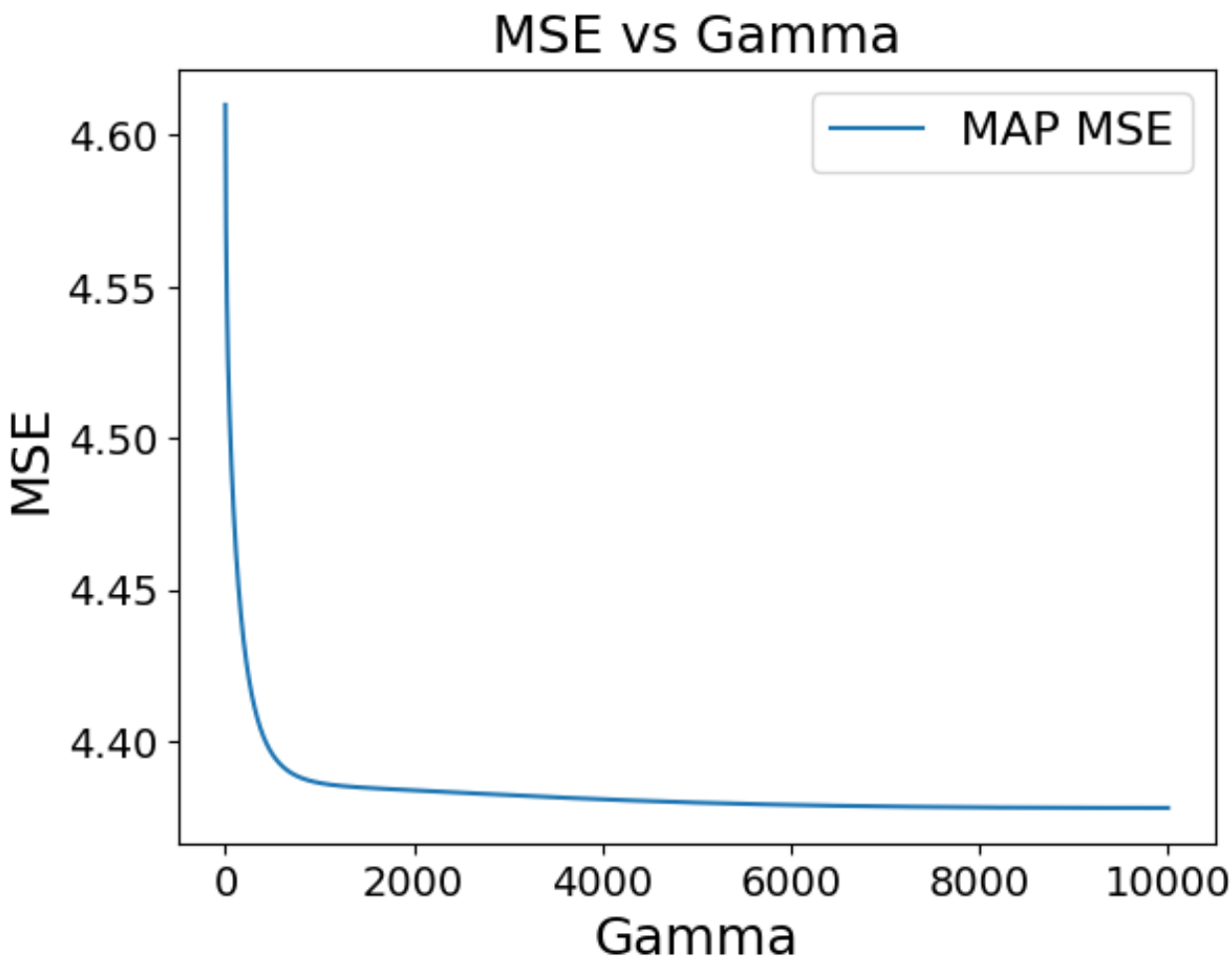
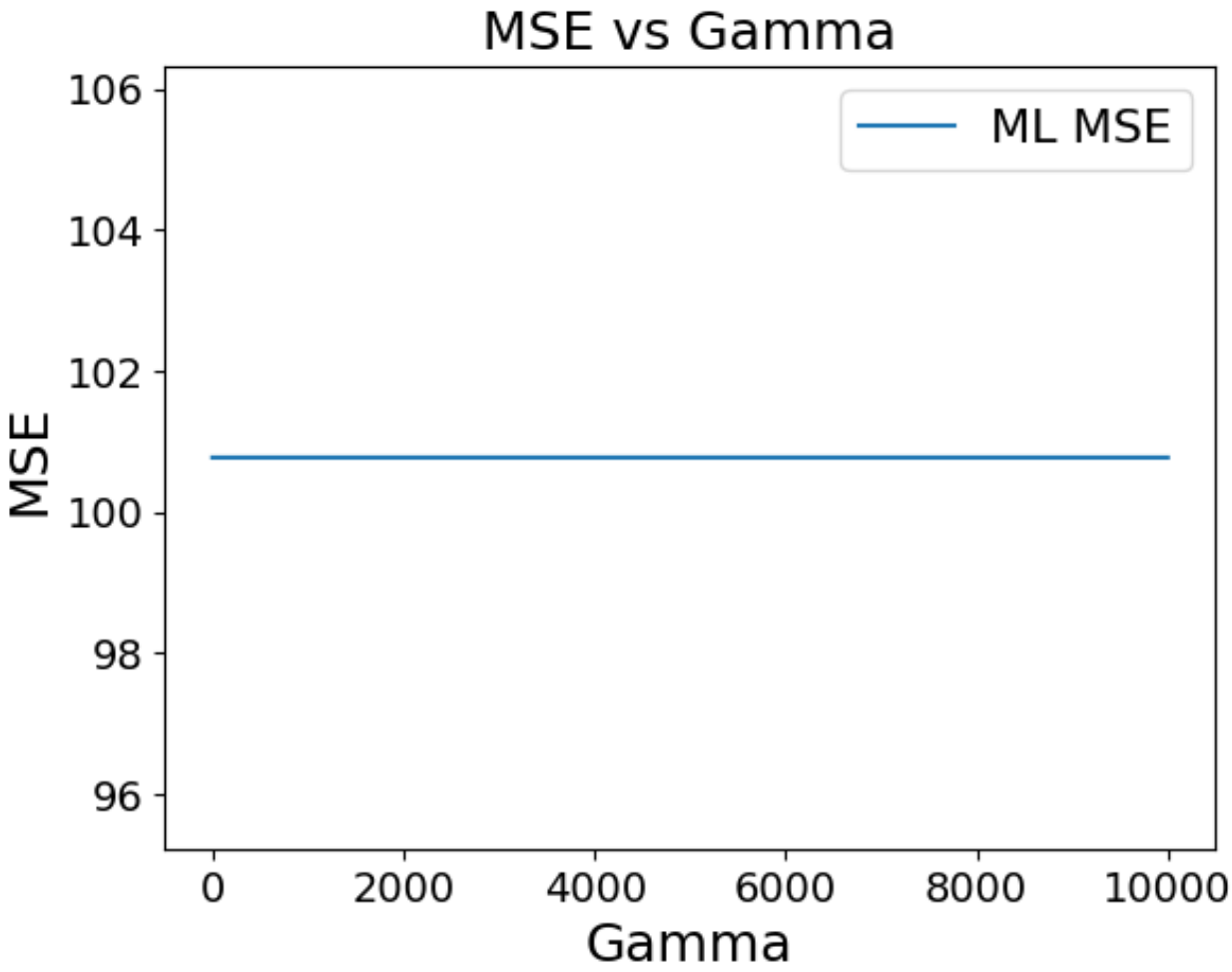


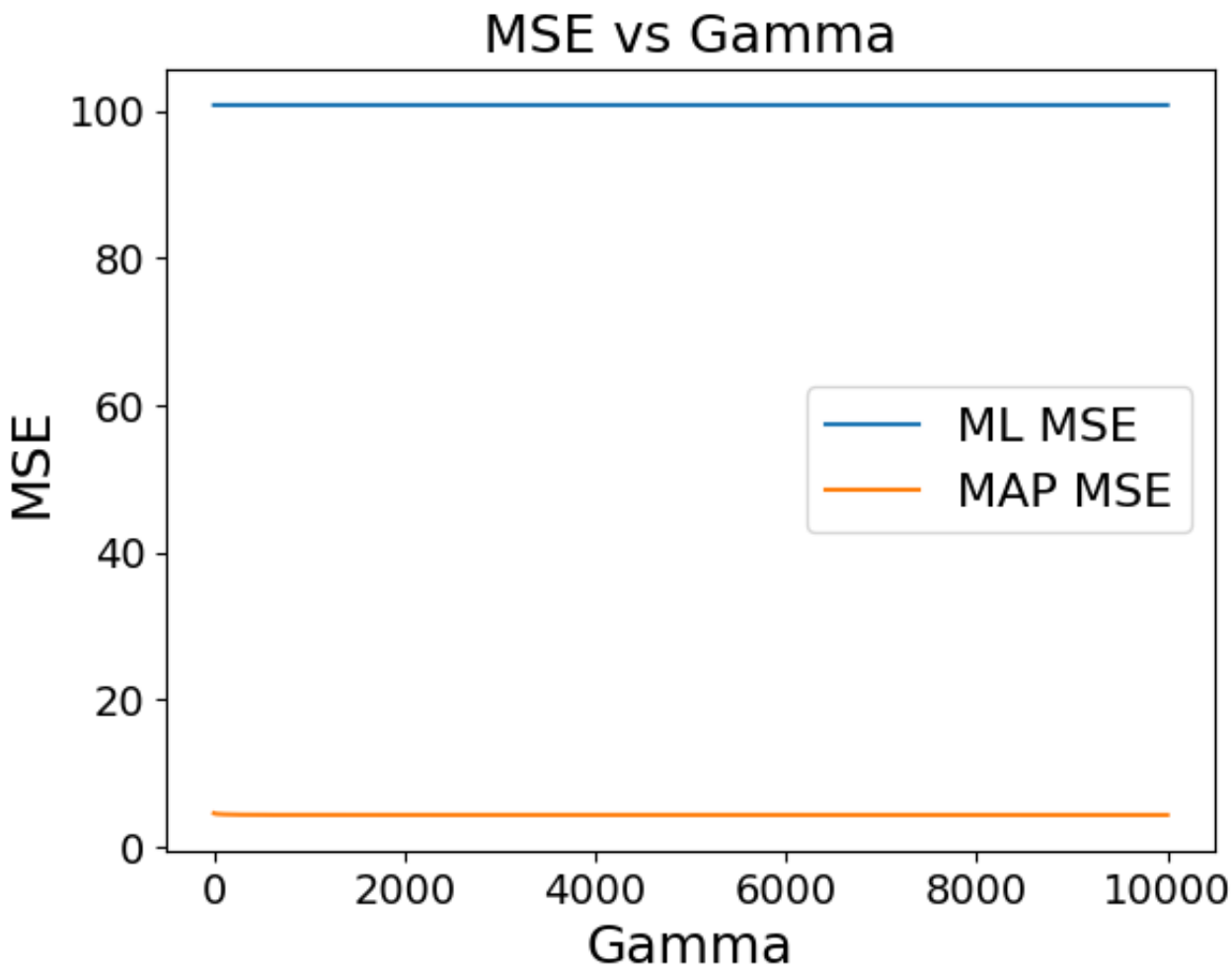
Fig 1.5.6 D2000 Training Data - Quadratic Logistic Fit

QUESTION 2


True Data
ML Estimate
MAP Estimate
40
20
0
−20
−40
y
−10
−5
0
5
10
x1
2
1
0
−1
−2
−3
x2


MSE vs Gamma
MAP MSE
4.60
4.55
4.50
4.45
4.40
MSE
0
2000
4000
6000
8000
10000
Gamma


MSE vs Gamma
ML MSE
106
104
102
100
98
96
MSE
0
2000
4000
6000
8000
10000
Gamma


MSE vs Gamma
MSE
Gamma
ML MSE
MAP MSE
0
20
40
60
80
100
0
2000
4000
6000
8000
10000

Question 2

The problem involves understanding the relationship between a scalar-real variable y and a two-dimensional real vector x, where y is modeled as $y = c(x, w) + v$.

Here, c denotes a cubic polynomial in x with coeffs w & v is a Gaussian random scalar with zero mean and variance σ^2.

Given a dataset D comprising N samples & in the form of $(x_1, y_1) \ldots (x_N, y_N)$, the assumption posits these samples as independently & identically distributed according to the aforementioned model.

From this dataset, two estimators for w are derived employing distinct approaches Maximum Likelihood (ML) & Maximum A Posteriori (MAP) parameter estimation techniques.

The ML Estimator is derived by maximizing the likelihood function $L(w|D)$ defined based on the assumed model & dataset.

Given the dataset $D = \{(x_1, y_1) \ldots (x_N, y_N)\}$ where $y = c(x, w) + v$ and assuming v is Gaussian with zero mean & variance σ^2, the likelihood function for the dataset can be expressed as the product of the conditional probabilities:

$$L(w|D) = \prod_{i=1}^{N} P(y_i | x_i, w) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(y_i - c(x_i, w))^2}{2\sigma^2}\right)$$

Maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood:

$$-\log L(w|D) = \frac{N}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - c(x_i, w))^2$$

In the case of MAP estimation, a prior assumption is introduced for w, stipulating a zero-mean Gaussian distribution with a covariance matrix of γI.

The MAP estimator is obtained by maximizing the posterior distribution considering this prior and the likelihood.

Incorporating the Gaussian Prior for w into the posterior distribution involves using Bayes Theorem. The posterior distribution of w given the dataset D is proportional to the likelihood multiplied by the prior:

$$p(w|D) \propto L(w) \cdot p(w)$$

Given the zero-mean Gaussian prior with covariance matrix γI, where I is identity matrix, the prior distribution of w is:

$$p(w) = \frac{1}{(2\pi)^{M/2} |\gamma I|^{1/2}} \exp\left(-\frac{1}{2} w^T (\gamma I)^{-1} w\right)$$

The MAP estimator maximizes the log-posterior:

$$w_{MAP} = \arg\max_w \left[\sum_{i=1}^{N} \left(-\frac{(y_i - c(x_i, w))^2}{2\sigma^2} - \frac{1}{2} \log\left(2\pi\sigma^2\right) \right) - \frac{1}{2} w^T (\gamma I)^{-1} w \right]$$

This formulation incorporates both the likelihood and the prior, leading to a more ~~regulation~~ regularized estimate by penalizing large values of w due to the prior term.

~~The~~

Gradient descent is a foundational optimization technique in machine learning used to minimize a cost function by iteratively adjusting model parameters. It operates by computing the ~~grad~~ gradient, the direction of steepest ascent, and updates the parameters in the opposite direction to minimize ~~the~~ the cost.

This iterative process continues until convergence or a stopping criterion is met. It's sensitive to the learning rate, influencing the step size in parameter updates, impacting convergence speed and accuracy.

Gradient descent's versatility makes it a fundamental tool for training models across various domains, offering an efficient approach for optimizing complex functions & enhancing ~~text~~ model performance through parameter adjustments.

QUESTION 3

Question 3:

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.

$$\begin{bmatrix} x_{MAP} \\ y_{MAP} \end{bmatrix} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} P\left(\begin{bmatrix} x \\ y \end{bmatrix} \mid \{r_1 \dots r_k\}\right)$$

$$= \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \left((2\pi\sigma_x\sigma_y)^{-1} e^{-\frac{1}{2}[x\ y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1}\begin{bmatrix} x \\ y \end{bmatrix}} \right) + \sum_{i=1}^{k} \log P\left(\begin{bmatrix} x \\ y \end{bmatrix} \mid r_i\right)$$

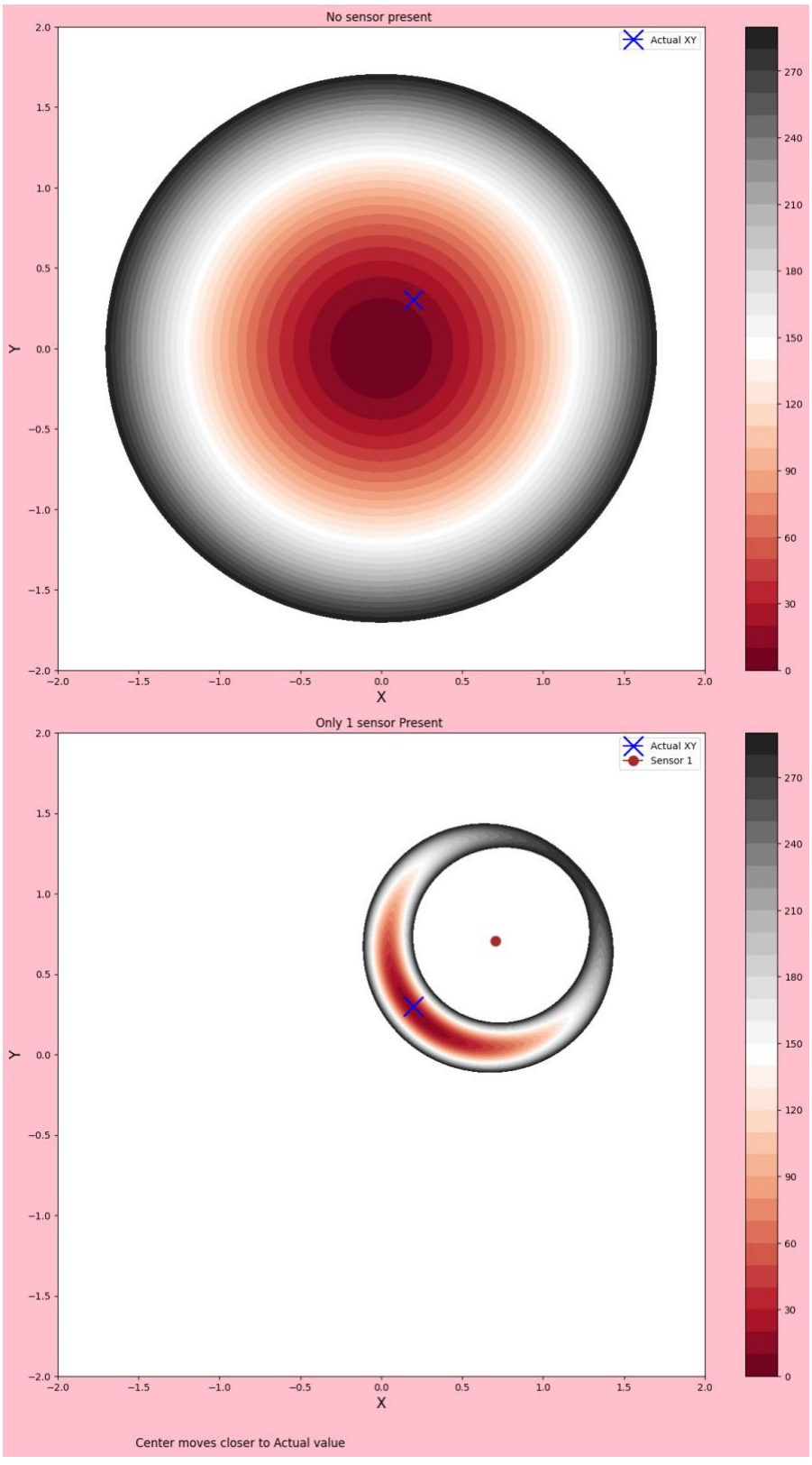
$$= \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \frac{-1}{2}[x\ y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \sum_{i=1}^{k} \log N(r_i \mid 0, \sigma_i^2)$$

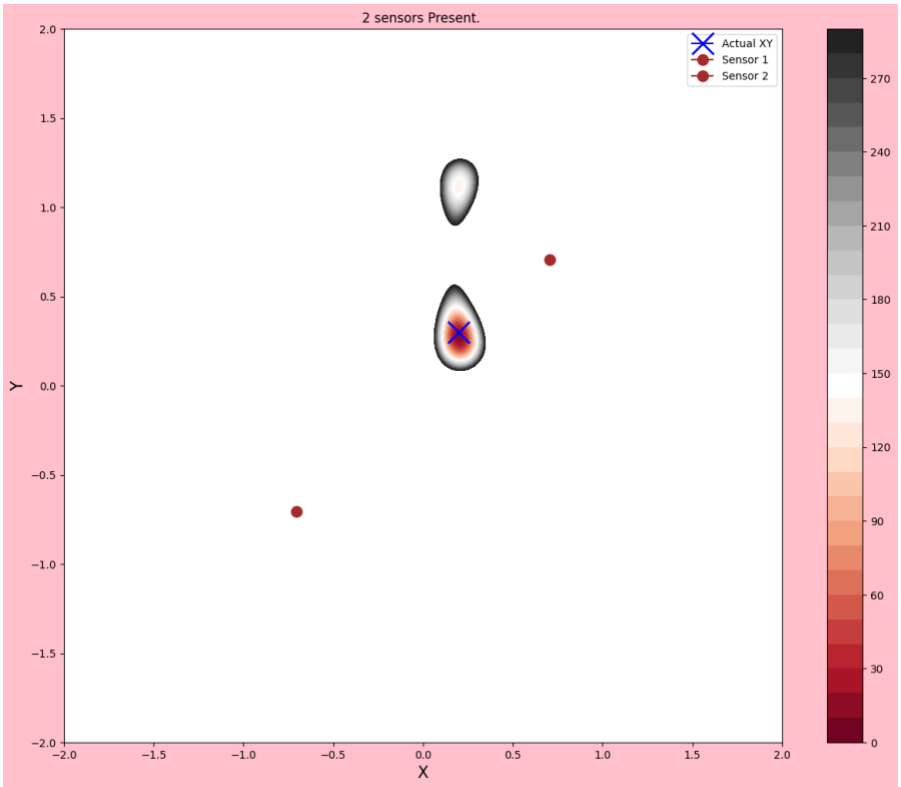
$$= \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \frac{-1}{2}[x\ y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \sum_{i=1}^{k} \log\left((2\pi\sigma_i^2)^{-\frac{1}{2}} e^{-\frac{(\|x_i - d_i\| - 0)^2}{2\sigma_i^2}}\right)$$

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

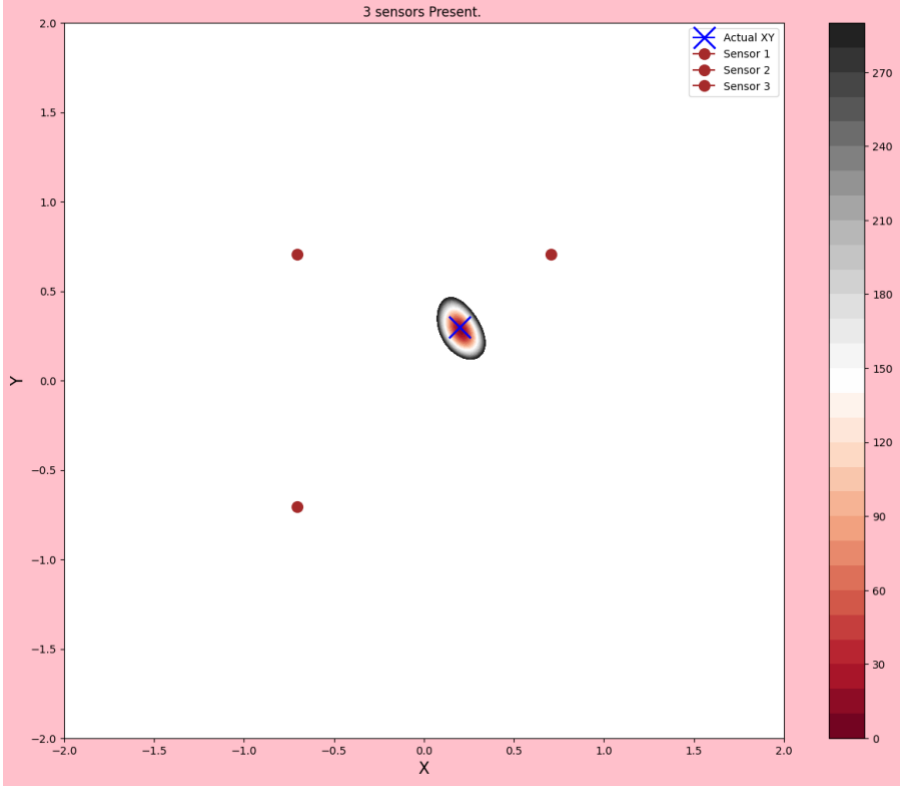
$$= \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmin}} [x\ y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1}\begin{bmatrix} x \\ y \end{bmatrix} + \sum_{i=1}^{k} \frac{(r_i - d_i)^2}{\sigma_i^2}$$

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

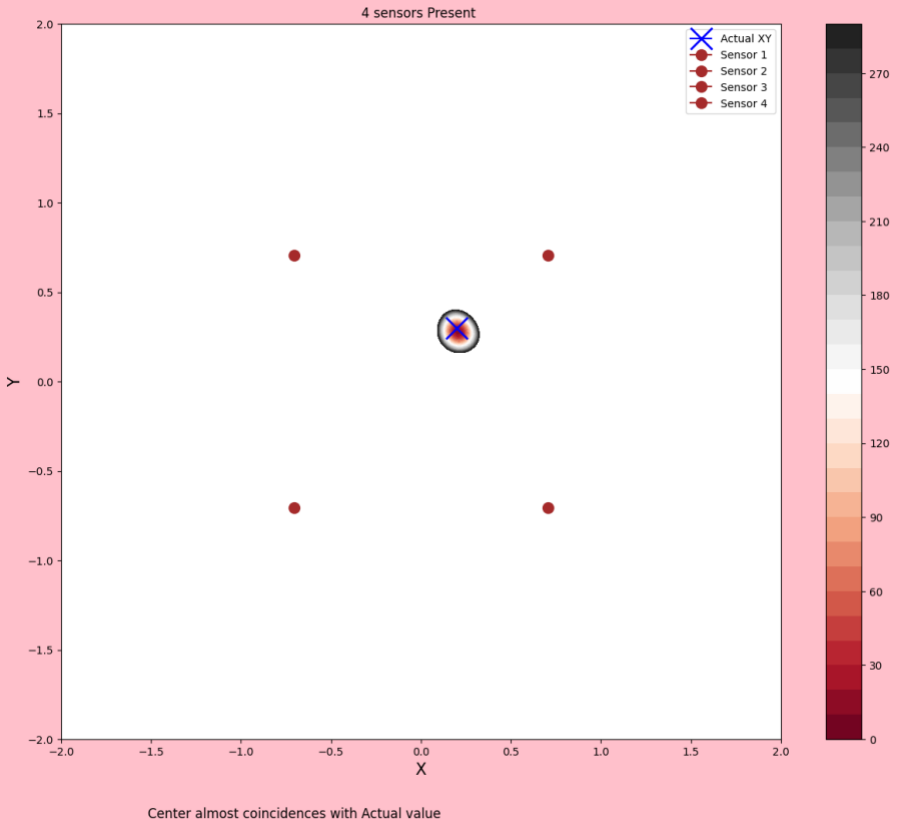

No sensor present
Actual XY
Y
X
Only 1 sensor Present
Actual XY
Sensor 1
Y
X
Center moves closer to Actual value



Center moves closer to Actual value



Center moves very close to Actual value



The code generates true vehicle coordinates as a random pair situated within the unit circle at the origin. It proceeds by calculating the distances from this true vehicle location based on evenly spaced landmarks around the circle, denoted by the value K. These distances are adjusted with additive Gaussian noise, following which the code computes the Maximum A Posteriori (MAP) estimation objective function for each point on a 128x128 mesh grid. These values are then plotted as equipluve contours.

Moreover, the code visualizes the landmark locations (depicted as green circles), their respective provided ranges, the unit circle (represented by a gray dot), and the accurate location (indicated by a red "+"). Faint blue circles are shown around their respective landmarks for clarity.

When all landmarks and the previous bias possess a y-coordinate of 0, the accuracy of the MAP estimate for K < 3 diminishes. The estimates become symmetric around the x-axis. However, for K = 3 and K = 4, the estimate significantly improves in precision. The contour graph highlights this enhancement, illustrating that for K = 3 and K = 4, the true location is merely two and one contour levels away, respectively, from the central estimate contour. Generally, with a rise in K, the MAP estimate's precision increases.

However, exceptions occur; for instance, in the transition from K = 1 to K = 2, one of the landmark's range measurements might be low enough to override the more accurate

estimate by the new second landmark. Nonetheless, as K grows significantly larger, this trend becomes more pronounced.

The contour graph allows the assessment of the estimator's accuracy by measuring the distance between the true location and the point corresponding to the lowest contour, typically positioned near the center of the innermost contour.
As K increases, the certainty of the estimator also escalates. This increased certainty is reflected in the contour graphs as a reduction in the area of locations with a high probability. Although less noticeable for smaller K values, following a single contour level as K increases (to about 40) vividly demonstrates this phenomenon.

QUESTION 4

Q4 - Assignment 2

⇒ Bayes classifiers minimize conditional risk defined as:

$$R(\alpha_i|x) = \sum_{j=1}^{c} \lambda(\alpha_i|\omega_j) P(\omega_j|x) \quad \text{for } i = 1, \ldots, c \text{ (multiclass case)}$$

Based on the $\lambda(\alpha_i|\omega_j)$ (condition loss) definition given,

$$R(\alpha_i|x) = \sum_{\substack{j=1 \\ j \neq i}}^{c} \lambda_s \cdot P(\omega_j|x) \quad \text{for } i = 1, \ldots, c.$$

$$= \lambda_s \underbrace{\sum_{\substack{j=1 \\ j \neq i}}^{c} P(\omega_j|x)}$$

$$= \lambda_s [1 - P(\omega_i|x)] \quad \text{for } i = 1, \ldots, c.$$

For $i = c+1$; $R(\alpha_{c+1}|x) = \lambda_r$

⇒ Min. risk is achieved if we decide ω_i if $R(\alpha_i|x) \leq R(\alpha_{c+1}|x)$

$$R(\alpha_i|x) \leq R(\alpha_{c+1}|x)$$

$$\lambda_s [1 - P(\omega_i|x)] \leq \lambda_r$$

$$P(\omega_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$$ decide ω_i and reject otherwise

* if $\lambda_r = 0$, then we always ~~regret~~ reject
* if $\lambda_r \geq \lambda_s$, we will never reject

Conclusion:

As mentioned before, the risk of action α_i must be lower than the risk of rejection (this is a necessary condition) we obtain $P(w_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$ ——— (1)

- Given result ——(1), if λ_r is 0, the decision will always be rejected for all the values of x.

- The higher the λ_r with respect to λ_s, we have less chances of rejecting the decision.

- If $\lambda_r = \lambda_s$, the class that will be selected is the class with the maximum posterior probability given

$$P(w_i|x) \geq P(w_j|x)$$
$$\forall j = 1, \ldots, c \text{ and } j \neq i$$

QUESTION 5

Question 5 - Assignment 2.

Let independent identically distributed (iid) samples be drawn from dataset $D = z_1 \dots z_n$ with $z_i = 1, 2, \dots k$.
Let the parameter vector for the pdf be $\theta = [\theta_1 \dots \theta_k]^T$, where $p(z_k = 1) = \theta_k$ is subject to constraints $\theta \geq 0$ and $\theta^T 1 = 1$

The MLP estimation is given by $\theta_{ML} = \arg\max_\theta \log[p(D|\theta)]$.
Considering iid samples and substituting $\theta_{z_i} = p(z_i|\theta)$.

$$\Rightarrow \quad \theta_{ML} = \arg\min_\theta -\frac{1}{N} \sum_{i=1}^{N} \log \theta_{z_i}.$$

We assume the inequality constraint will be inactive at the solution i.e. $\theta_k > 0$, and we take the Langrangian considering the equality constraint: $L(\theta, \lambda) = -\frac{1}{N} \sum_{i=1}^{N} \ln \theta_{z_i} - \lambda(1 - \theta^T 1)$.

We set the derivative to zero to find the solution: $0 = \nabla_{\theta, \lambda} L(\theta, \lambda)$

$$= -\frac{1}{N} \sum_{i=1}^{N} \frac{\delta}{\delta \theta_k} \log \theta_{z_i} + \lambda \frac{\delta}{\delta \theta_k} (\theta^T 1),$$

Using the kronecker's delta $\delta_{z_i} = \begin{cases} 0 & \text{if } z \neq l, \\ 1 & \text{if } z = l \end{cases}$

$$\nabla_{\theta_k} L(\theta, \lambda) = -\frac{1}{N} \sum_{i=1}^{N} \frac{1}{\theta_l} \delta_{z_i l} + \lambda = -\frac{N_l}{N \theta_l} + \lambda = 0$$

We find $\theta_i, \lambda = \frac{N_i}{N}$ and by summing all k terms:

$$\lambda(\theta_1 + \ldots + \theta_k) = \frac{N_1 + \ldots + Nk}{N} \Rightarrow \lambda = 1 \Rightarrow \theta_i = \frac{N_i}{N} = \theta_{ML}$$

We now find the maximum a posteriori parameter estimation assuming $p(\theta)$ defined by the Dirichlet distribution with hyperparameter α, $P(\theta|\alpha) = \frac{1}{B(\alpha)} \prod_{k=1}^{K} \theta_k^{\alpha k - 1}$,

where B is a normalization constant that ensures the prior integrates to 1. The estimation is defined by

$$\theta_{MAP} = \underset{\theta}{\operatorname{argmax}} \log p(\theta|D)$$

Using Bayes Rule,

$$\theta_{MAP} = \underset{\theta}{\operatorname{argmax}} \log \frac{P(D|\theta)p(\theta)}{P(D)} = \underset{\theta}{\operatorname{argmax}} \log p(D|\theta) + \log p(\theta)$$

where we omit $p(D)$ as it's not dependent on θ.

Substituting in the Dirichlet distribution prior,

$$\theta_{MAP} = \underset{\theta}{\operatorname{argmax}} \frac{1}{N} \log \theta_{zi} + \log \left[\frac{\prod_{k=1}^{k} \theta^{\alpha k - 1} k}{B(\alpha)} \right]$$

We take the Lagrangian and use the same procedure used for the ML derivation to find,

$$\theta_i, \lambda = \frac{N_i}{N} + \alpha^T 1 - k$$

By summing all terms, we find $\lambda = \alpha^T 1 + 1 - k$. We substitute in the eqn above to conclude:

$$\theta_{MAP} = \frac{\frac{N_i}{N} + \alpha^T 1 - k}{\alpha^T 1 - k + 1}$$

APPENDIX

QUESTION 1

```
clear all;
close all;

Part1 = 1;
Part2 = 1;
dimension = 2;

% Defining different sample sizes
D.d100.N = 20;
D.d1000.N = 200;
D.d10k.N = 2000;
D.d20k.N = 10000;
DType = fieldnames(D);

% Defining parameters for GMM
p = [0.6, 0.4];

mu0 = [-1, -1; 1, 1]';
Sigma0(:,:,1) = [1, 0; 0, 1];
Sigma0(:,:,2) = [1, 0; 0, 1];
alpha0 = [0.5, 0.5];

mu1 = [-1, 1; 1, -1]';
Sigma1(:,:,1) = [2, 0; 0, 2];
Sigma1(:,:,2) = [2, 0; 0, 2];
alpha1 = [0.5, 0.5];

figure;

% Generating data points based on GMM, assigning labels for classes plotting the data
points in different colors for each class
for ind = 1:length(DType)
    D.(DType{ind}).x = zeros(dimension, D.(DType{ind}).N);

    D.(DType{ind}).labels = rand(1, D.(DType{ind}).N) >= p(1);
    D.(DType{ind}).N0 = sum(~D.(DType{ind}).labels);
    D.(DType{ind}).N1 = sum(D.(DType{ind}).labels);
    D.(DType{ind}).phat(1) = D.(DType{ind}).N0 / D.(DType{ind}).N;
    D.(DType{ind}).phat(2) = D.(DType{ind}).N1 / D.(DType{ind}).N;

    [D.(DType{ind}).x(:, ~D.(DType{ind}).labels), ...
     D.(DType{ind}).dist(:, ~D.(DType{ind}).labels)] = ...
        randGMM(D.(DType{ind}).N0, alpha0, mu0, Sigma0);
```

```
    [D.(DType{ind}).x(:, D.(DType{ind}).labels), ...
     D.(DType{ind}).dist(:, D.(DType{ind}).labels)] = ...
        randGMM(D.(DType{ind}).N1, alpha1, mu1, Sigma1);

    subplot(2, 2, ind);
    plot(D.(DType{ind}).x(1, ~D.(DType{ind}).labels), ...
         D.(DType{ind}).x(2, ~D.(DType{ind}).labels), 'r.', 'DisplayName', 'Class 0');
    hold all;
    plot(D.(DType{ind}).x(1, D.(DType{ind}).labels), ...
         D.(DType{ind}).x(2, D.(DType{ind}).labels), 'c.', 'DisplayName', 'Class 1');
    grid on;
    xlabel('x1');
    ylabel('x2');
    title([num2str(D.(DType{ind}).N) ' Samples Distribution']);
end

legend 'show';

% Evaluating GMMs for a specific sample size, Calculating discriminant scores,
probabilities, & error metrics, Plotting ROC curve, minimum error point, and classifier
decisions
if Part1
    px0 = evalGMM(D.d20k.x, alpha0, mu0, Sigma0);
    px1 = evalGMM(D.d20k.x, alpha1, mu1, Sigma1);
    discScore = log(px1 ./ px0);
    sortDS = sort(discScore);

    logGamma = [min(discScore) - eps, sort(discScore) + eps];
    prob = CalcProb(discScore, logGamma, D.d20k.labels, D.d20k.N0, D.d20k.N1,
D.d20k.phat);
    logGamma_ideal = log(p(1) / p(2));
    decision_ideal = discScore > logGamma_ideal;
    p10_ideal = sum(decision_ideal == 1 & D.d20k.labels == 0) / D.d20k.N0;
    p11_ideal = sum(decision_ideal == 1 & D.d20k.labels == 1) / D.d20k.N1;
    pFE_ideal = (p10_ideal * D.d20k.N0 + (1 - p11_ideal) * D.d20k.N1) / (D.d20k.N0 +
D.d20k.N1);

    [prob.min_pFE, prob.min_pFE_ind] = min(prob.pFE);
    if length(prob.min_pFE_ind) > 1
        [~, minDistTheory_ind] = min(abs(logGamma(prob.min_pFE_ind) -
logGamma_ideal));
        prob.min_pFE_ind = prob.min_pFE_ind(minDistTheory_ind);
    end

    minGAMMA = exp(logGamma(prob.min_pFE_ind));
    prob.min_FP = prob.p10(prob.min_pFE_ind);
```

```
    prob.min_TP = prob.p11(prob.min_pFE_ind);

    plotROC(prob.p10, prob.p11, prob.min_FP, prob.min_TP);
    hold all;
    plot(p10_ideal, p11_ideal, 'x', 'DisplayName', 'Ideal Min. Error');
    plotMinPFE(logGamma, prob.pFE, prob.min_pFE_ind);
    plotDecisions(D.d20k.x, D.d20k.labels, decision_ideal);

    plotERMContours(D.d20k.x, alpha0, mu0, Sigma0, alpha1, mu1, Sigma1,
logGamma_ideal);
end

% Part 2: Classification with MLP Estimation

options = optimset('MaxFunEvals', 60000, 'MaxIter', 20000);

% Performing MLP Estimation for linear and quadratic logistic fits, calculating decision
scores, probabilities, and classifier decisions
% Plot data points with classifier decisions for both linear and quadratic fits
for ind = 1:length(DType)
    lin.x = [ones(1, D.(DType{ind}).N); D.(DType{ind}).x];
    lin.init = zeros(dimension + 1, 1);

    lin.theta = fminsearch(@(theta)(costFun(theta, lin.x, D.(DType{ind}).labels)), lin.init,
options);
    lin.discScore = lin.theta' * [ones(1, D.d20k.N); D.d20k.x];
    gamma = 0;
    lin.prob = CalcProb(lin.discScore, gamma, D.d20k.labels, D.d20k.N0, D.d20k.N1,
D.d20k.phat);

    quad.x = [ones(1, D.(DType{ind}).N); D.(DType{ind}).x;...
            D.(DType{ind}).x(1, :).^2;...
            D.(DType{ind}).x(1, :).*D.(DType{ind}).x(2, :);...
            D.(DType{ind}).x(2, :).^2];
    quad.init = zeros(2*(dimension + 1), 1);

    quad.theta = fminsearch(@(theta)(costFun(theta, quad.x, D.(DType{ind}).labels)),
quad.init, options);
    quad.xScore = [ones(1, D.d20k.N); D.d20k.x; D.d20k.x(1, :).^2;...
                D.d20k.x(1, :).*D.d20k.x(2, :); D.d20k.x(2, :).^2];
    quad.discScore = quad.theta' * quad.xScore;
    gamma = 0;
    quad.prob = CalcProb(quad.discScore, gamma, D.d20k.labels, D.d20k.N0,
D.d20k.N1, D.d20k.phat);

    plotDecisions(D.d20k.x, D.d20k.labels, lin.prob.decisions);
```

```
    title(sprintf('Data & Classifier Decisions Against True Label for Linear Logistic
Fit\nProbability of Error=%1.1f%%', 100*lin.prob.pFE));

    plotDecisions(D.d20k.x, D.d20k.labels, quad.prob.decisions);
    title(sprintf('Data & Classifier Decisions Against True Label for Quadratic Logistic
Fit\nProbability of Error=%1.1f%%', 100*quad.prob.pFE));
end

% Computing the cost function for logistic regression
function cost = costFun(theta, x, labels)
    h = 1./(1 + exp(-x' * theta));
    cost = -1/length(h) * sum((labels' .* log(h) + (1 - labels)' .* (log(1 - h))));
end

% Generating random data points based on a Gaussian Mixture Model
function [x, labels] = randGMM(N, alpha, mu, Sigma)
    d = size(mu, 1);
    cum_alpha = [0, cumsum(alpha)];
    u = rand(1, N);
    x = zeros(d, N);
    labels = zeros(1, N);

    for m = 1:length(alpha)
        ind = find(cum_alpha(m) < u & u <= cum_alpha(m + 1));
        x(:, ind) = randGaussian(length(ind), mu(:, m), Sigma(:, :, m));
        labels(ind) = m - 1;
    end
end

% Generating random data points following a Gaussian distribution
function x = randGaussian(N, mu, Sigma)
    n = length(mu);
    z = randn(n, N);
    A = Sigma^(1/2);
    x = A * z + repmat(mu, 1, N);
end

% Evaluating the GMM for given data points
function gmm = evalGMM(x, alpha, mu, Sigma)
    gmm = zeros(1, size(x, 2));

    for m = 1:length(alpha)
        gmm = gmm + alpha(m) * evalGaussian(x, mu(:, m), Sigma(:, :, m));
    end
end

% Evaluating the Gaussian function for given data points
```

```
function g = evalGaussian(x, mu, Sigma)
    [n, N] = size(x);
    invSigma = inv(Sigma);
    C = (2*pi)^(-n/2) * det(invSigma)^(1/2);
    E = -0.5 * sum((x - repmat(mu, 1, N)) .* (invSigma * (x - repmat(mu, 1, N))), 1);
    g = C * exp(E);
end

% Calculating probabilities and error metrics based on decision scores
function prob = CalcProb(discScore, logGamma, labels, N0, N1, phat)
    for ind = 1:length(logGamma)
        prob.decisions = discScore >= logGamma(ind);
        Num_pos(ind) = sum(prob.decisions);
        prob.p10(ind) = sum(prob.decisions == 1 & labels == 0) / N0;
        prob.p11(ind) = sum(prob.decisions == 1 & labels == 1) / N1;
        prob.p01(ind) = sum(prob.decisions == 0 & labels == 1) / N1;
        prob.p00(ind) = sum(prob.decisions == 0 & labels == 0) / N0;
        prob.pFE(ind) = prob.p10(ind) * phat(1) + prob.p01(ind) * phat(2);
    end
end

% Plots contours for estimated GMMs
function plotContours(x, alpha, mu, Sigma)
    figure

    if size(x, 1) == 2
        plot(x(1, :), x(2, :), 'b.');
        xlabel('x1');
        ylabel('x2');
        title('Data and Estimated GMM Contours');
        axis equal;
        hold on;
        rangex1 = [min(x(1, :)), max(x(1, :))];
        rangex2 = [min(x(2, :)), max(x(2, :))];
        [x1Grid, x2Grid, zGMM] = contourGMM(alpha, mu, Sigma, rangex1, rangex2);
        contour(x1Grid, x2Grid, zGMM);
        axis equal;
    end
end

% Computing contours for GMMs
function [x1Grid, x2Grid, zGMM] = contourGMM(alpha, mu, Sigma, rangex1, rangex2)
    x1Grid = linspace(floor(rangex1(1)), ceil(rangex1(2)), 101);
    x2Grid = linspace(floor(rangex2(1)), ceil(rangex2(2)), 91);
    [h, v] = meshgrid(x1Grid, x2Grid);
    GMM = evalGMM([h(:)';v(:)'], alpha, mu, Sigma);
    zGMM = reshape(GMM, 91, 101);
```

```
end

% Plotting the ROC curve
function plotROC(p10, p11, min_FP, min_TP)
    figure;
    plot(p10, p11, 'DisplayName', 'ROC Curve', 'LineWidth', 2);
    hold on;
    plot(min_FP, min_TP, 'o', 'DisplayName', 'Estimated Min. Error', 'LineWidth', 2);
    xlabel('Prob. False Positive');
    ylabel('Prob. True Positive');
    title('Minimum Expected Risk ROC Curve');
    legend('show');
    grid on;
    box on;
end

% Plots minimum PFE against Gamma
function plotMinPFE(logGamma, pFE, min_pFE_ind)
    figure;
    plot(logGamma, pFE, 'DisplayName', 'Errors', 'LineWidth', 2);
    hold on;
    plot(logGamma(min_pFE_ind), pFE(min_pFE_ind), 'ro', 'DisplayName', 'Minimum
Error', 'LineWidth', 2);
    xlabel('Gamma');
    ylabel('Proportion of Errors');
    title('Probability of Error vs. Gamma');
    grid on;
    legend('show');
end

% Plotting data points and classifier decisions
function plotDecisions(x, labels, decisions)
    ind00 = find(decisions == 0 & labels == 0);
    ind10 = find(decisions == 1 & labels == 0);
    ind01 = find(decisions == 0 & labels == 1);
    ind11 = find(decisions == 1 & labels == 1);
    figure;
    plot(x(1, ind00), x(2, ind00), 'og', 'DisplayName', 'Class 0, Correct');
    hold on;
    plot(x(1, ind10), x(2, ind10), 'or', 'DisplayName', 'Class 0, Incorrect');
    hold on;
    plot(x(1, ind01), x(2, ind01), '+r', 'DisplayName', 'Class 1, Correct');
    hold on;
    plot(x(1, ind11), x(2, ind11), '+g', 'DisplayName', 'Class 1, Incorrect');
    hold on;
    axis equal;
    grid on;
```

```
    title('Data and respective Classifier Decisions versus True Labels');
    xlabel('x_1');
    ylabel('x_2');
    legend('AutoUpdate', 'off');
    legend('show');
end

% Plotting contours for the Equilibrium Risk Minimization (ERM)
function plotERMContours(x, alpha0, mu0, Sigma0, alpha1, mu1, Sigma1,
logGamma_ideal)
    horizontalGrid = linspace(floor(min(x(1, :))), ceil(max(x(1, :))), 101);
    verticalGrid = linspace(floor(min(x(2, :))), ceil(max(x(2, :))), 91);
    [h, v] = meshgrid(horizontalGrid, verticalGrid);
    discriminantScoreGridValues = log(evalGMM([h(:)'; v(:)'], alpha1, mu1, Sigma1)) -
log(evalGMM([h(:)'; v(:)'], alpha0, mu0, Sigma0)) - logGamma_ideal;
    minDSGV = min(discriminantScoreGridValues);
    maxDSGV = max(discriminantScoreGridValues);
    discriminantScoreGrid = reshape(discriminantScoreGridValues, 91, 101);
    contour(horizontalGrid, verticalGrid, discriminantScoreGrid, [minDSGV * [0.9, 0.6,
0.3], 0, [0.3, 0.6, 0.9] * maxDSGV]);
    lgd=legend('Correct decisions for data from Class 0', 'Incorrect decisions for data
from Class 0', 'Incorrect decisions for data from Class 1', 'Correct decisions for data
from Class 1', 'Equilevel contours of the discriminant function');
    set(lgd, 'FontSize', 6);
end
```

QUESTION 2

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import multivariate_normal
from sklearn.metrics import confusion_matrix
from math import ceil, floor
from sklearn.preprocessing import PolynomialFeatures
import numpy as np
import pylab
from mpl_toolkits.mplot3d import Axes3D

def hw2q2():
    Ntrain = 100
    data = generateData(Ntrain)
    plot3(data[0,:],data[1,:],data[2,:])
    xTrain = data[0:2,:]
    yTrain = data[2,:]

    Ntrain = 1000
    data = generateData(Ntrain)
    plot3(data[0,:],data[1,:],data[2,:])
    print("hw2q2,data.shape",data.shape)
    xValidate = data[0:2,:]
    yValidate = data[2,:]

print("hw2q2,xValidate.shape,yValidate.shape:",xValidate.shape,yValidate.s
hape)

    return xTrain,yTrain,xValidate,yValidate

def generateData(N):
    gmmParameters = {}
    gmmParameters['priors'] = [.3,.4,.3] # priors should be a row vector
    gmmParameters['meanVectors'] = np.array([[-10, 0, 10], [0, 0, 0], [10,
0, -10]])
    gmmParameters['covMatrices'] = np.zeros((3, 3, 3))
    gmmParameters['covMatrices'][:,:,0] = np.array([[1, 0, -3], [0, 1, 0],
[-3, 0, 15]])
    gmmParameters['covMatrices'][:,:,1] = np.array([[8, 0, 0], [0, .5, 0],
[0, 0, .5]])
    gmmParameters['covMatrices'][:,:,2] = np.array([[1, 0, -3], [0, 1, 0],
[-3, 0, 15]])
    x,labels = generateDataFromGMM(N,gmmParameters)
    return x
```

```
def generateDataFromGMM(N,gmmParameters):
#    Generates N vector samples from the specified mixture of Gaussians
#    Returns samples and their component labels
#    Data dimensionality is determined by the size of mu/Sigma parameters
    priors = gmmParameters['priors'] # priors should be a row vector
    meanVectors = gmmParameters['meanVectors']
    covMatrices = gmmParameters['covMatrices']
    n = meanVectors.shape[0] # Data dimensionality
    C = len(priors) # Number of components
    x = np.zeros((n,N))
    labels = np.zeros((1,N))
    # Decide randomly which samples will come from each component
    u = np.random.random((1,N))
    thresholds = np.zeros((1,C+1))
    thresholds[:,0:C] = np.cumsum(priors)
    thresholds[:,C] = 1
    for l in range(C):
        indl = np.where(u <= float(thresholds[:,l]))
        Nl = len(indl[1])
        labels[indl] = (l+1)*1
        u[indl] = 1.1
        x[:,indl[1]] =
np.transpose(np.random.multivariate_normal(meanVectors[:,l],
covMatrices[:,:,l], Nl))

    return x,labels

def plot3(a,b,c,mark = "x", col = "b"):
    pylab.ion()
    fig = pylab.figure()
    ax = Axes3D(fig)
    ax.scatter(a, b, c, marker = mark, color = col)
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")
    ax.set_zlabel("y")
    ax.set_title('Training Dataset')




np.set_printoptions(suppress=True)
np.random.seed(45)
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 18)
plt.rc('axes', labelsize = 18)
```

```
plt.rc('xtick', labelsize = 14)
plt.rc('ytick', labelsize = 14)
plt.rc('legend', fontsize = 16)
plt.rc('figure', titlesize = 18)



# Breaking the matrix X & vector y into batches
def batch(X, y, batch_size, N):
    X_batch = []
    y_batch = []

    # Iterate over N in batch_size steps, last batch may be < batch_size
    for i in range(0, N, batch_size):
        nxt = min(i + batch_size, N + 1)
        X_batch.append(X[i:nxt, :])
        y_batch.append(y[i:nxt])

    return X_batch, y_batch


def gradient_descent(loss_func, theta0, X, y, N, *args, **kwargs):

    # Default options
    max_epoch = kwargs['max_epoch'] if 'max_epoch' in kwargs else 200
    alpha = kwargs['alpha'] if 'alpha' in kwargs else 0.1
    epsilon = kwargs['tolerance'] if 'tolerance' in kwargs else 1e-6

    batch_size = kwargs['batch_size'] if 'batch_size' in kwargs else 10

    # Turning data into batches
    X_batch, y_batch = batchify(X, y, batch_size, N)
    num_batches = len(y_batch)
    print("\n\n\n%d Batches of size %d:" % (num_batches, batch_size))
    print("\n\n\n")
    theta = theta0
    m_t = np.zeros(theta.shape)

    trace = {}
    trace['loss'] = []
    trace['theta'] = []

    # Main loop:
    for epoch in range(1, max_epoch + 1):
        loss_epoch = 0
        for b in range(num_batches):
```

```
            X_b = X_batch[b]
            y_b = y_batch[b]

            # Compute NLL loss and gradient of NLL function
            loss, gradient = loss_func(theta, X_b, y_b, *args)
            loss_epoch += loss

            # Steepest descent update
            theta = theta - alpha * gradient

            # Terminating Condition is based on how close we are to
minimum (gradient = 0)
            if np.linalg.norm(gradient) < epsilon:
                print("Gradient Descent has converged after {}
epochs".format(epoch))
                break
        trace['loss'].append(np.mean(loss_epoch))
        trace['theta'].append(theta)

        if np.linalg.norm(gradient) < epsilon:
            break

    return theta, trace


def cubic_trans(X):
    n = X.shape[1]
    phi_X = X
    phi_X = np.column_stack((phi_X, X[:, 1] * X[:, 1],              X[:,
1] * X[:, 2],              X[:, 2] * X[:, 2],              \
                                    X[:, 1] * X[:, 1] * X[:, 1],    X[:,
1] * X[:, 1] * X[:, 2],    X[:, 1] * X[:, 2] * X[:, 2],    \
                                    X[:, 2] * X[:, 2] * X[:, 2]
                                    ))

    return phi_X


def loss_linreg(theta, X, y, sigma2=1):
    B = X.shape[0]
    predictions = X.dot(theta)
    error = predictions - y
    loss_f = (1 / (2 * sigma2)) * np.sum(error ** 2)
    g = (1 / (B * sigma2)) * X.T.dot(error)
    return loss_f, g
```

```
def map_gamma(X, y, gamma, sigma2=1):
    reg_term = gamma * sigma2 * np.identity(X.shape[1])
    theta = np.linalg.inv(X.T.dot(X) + reg_term).dot(X.T.dot(y))
    return theta


def meansq_err(X,y,theta):
    y_predict = X.dot(theta) #+ noiseV
    mse = np.mean((y - y_predict)**2)
    return mse

# Options for mini-batch gradient descent
option = {}
option['max_epoch'] = 100
option['alpha'] = 1e-6
option['tolerance'] = 1e-3
option['batch_size'] = 10



def main():
    mu = np.zeros(10)
    sigma2 = 1
    sigma = np.identity(10)*sigma2
    mu = 0
    sigma = 1


    Ntrain = 100
    Nvalidate = 1000
    xTrain, yTrain, xValidate, yValidate = hw2q2()
    print("xTrain, yTrain, xValidate, yValidate",xTrain.shape,
yTrain.shape, xValidate.shape, yValidate.shape)
    xTrain, yTrain, xValidate, yValidate = xTrain.transpose(),
yTrain.transpose(), xValidate.transpose(), yValidate.transpose()
    print("xTrain, yTrain, xValidate, yValidate",xTrain.shape,
yTrain.shape, xValidate.shape, yValidate.shape)

    noiseT = multivariate_normal.rvs(mu,sigma,Ntrain)
    noiseV = multivariate_normal.rvs(mu,sigma,Nvalidate)

    xAugT = np.column_stack((np.ones(Ntrain), xTrain))
    yAug = np.column_stack((np.ones(Ntrain), yTrain))
    X3train = cubic_transformation(xAugT) #+ noiseT

    xAugV = np.column_stack((np.ones(Nvalidate), xValidate))
```

```
    X3validate = cubic_transformation(xAugV) #+ noiseT

    nCubic = X3train.shape[1]
    theta0 = np.random.randn(nCubic)

    theta_gd, trace = gradient_descent(loss_linreg, theta0, X3train,
yTrain, Ntrain, **opts)
    theta_map = map_gamma(X3train,yTrain,0)

    #Results
    print('theta start:')
    print(theta0)
    print('theta MLE:')
    print(theta_gd)
    print('theta MAP:')
    print(theta_map)
    print()

    mse_gd = meansq_err(X3validate,yValidate,theta_gd)
    mse_map = meansq_err(X3validate,yValidate,theta_map)
    print('MSE GD:', mse_gd)
    print('MSE MAP:', mse_map)

    y_map = X3validate.dot(theta_map) + noiseV
    y_gd = X3validate.dot(theta_gd) + noiseV


    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(xValidate[:, 0], xValidate[:, 1], yValidate, marker='x',
color='w', label='True Data')
    ax.scatter(X3validate[:, 1], X3validate[:, 2], y_gd, marker='x',
color='r', label='ML Estimate')
    ax.scatter(X3validate[:, 1], X3validate[:, 2], y_map, marker='x',
color='g', label='MAP Estimate')
    ax.set_xlabel(r"$x1$")
    ax.set_ylabel(r"$x2$")
    ax.set_zlabel(r"$y$")
    ax.legend()
    plt.show()

    trials = 100000
    gamma = np.linspace(0.0000000001,10000,trials)
    mse_range = []
    mse_range_ml = []
```

```
    theta_ml, trace_ml = gradient_descent(loss_linreg, theta0, X3train,
yTrain, Ntrain, **opts)

    for i in range(trials):

        theta_temp = map_gamma(X3train,yTrain,gamma[i])
        mse_range.append(mean_square_err(X3validate,yValidate,theta_temp))
        mse_range_ml.append(mean_square_err(X3validate, yValidate,
theta_ml))

    fig1 = plt.figure()
    plt.plot(gamma,mse_range,label='MAP MSE')
    plt.title("MSE vs Gamma")
    plt.xlabel('Gamma')
    plt.ylabel('MSE')
    plt.legend()
    plt.show()

    print("\n\n\n\n")
    fig2 = plt.figure()
    plt.plot(gamma,mse_range_ml,label='ML MSE')
    plt.title("MSE vs Gamma")
    plt.xlabel('Gamma')
    plt.ylabel('MSE')
    plt.legend()
    plt.show()

    print("\n\n\n\n")
    fig3 = plt.figure()
    plt.plot(gamma,mse_range_ml,label='ML MSE')
    plt.plot(gamma,mse_range,label='MAP MSE')
    plt.title("MSE vs Gamma")
    plt.xlabel('Gamma')
    plt.ylabel('MSE')
    plt.legend()
    plt.show()

    return

if __name__ == '__main__':
    main()
```

QUESTION 3

```
# Imports
import matplotlib.pyplot as plt
import numpy as np
import sympy as sym

# Positioning the sensors - each representing a sensor
k1 = (1/np.sqrt(2),1/np.sqrt(2))
k2 = (-1/np.sqrt(2),1/np.sqrt(2))
k3 = (-1/np.sqrt(2),-1/np.sqrt(2))
k4 = (1/np.sqrt(2),-1/np.sqrt(2))

# Actual location of object
actual_xy = (0.2,0.3)
sensors = [k1,k2,k3,k4]


# Setting the sigma values
sig_x=0.1
sig_y=0.1
sig_i=.01

def dist_pt(p1,p2):
    # Assuming points are tuples and calculates Euclidean Distance

    return ((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)**(0.5)

def val_measure(k):
    # Measuring distance between sensor and actual position
    value = dist_pt(k,actual_xy) + np.random.normal(scale=sig_i)
    if value > 0:
        return value
    else:
        val_measure(k)

sensor_val = dict()
sensor_val[k1] = round(val_measure(k1),5)
sensor_val[k2] = round(val_measure(k2),5)
sensor_val[k3] = round(val_measure(k3),5)
sensor_val[k4] = round(val_measure(k4),5)

print ("Data of the Sensor is :\n", sensor_val)

for j,i in enumerate(sensor_val):
    print ("Sensor k"+str(j+1)+" distance from Actual positon", sensor_val[i])
```

```
from numpy.core.numeric import ones_like
# Pre-defining the contour level for all plots
cont_lev=[]

for i in range(0,300,10):
    cont_lev.append(i)

# Creating a meshgrid
x = np.linspace(-2, 2, 1000)
y = np.linspace(-2, 2, 1000)
X, Y = np.meshgrid(x, y)


# Getting MAP objective values
def f0(x, y):
    # No sensor present
    priors = (x ** 2) / (sig_x ** 2) + (y ** 2) / (sig_y ** 2)
    return priors

def f1(x, y):
    # 1 sensor present
    one_sensor = (np.square(sensor_val[k1] - np.sqrt((k1[0] - x) ** 2 + (k1[1] - y) ** 2))) *
(1 / sig_i ** 2)
    priors = (x ** 2) / (sig_x ** 2) + (y ** 2) / (sig_y ** 2)
    return one_sensor + priors

def f2(x, y):
    # 2 sensors present
    priors = (x ** 2) / (sig_x ** 2) + (y ** 2) / (sig_y ** 2)
    one_sensor = (np.square(sensor_val[k1] - np.sqrt((k1[0] - x) ** 2 + (k1[1] - y) ** 2))) *
(1 / sig_i ** 2)
    two_sensor = (np.square(sensor_val[k2] - np.sqrt((k2[0] - x) ** 2 + (k2[1] - y) ** 2))) *
(1 / sig_i ** 2)
    return one_sensor + two_sensor + priors

def f3(x, y):
    # 3 sensors present
    priors = (x**2)/(sig_x**2)+(y**2)/(sig_y**2)
    one_sensor = (np.square(sensor_val[k1] - np.sqrt((k1[0] - x) ** 2 + (k1[1] - y) ** 2))) *
(1 / sig_i ** 2)
    two_sensor = (np.square(sensor_val[k2] - np.sqrt((k2[0] - x) ** 2 + (k2[1] - y) ** 2))) *
(1 / sig_i ** 2)
    three_sensor = (np.square(sensor_val[k3] - np.sqrt((k3[0] - x) ** 2 + (k3[1] - y) ** 2))) *
(1 / sig_i ** 2)
    return one_sensor + two_sensor + three_sensor + priors
```

```
def f4(x, y):
    # 4 sensors present
    priors = (x ** 2) / (sig_x ** 2) + (y ** 2) / (sig_y ** 2)
    one_sensor = (np.square(sensor_val[k1] - np.sqrt((k1[0] - x) ** 2 + (k1[1] - y) ** 2))) *
(1 / sig_i ** 2)
    two_sensor = (np.square(sensor_val[k2] - np.sqrt((k2[0] - x) ** 2 + (k2[1] - y) ** 2))) *
(1 / sig_i ** 2)
    three_sensor = (np.square(sensor_val[k3] - np.sqrt((k3[0] - x) ** 2 + (k3[1] - y) ** 2))) *
(1 / sig_i ** 2)
    four_sensor = (np.square(sensor_val[k4] - np.sqrt((k4[0] - x) ** 2 + (k4[1] - y) ** 2))) *
(1 / sig_i ** 2)
    return one_sensor + two_sensor + three_sensor + four_sensor + priors

def plots(f,num_sensors):
    # Plotting graphs based on number of sensors and MAP objectie values
    Z = f(X, Y)   # computing Z

    from matplotlib.pyplot import figure

    fig = figure(num = None, figsize = (15, 12), dpi = 100, facecolor = 'pink', edgecolor=
'white')

    # plot contour
    plt.contourf(X, Y, Z, 20, cmap='RdGy',levels = cont_lev);

    # set true labels and sensor positions
    plt.plot([actual_xy[0]],  [actual_xy[1]],  marker  =  'x',  markersize  =  20,  color  =
"blue",  label  =  "Actual XY",  mew = 2)
    if num_sensors  ==  1:
        plt.plot([k1[0]],  [k1[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 1")
    elif num_sensors == 2:
        plt.plot([k1[0]],  [k1[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 1")
        plt.plot([k3[0]],  [k3[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 2")
    elif num_sensors == 3:
        plt.plot([k1[0]],  [k1[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 1")
        plt.plot([k2[0]],  [k2[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 2")
        plt.plot([k3[0]],  [k3[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 3")
    elif num_sensors == 4:
        plt.plot([k1[0]],  [k1[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 1")
```

```
        plt.plot([k2[0]],  [k2[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 2")
        plt.plot([k3[0]],  [k3[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 3")
        plt.plot([k4[0]],  [k4[1]],  marker = 'o',  markersize = 10,  color = "brown", label =
"Sensor 4")

    plt.xlim(-2,2)
    plt.ylim(-2,2)
    plt.colorbar();
    plt.legend();
    return fig

Fig = plots(f0,0)

plt.title('No sensor present ', fontsize = 12)
plt.xlabel('X', fontsize = 15);
plt.ylabel('Y', fontsize = 15);
fig.text(.2, 0.025, 'Centered at origin 0,0 -> which is the prior value', fontsize = 12);

fig = plots(f1,1)

plt.title('Only 1 sensor Present ', fontsize = 12)
plt.xlabel('X', fontsize = 15)
plt.ylabel('Y', fontsize = 15);
fig.text(.2,0.025, 'Center moves closer to Actual value', fontsize = 12);

fig = plots(f2,2)

plt.title('2 sensors Present. ', fontsize = 12)
plt.xlabel('X', fontsize = 15)
plt.ylabel('Y', fontsize = 15);
fig.text(.2,0.025, 'Center moves closer to Actual value', fontsize = 12);

fig = plots(f3,3)

plt.title('3 sensors Present. ', fontsize = 12)
plt.xlabel('X', fontsize = 15)
plt.ylabel('Y', fontsize = 15);
fig.text(.2,0.025, 'Center moves very close to Actual value', fontsize = 12);

fig = plots(f4,4)

plt.title('4 sensors Present ', fontsize = 12)
plt.xlabel('X', fontsize = 15)
plt.ylabel('Y', fontsize = 15);
fig.text(.2,0.025, 'Center almost coincidences with Actual value', fontsize = 12);
```

Code help from Prof Deniz and Github.