Northeastern University
College of Engineering

October 28, 2024

EECE 5644: Introduction to Machine Learning & Pattern Recognition
Assignment 2

Name: Pradnyesh Choudhari
NUID: 002339243

GitHub Link - https://github.com/Prad06/eece-5644-northeastern

Assignment 2 - Question 1

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
from scipy.stats import multivariate_normal
from scipy.optimize import minimize
```

In [ ]:
```
np.random.seed(2)
```

Data Creation Part

Question 1 (20%)

The probability density function (pdf) for a 2-dimensional real-valued random vector $\mathbf{X}$ is as follows: $p(\mathbf{x}) = P(L=0)p(\mathbf{x}|L=0) + P(L=1)p(\mathbf{x}|L=1)$. Here L is the true class label that indicates which class-label-conditioned pdf generates the data.

The class priors are $P(L=0) = 0.6$ and $P(L=1) = 0.4$. The class class-conditional pdfs are $p(\mathbf{x}|L=0) = w_{01}g(\mathbf{x}|\mathbf{m}_{01}, \mathbf{C}_{01}) + w_{02}g(\mathbf{x}|\mathbf{m}_{02}, \mathbf{C}_{02})$ and $p(\mathbf{x}|L=1) = w_{11}g(\mathbf{x}|\mathbf{m}_{11}, \mathbf{C}_{11}) + w_{12}g(\mathbf{x}|\mathbf{m}_{12}, \mathbf{C}_{12})$, where $g(\mathbf{x}|\mathbf{m}, \mathbf{C})$ is a multivariate Gaussian probability density function with mean vector $\mathbf{m}$ and covariance matrix $\mathbf{C}$. The parameters of the class-conditional Gaussian pdfs are: $w_{i1} = w_{i2} = 1/2$ for $i \in \{1, 2\}$, and

$$\mathbf{m}_{01} = \begin{bmatrix} -0.9 \\ -1.1 \end{bmatrix} \quad \mathbf{m}_{02} = \begin{bmatrix} 0.8 \\ 0.75 \end{bmatrix} \quad \mathbf{m}_{11} = \begin{bmatrix} -1.1 \\ 0.9 \end{bmatrix} \quad \mathbf{m}_{12} = \begin{bmatrix} 0.9 \\ -0.75 \end{bmatrix} \quad \mathbf{C}_{ij} = \begin{bmatrix} 0.75 & 0 \\ 0 & 1.25 \end{bmatrix} \text{ for all } \{ij\} \text{ pairs.}$$

For numerical results requested below, generate the following independent datasets each consisting of iid samples from the specified data distribution, and in each dataset make sure to include the true class label for each sample.

- D_{train}^{20} consists of 20 samples and their labels for training;
- D_{train}^{200} consists of 200 samples and their labels for training;
- D_{train}^{2000} consists of 2000 samples and their labels for training;
- $D_{validate}^{10K}$ consists of 10000 samples and their labels for validation;

In [ ]:
```
priors = np.array([0.6, 0.4])
means = np.array([[[-0.9, -1.1], [0.8, 0.75]], [[-1.1, 0.9], [0.9, -0.75]]])
cov = np.array([[0.75, 0], [0, 1.25]])
weight = 0.5


def generate_data(n_components, priors, means, cov):
    n_components_counts = (n_components * np.array(priors)).astype(int)

    def generate_samples(label, mean_idx):
        return np.random.multivariate_normal(
            means[label][mean_idx], cov, n_components_counts[label] // 2
        )

    samples_0 = np.vstack([generate_samples(0, 0), generate_samples(0, 1)])
    samples_1 = np.vstack([generate_samples(1, 0), generate_samples(1, 1)])

    samples = np.vstack([samples_0, samples_1])
```

```
    labels = np.hstack(
        [np.zeros(n_components_counts[0]), np.ones(n_components_counts[1])]
    )

    means_out = {
        f"{label}{idx+1}": np.mean(generate_samples(label, idx), axis=0)
        for label in [0, 1]
        for idx in [0, 1]
    }

    covs_out = {
        f"{label}{idx+1}": np.cov(generate_samples(label, idx), rowvar=False)
        for label in [0, 1]
        for idx in [0, 1]
    }

    return samples, labels, means_out, covs_out
```

In [ ]:
```
D_20, L_20, M_20, C_20 = generate_data(20, priors, means, cov)
D_200, L_200, M_200, C_200 = generate_data(200, priors, means, cov)
D_2000, L_2000, M_2000, C_2000 = generate_data(2000, priors, means, cov)
D_10K_test, L_10K_test, M_10K_test, C_10K_test = generate_data(
    10000, priors, means, cov
)

means_data = {"D_20": M_20, "D_200": M_200, "D_2000": M_2000, "D_10K_test": M_10K_test}
covs_data = {"D_20": C_20, "D_200": C_200, "D_2000": C_2000, "D_10K_test": C_10K_test}
```

Part 1

Part 1: (6%) Determine the theoretically optimal classifier that achieves minimum probability of error using the knowledge of the true pdf. Specify the classifier mathematically and implement it; then apply it to all samples in $D^{10K}_{validate}$. From the decision results and true labels for this validation set, estimate and plot the ROC curve for a corresponding discriminant score for this classifier, and on the ROC curve indicate, with a special marker, the location of the min-P(error) classifier. Also report an estimate of the min-P(error) achievable, based on counts of decision-truth label pairs on $D^{10K}_{validate}$. Optional: As supplementary visualization, generate a plot of the decision boundary of this classification rule overlaid on the validation dataset. This establishes an aspirational performance level on this data for the following approximations.

Theoretical Optimal Classifier

$$P(L=0) = 0.6$$

$$P(L=1) = 0.4$$

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

Using 0-1 Loss Matrix:

$$\frac{P(X|L=0)}{P(X|L=1)} \overset{?}{=} \frac{P(L=0)}{P(L=1)}$$

Next, for the Gaussian functions:

$$\frac{0.5 \cdot g(x|m_{01}, C_{01}) + 0.5 \cdot g(x|m_{02}, C_{02})}{0.5 \cdot g(x|m_{11}, C_{11}) + 0.5 \cdot g(x|m_{12}, C_{12})} \stackrel{?}{=} \frac{0.6}{0.4}$$

Finally:

$$\frac{0.5 \cdot g(x|m_{01}, C_{01}) + 0.5 \cdot g(x|m_{02}, C_{02})}{0.5 \cdot g(x|m_{11}, C_{11}) + 0.5 \cdot g(x|m_{12}, C_{12})} \stackrel{?}{=} 1.5$$

In [ ]:
```python
def get_px_given_label(label, name, samples, means, cov, weight):
    return weight * (
        multivariate_normal.pdf(
            samples, means[name][f"{label}1"], cov[name][f"{label}1"]
        )
        + multivariate_normal.pdf(
            samples, means[name][f"{label}2"], cov[name][f"{label}2"]
        )
    )
```

In [ ]:
```python
def apply_map(name, means, cov, priors, X):
    potential_labels = [0, 1]
    pl = len(potential_labels)
    rows = X.shape[0]
    pxgivenl = np.zeros((pl, rows))

    for cls in potential_labels:
        pxgivenl[cls, :] = get_px_given_label(cls, name, X, means, cov, weight)

    px = priors @ pxgivenl
    classPosteriors = (priors[:, np.newaxis] * pxgivenl) / px

    lossMatrix_0_1 = np.ones((pl, pl)) - np.eye(pl)
    expectedRisks = lossMatrix_0_1 @ classPosteriors
    decisions = np.argmin(expectedRisks, axis=0)

    return decisions


# def apply_map(name, means, cov, priors, X):
#     pl = len(priors)
#     pxgivenl = np.array(
#         [get_px_given_label(cls, name, X, means, cov, 0.5) for cls in range(pl)]
#     )
#     px = priors @ pxgivenl
#     classPosteriors = (priors[:, np.newaxis] * pxgivenl) / px
#     expectedRisks = (np.ones((pl, pl)) - np.eye(pl)) @ classPosteriors
#     decisions = np.argmin(expectedRisks, axis=0)

#     return decisions
```

In [ ]:
```python
def predict_map_model(name, means, cov, X):
    pxgivenl = np.array(
        [get_px_given_label(cls, name, X, means, cov, weight) for cls in [0, 1]]
    )
    px = priors @ pxgivenl
    classPosteriors = (priors[:, np.newaxis] * pxgivenl) / px
```

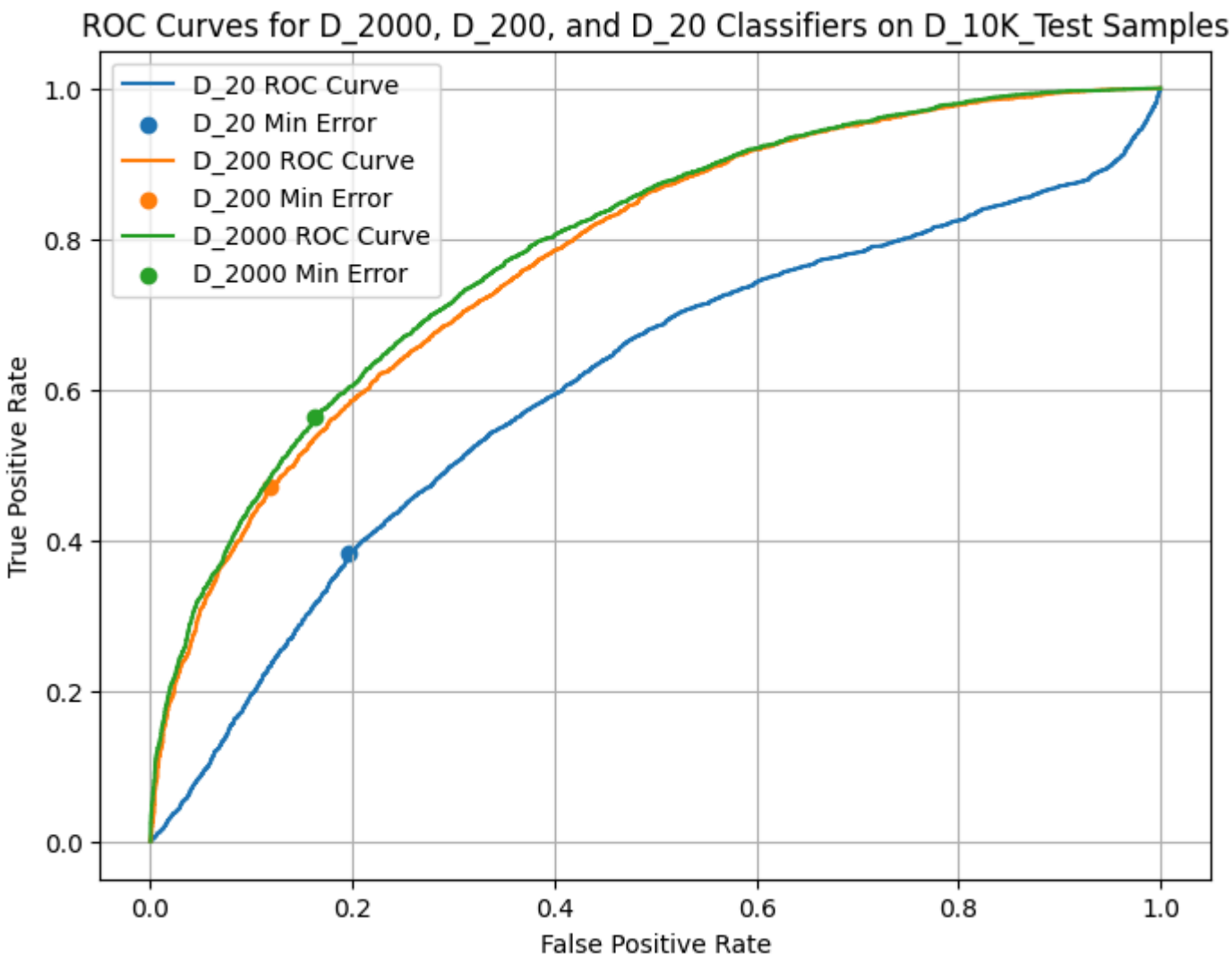
```
    lossMatrix_0_1 = np.ones((2, 2)) - np.eye(2)
    expectedRisks = lossMatrix_0_1 @ classPosteriors
    decisions = np.argmin(expectedRisks, axis=0)

    return decisions, classPosteriors[1, :]
```

In [ ]:
```
decisions_D20, decisions_D200, decisions_D2000 = [
    apply_map(name, means_data, covs_data, priors, X)
    for name, X in [("D_20", D_20), ("D_200", D_200), ("D_2000", D_2000)]
]
```

In [ ]:
```
fprs, tprs, errors, dout = [], [], [], []
for name in ["D_20", "D_200", "D_2000"]:
    decisions, posteriors = predict_map_model(name, means_data, covs_data, D_10K_test)
    fpr, tpr, thresholds = roc_curve(L_10K_test, posteriors)
    fprs.append(fpr)
    tprs.append(tpr)
    errors.append(0.4 * (1 - tpr) + 0.6 * fpr)
    dout.append(decisions)
```

In [ ]:
```
def min_error_analysis(fpr, tpr, error):
    idx_min_error = np.argmin(error)
    return fpr[idx_min_error], tpr[idx_min_error], error[idx_min_error]


min_errors = [
    min_error_analysis(fpr, tpr, error) for fpr, tpr, error in zip(fprs, tprs, errors)
]
```

In [ ]:
```
plt.figure(figsize=(8, 6))
for i, label in enumerate(["D_20", "D_200", "D_2000"]):
    plt.plot(fprs[i], tprs[i], label=f"{label} ROC Curve")
    plt.scatter(*min_errors[i][:2], label=f"{label} Min Error")

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curves for D_2000, D_200, and D_20 Classifiers on D_10K_Test Samples")
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:
```
print("PErrors on 10k Test Set based on the classifiers, min being at D_2000")
print(f"Estimated min-P(error) for D_20 classifier: {min_errors[0][2]:.4f}")
print(f"Estimated min-P(error) for D_200 classifier: {min_errors[1][2]:.4f}")
print(f"Estimated min-P(error) for D_2000 classifier: {min_errors[2][2]:.4f}")
```

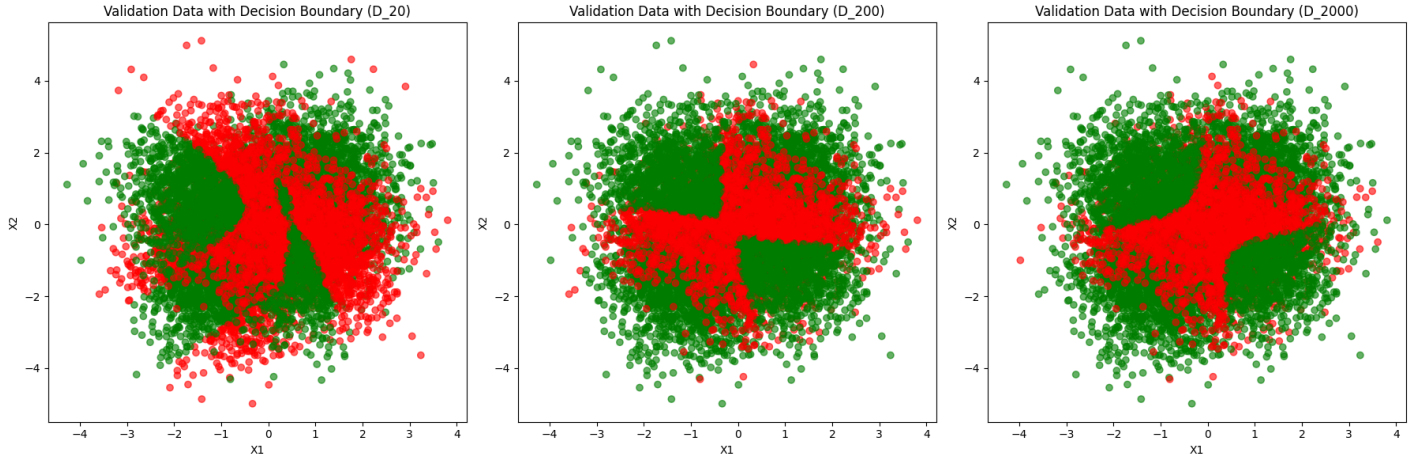
```
PErrors on 10k Test Set based on the classifiers, min being at D_2000
Estimated min-P(error) for D_20 classifier: 0.3655
Estimated min-P(error) for D_200 classifier: 0.2826
Estimated min-P(error) for D_2000 classifier: 0.2721
```

The ROC curves and minimum error points show that the classifier's performance improves as the training set size increases. The classifier trained on D_2000 achieves the lowest error rate, which is our best estimate of the min-P(error) achievable by the theoretical optimal classifier. This performance is expected as the larger dataset allows for better estimation of the true underlying distribution.

Optional Part

In [ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(18, 6))

colors_1 = np.where(dout[0] == L_10K_test, "green", "red")
axs[0].scatter(D_10K_test[:, 0], D_10K_test[:, 1], c=colors_1, alpha=0.6)
axs[0].set_title("Validation Data with Decision Boundary (D_20)")
axs[0].set_xlabel("X1")
axs[0].set_ylabel("X2")
```

```
colors_2 = np.where(dout[1] == L_10K_test, "green", "red")
axs[1].scatter(D_10K_test[:, 0], D_10K_test[:, 1], c=colors_2, alpha=0.6)
axs[1].set_title("Validation Data with Decision Boundary (D_200)")
axs[1].set_xlabel("X1")
axs[1].set_ylabel("X2")

colors_3 = np.where(dout[2] == L_10K_test, "green", "red")
axs[2].scatter(D_10K_test[:, 0], D_10K_test[:, 1], c=colors_3, alpha=0.6)
axs[2].set_title("Validation Data with Decision Boundary (D_2000)")
axs[2].set_xlabel("X1")
axs[2].set_ylabel("X2")

plt.tight_layout()
plt.show()
```



Part 2 (including optional)

Part 2: (12%) (a) Using the maximum likelihood parameter estimation technique train three separate logistic-linear-function-based approximations of class label posterior functions given a sample. For each approximation use one of the three training datasets D_{train}^{20}, D_{train}^{200}, D_{train}^{2000}. When optimizing the parameters, specify the optimization problem as minimization of the negative-log-likelihood of the training dataset, and use your favorite numerical optimization approach, such as gradient descent or Matlab's fminsearch. Determine how to use these class-label-posterior approximations to classify a sample in order to approximate the minimum-P(error) classification rule; apply these three approximations of the class label posterior function on samples in $D_{validate}^{10K}$, and estimate the probability of error that these three classification rules will attain (using counts of decisions on the validation set). Optional: As supplementary visualization, generate plots of the decision boundaries of these trained classifiers superimposed on their respective training datasets and the validation dataset. (b) Repeat the process described in Part (2a) using a logistic-quadratic-function-based approximation of class label posterior functions given a sample.

Linear Logistic Function

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

$$z(x) = [1\ X^T]^T$$

$$[1 \quad X^T]^T$$

$$P(L=1|x,w) = h_w(x) \; and \; P(L=0|x,w) = 1 - h_w(x)$$

$$P(L|x,w) = (h_w(x))^L (1-h_w(x))^{1-L}$$

given we have data with n iid samples,

$$\hat{w}_{ML} = \text{\argmax}_w P(L|x,w)$$

$$\hat{w}_{ML} = \text{\argmax}_w \prod_{i=1}^{n} P(L|x,w)$$

$$\hat{w}_{ML} = \text{\argmax}_w \prod_{i=1}^{n} (h_w(x))^{L_i} (1-h_w(x))^{1-L_i}$$

taking log on both the sides,

$$\hat{w}_{ML} = \text{\argmax}_w \prod_{i=1}^{n} log((h_w(x))^{L_i} (1-h_w(x))^{1-L_i})$$

$$\hat{w}_{ML} = \text{\argmax}_w \sum_{i=1}^{n} L_i log((h_w(x)) + (1-L_i)(1-h_w(x)))$$

converting into a minimization problem,

$$\hat{w}_{ML} = \text{\argmin}_w \frac{-1}{N} \sum_{i=1}^{n} L_i log((h_w(x)) + (1-L_i)(1-h_w(x)))$$

$$\hat{w}_{ML} = \text{\argmin}_w \frac{-1}{N} \sum_{i=1}^{N} [L_i \log(h_w(x_i)) + (1-L_i) \log(1-h_w(x_i))]$$

In [ ]:
```python
def get_confusion_matrix(labels, decisions, n):
    confusion_matrix = np.zeros((n, n), dtype=int)
    for true, pred in zip(labels, decisions):
        true = int(true)
        pred = int(pred)
        confusion_matrix[pred, true] += 1
    return confusion_matrix


def get_perror(confusion_matrix):
    total_predictions = confusion_matrix.sum()
    correct_classifications = confusion_matrix.diagonal().sum()
    misclassifications = total_predictions - correct_classifications
    perror = misclassifications / total_predictions
    return perror


def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def cost_function(w, X, n, y):
    h = sigmoid(np.dot(X, w))
```

```
    cost = (-1 / n) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    return cost


def optimize_weights(X, y, case):
    n = X.shape[0]
    if case == "L":
        X_train = np.hstack([np.ones((X.shape[0], 1)), X])
        initial_w = np.zeros(X_train.shape[1])
    elif case == "Q":
        X_train = np.hstack([
            np.ones((X.shape[0], 1)),
            X[:, 0].reshape(-1, 1),
            X[:, 1].reshape(-1, 1),
            (X[:, 0]**2).reshape(-1, 1),
            (X[:, 0] * X[:, 1]).reshape(-1, 1),
            (X[:, 1]**2).reshape(-1, 1)
        ])
        initial_w = np.zeros(X_train.shape[1])
        initial_w = np.zeros(X_train.shape[1])
    result = minimize(cost_function, initial_w, args=(X_train, n, y), method="Nelder-Mea
    return result.x


def predict(new_data, optimized_weights, case):
    if case == "L":
        new_data_with_bias = np.hstack([np.ones((new_data.shape[0], 1)), new_data])
    if case == "Q":
        new_data_with_bias = np.hstack([
            np.ones((new_data.shape[0], 1)),
            new_data[:, 0].reshape(-1, 1),
            new_data[:, 1].reshape(-1, 1),
            (new_data[:, 0]**2).reshape(-1, 1),
            (new_data[:, 0] * new_data[:, 1]).reshape(-1, 1),
            (new_data[:, 1]**2).reshape(-1, 1)
        ])
    probabilities = sigmoid(np.dot(new_data_with_bias, optimized_weights))
    predictions = (probabilities >= 0.5).astype(int)
    return predictions, probabilities


result_20_L = optimize_weights(D_20, L_20, "L")
result_200_L = optimize_weights(D_200, L_200, "L")
result_2000_L = optimize_weights(D_2000, L_2000, "L")
result_20_Q = optimize_weights(D_20, L_20, "Q")
result_200_Q = optimize_weights(D_200, L_200, "Q")
result_2000_Q = optimize_weights(D_2000, L_2000, "Q")

decisions_D20_L, prob_D20_L = predict(D_10K_test, result_20_L, "L")
decisions_D200_L, prob_D200_L = predict(D_10K_test, result_200_L, "L")
decisions_D2000_L, prob_D2000_L = predict(D_10K_test, result_2000_L, "L")
decisions_D20_Q, prob_D20_Q = predict(D_10K_test, result_20_Q, "Q")
decisions_D200_Q, prob_D200_Q = predict(D_10K_test, result_200_Q, "Q")
decisions_D2000_Q, prob_D2000_Q = predict(D_10K_test, result_2000_Q, "Q")

mat_20_L = get_confusion_matrix(decisions_D20_L, L_10K_test, 2)
mat_200_L = get_confusion_matrix(decisions_D200_L, L_10K_test, 2)
mat_2000_L = get_confusion_matrix(decisions_D2000_L, L_10K_test, 2)
mat_20_Q = get_confusion_matrix(decisions_D20_Q, L_10K_test, 2)
```

```
mat_200_Q = get_confusion_matrix(decisions_D200_Q, L_10K_test, 2)
mat_2000_Q = get_confusion_matrix(decisions_D2000_Q, L_10K_test, 2)

p_20_L = get_perror(mat_20_L)
p_200_L = get_perror(mat_200_L)
p_2000_L = get_perror(mat_2000_L)
p_20_Q = get_perror(mat_20_Q)
p_200_Q = get_perror(mat_200_Q)
p_2000_Q = get_perror(mat_2000_Q)
```

In [ ]:
```
print(f"The bias and weights for the D_20: {result_20_L}")
print(f"The bias and weights for the D_200: {result_200_L}")
print(f"The bias and weights for the D_2000: {result_2000_L}")
print(f"The bias and weights for the D_20, Quadratic case: {result_20_Q}")
print(f"The bias and weights for the D_200, Quadratic case: {result_200_Q}")
print(f"The bias and weights for the D_2000, Quadratic case: {result_2000_Q}")
```

```
The bias and weights for the D_20: [-0.37342715  0.12084368  0.05388029]
The bias and weights for the D_200: [-0.42170258 -0.11907731  0.12325838]
The bias and weights for the D_2000: [-0.40971034 -0.02950639  0.12412103]
The bias and weights for the D_20, Quadratic case: [-0.48996073 -0.4110244  -0.39225219
0.41993891 -1.54556269 -0.42944944]
The bias and weights for the D_200, Quadratic case: [-0.52909416 -0.17118677 -0.01874403
-0.01207612 -0.78185205 -0.00981614]
The bias and weights for the D_2000, Quadratic case: [-0.52337666  0.01366269  0.0245992
0.12639664 -0.88698787 -0.05012994]
```

In [ ]:
```
print("Confusion Matrix D_20 Linear")
mat_20_L
```

```
Confusion Matrix D_20 Linear
```

Out[ ]:
```
array([[5924,   76],
       [3985,   15]])
```

In [ ]:
```
print("Confusion Matrix D_200 Linear")
mat_200_L
```

```
Confusion Matrix D_200 Linear
```

Out[ ]:
```
array([[5978,   22],
       [3702,  298]])
```

In [ ]:
```
print("Confusion Matrix D_2000 Linear")
mat_2000_L
```

```
Confusion Matrix D_2000 Linear
```

Out[ ]:
```
array([[5976,   24],
       [3942,   58]])
```

In [ ]:
```
print("Confusion Matrix D_20 Quadratic")
mat_20_Q
```

```
Confusion Matrix D_20 Quadratic
```

Out[ ]:
```
array([[4897, 1103],
       [1761, 2239]])
```

In [ ]:
```
print("Confusion Matrix D_200 Quadratic")
mat_200_Q
```

```
Confusion Matrix D_200 Quadratic
```

Out[ ]:
```
array([[5426,  574],
       [2270, 1730]])
```

In [ ]:
```
print("Confusion Matrix D_2000 Quadratic")
mat_2000_Q
```

```
Confusion Matrix D_2000 Quadratic
```

Out[ ]:
```
array([[5250,  750],
       [2023, 1977]])
```

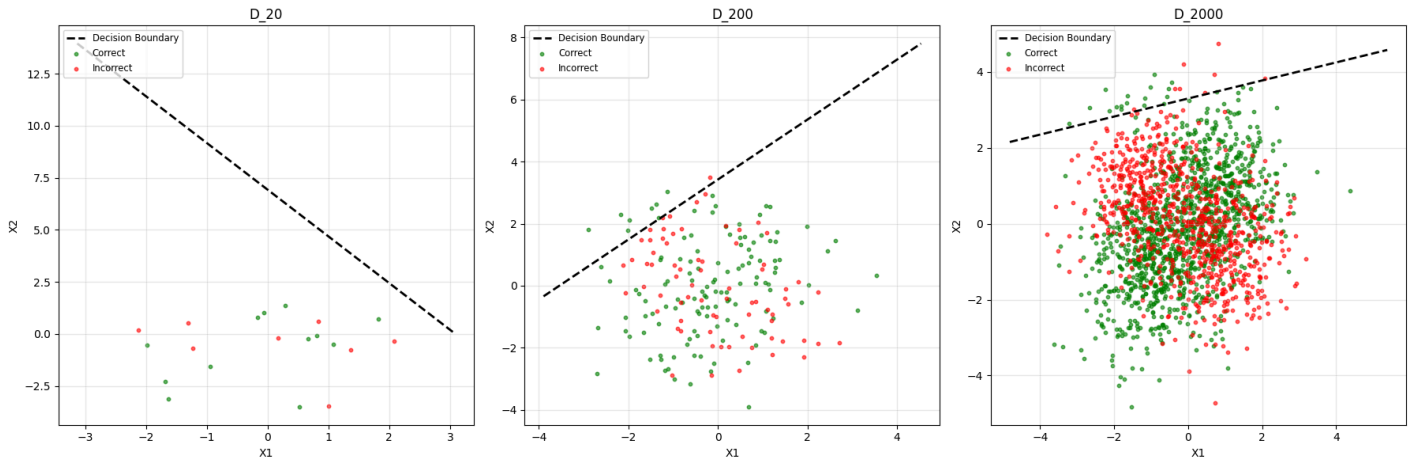
In [ ]:
```
print(f"The perror for the D_20: {p_20_L}")
print(f"The perror for the D_200: {p_200_L}")
print(f"The perror for the D_2000: {p_2000_L}")
print(f"The perror for the D_20, Quadratic case:: {p_20_Q}")
print(f"The perror for the D_200, Quadratic case:: {p_200_Q}")
print(f"The perror for the D_2000, Quadratic case:: {p_2000_Q}")
```

```
The perror for the D_20: 0.4061
The perror for the D_200: 0.3724
The perror for the D_2000: 0.3966
The perror for the D_20, Quadratic case:: 0.2864
The perror for the D_200, Quadratic case:: 0.2844
The perror for the D_2000, Quadratic case:: 0.2773
```

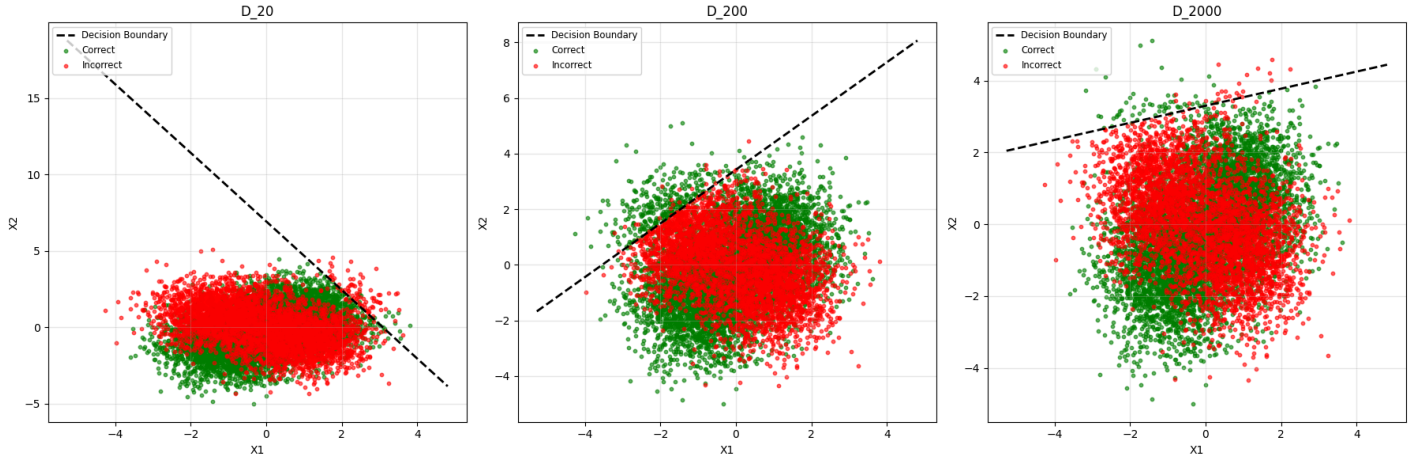
In [ ]:
```
def plot_decision_boundaries(datasets, labels, weights_list, titles, perrors, case):
    fig, axs = plt.subplots(1, 3, figsize=(18, 6))
    axs = axs.ravel()

    for i, (X, y, weights, title) in enumerate(zip(datasets, labels, weights_list, title
        ax = axs[i]

        if case == "L":
            w0, w1, w2 = weights
            z = w0 + w1 * X[:, 0] + w2 * X[:, 1]

        else:  # case == "Q"
            w0, w1, w2, w3, w4, w5 = weights
            z = (w0 + w1 * X[:, 0] + w2 * X[:, 1] +
                 w3 * X[:, 0]**2 + w4 * X[:, 1]**2 + w5 * X[:, 0] * X[:, 1])

        x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        x1_grid = np.linspace(x1_min, x1_max, 1000)

        if case == "L":
            X2_boundary = -(w0 + w1 * x1_grid) / w2
            ax.plot(x1_grid, X2_boundary, "k--", label="Decision Boundary", linewidth=2)

        else:
            X2_boundary_positive = (- (w5 * x1_grid) +
                                    np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1

            X2_boundary_negative = (- (w5 * x1_grid) -
                                    np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1

            ax.plot(x1_grid, X2_boundary_positive, "k--", label="Decision Boundary Posit
            ax.plot(x1_grid, X2_boundary_negative, "k--", label="Decision Boundary Negat


        y_pred = (z >= 0).astype(int)
        correct_mask = y_pred == y
```

```
        incorrect_mask = y_pred != y

        ax.scatter(
            X[correct_mask, 0],
            X[correct_mask, 1],
            color="green",
            label="Correct",
            alpha=0.6,
            s=10
        )

        ax.scatter(
            X[incorrect_mask, 0],
            X[incorrect_mask, 1],
            color="red",
            label="Incorrect",
            alpha=0.6,
            s=10
        )

        ax.set_title(f"{title}")

        ax.set_xlabel("X1")
        ax.set_ylabel("X2")
        ax.legend(loc="upper left", fontsize="small")
        ax.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

In [ ]:
```
datasets = [D_20, D_200, D_2000]
labels = [L_20, L_200, L_2000]
weights_list = [result_20_L, result_200_L, result_2000_L]
titles = ["D_20", "D_200", "D_2000"]
perrors = [p_20_L, p_200_L, p_2000_L]
plot_decision_boundaries(datasets, labels, weights_list, titles, perrors, "L")
```

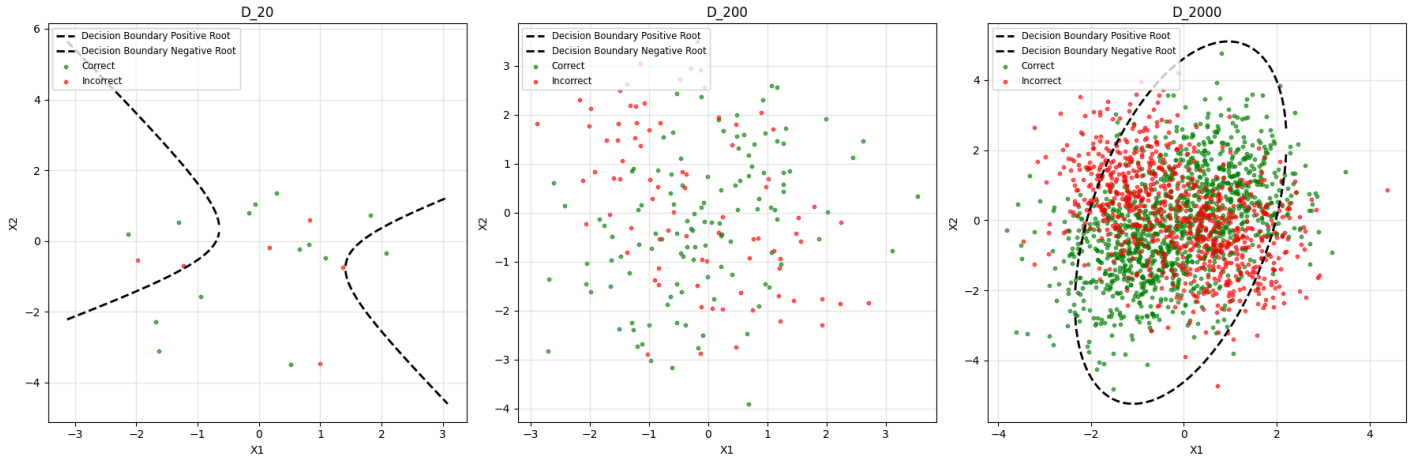


In [ ]:
```
datasets = [D_10K_test] * 3
labels = [L_10K_test] * 3
weights_list = [result_20_L, result_200_L, result_2000_L]
titles = ["D_20", "D_200", "D_2000"]
perrors = [p_20_L, p_200_L, p_2000_L]
plot_decision_boundaries(datasets, labels, weights_list, titles, perrors, "L")
```



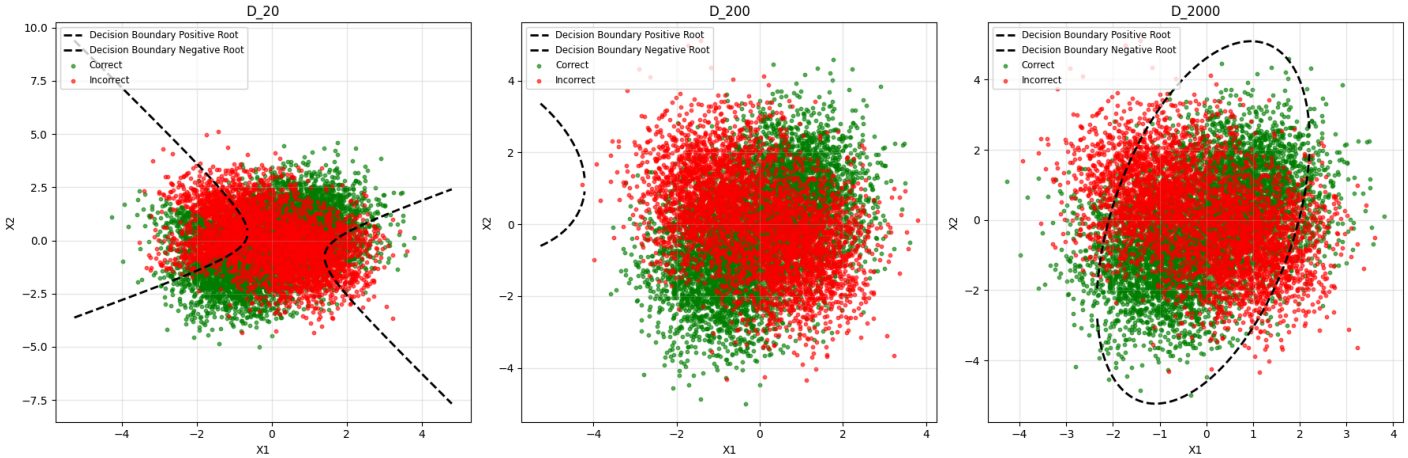
In [ ]:
```
datasets = [D_20, D_200, D_2000]
labels = [L_20, L_200, L_2000]
weights_list = [result_20_Q, result_200_Q, result_2000_Q]
titles = ["D_20", "D_200", "D_2000"]
perrors = [p_20_Q, p_200_Q, p_2000_Q]
plot_decision_boundaries(datasets, labels, weights_list, titles, perrors, "Q")
```

```
/var/folders/hh/njmxn3hn0qgb96zkxs4bc_pr0000gn/T/ipykernel_92674/268178775.py:26: Runtime
Warning: invalid value encountered in sqrt
  np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1_grid + w3 * x1_grid ** 2))) / (2 *
w2)
/var/folders/hh/njmxn3hn0qgb96zkxs4bc_pr0000gn/T/ipykernel_92674/268178775.py:29: Runtime
Warning: invalid value encountered in sqrt
  np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1_grid + w3 * x1_grid ** 2))) / (2 *
w2)
```



In [ ]:
```
datasets = [D_10K_test] * 3
labels = [L_10K_test] * 3
weights_list = [result_20_Q, result_200_Q, result_2000_Q]
titles = ["D_20", "D_200", "D_2000"]
perrors = [p_20_Q, p_200_Q, p_2000_Q]
plot_decision_boundaries(datasets, labels, weights_list, titles, perrors, "Q")
```

```
/var/folders/hh/njmxn3hn0qgb96zkxs4bc_pr0000gn/T/ipykernel_92674/268178775.py:26: Runtime
Warning: invalid value encountered in sqrt
  np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1_grid + w3 * x1_grid ** 2))) / (2 *
w2)
/var/folders/hh/njmxn3hn0qgb96zkxs4bc_pr0000gn/T/ipykernel_92674/268178775.py:29: Runtime
Warning: invalid value encountered in sqrt
  np.sqrt((w5 * x1_grid) ** 2 - 4 * w2 * (w0 + w1 * x1_grid + w3 * x1_grid ** 2))) / (2 *
w2)
```



Part 3 - Discussion

Discussion: (2%) How does the performance of your classifiers trained in this part compare to each other considering differences in number of training samples and function form? How do they compare to the theoretically optimal classifier from Part 1? Briefly discuss results and insights.

The theoretical optimal classifier, showed improving performance as the training set size increased. The estimated minimum probability of error (min-P(error)) for each dataset was:

- D_20 classifier: 0.3655
- D_200 classifier: 0.2826
- D_2000 classifier: 0.2721

The linear logistic regression model showed varying performance across different dataset sizes:

- D_20: P(error) = 0.4061
- D_200: P(error) = 0.3724
- D_2000: P(error) = 0.3966

The quadratic logistic regression model demonstrated superior performance compared to the linear model:

- D_20: P(error) = 0.2864
- D_200: P(error) = 0.2844
- D_2000: P(error) = 0.2773

Comparison of Models

The quadratic logistic regression model consistently outperformed the linear model across all dataset sizes. The decision boundary is non-linear. For the linear model, increasing the dataset size didn't always lead to better performance. However, the quadratic model showed an improvement as the dataset size increased. The quadratic logistic regression model achieved performance close to the theoretical optimal classifier, especially for the larger datasets.

Assignment 2 - Question 2

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pylab
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import minimize
```

Question 2 (20%)

Assume that scalar-real y and two-dimensional real vector $\mathbf{x}$ are related to each other according to $y = c(\mathbf{x}, \mathbf{w}) + v$, where $c(., \mathbf{w})$ is a cubic polynomial in $\mathbf{x}$ with coefficients $\mathbf{w}$ and v is a random Gaussian random scalar with mean zero and σ^2-variance.

Given a dataset $D = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ with N samples of $(\mathbf{x}, y)$ pairs, with the assumption that these samples are independent and identically distributed according to the model, derive two estimators for $\mathbf{w}$ using maximum-likelihood (ML) and maximum-a-posteriori (MAP) parameter estimation approaches as a function of these data samples. For the MAP estimator, assume that $\mathbf{w}$ has a zero-mean Gaussian prior with covariance matrix $\gamma\mathbf{I}$.

Having derived the estimator expressions, implement them in code and apply to the dataset generated by the attached Matlab script. Using the *training dataset*, obtain the ML estimator and the MAP estimator for a variety of γ values ranging from 10^{-m} to 10^{n}. Evaluate each *trained* model by calculating the average-squared error between the y values in the *validation samples* and model estimates of these using $c(., \mathbf{w}_{trained})$. How does your MAP-trained model perform on the validation set as γ is varied? How is the MAP estimate related to the ML estimate? Describe your experiments, visualize and quantify your analyses (e.g. average squared error on validation dataset as a function of hyperparameter γ) with data from these experiments.

Note: Point split will be 20% for ML and 20% for MAP estimator results and discussion.

In [ ]:
```
np.random.seed(7)
```

In [ ]:
```
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,:])
    xValidate = data[0:2,:]
    yValidate = data[2,:]

    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], covM

    return x,labels

def plot3(a,b,c,mark="o",col="b"):
  fig = plt.figure(figsize=(10, 8))
  ax = fig.add_subplot(111, projection='3d')
  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')
```

MLE & MAP

Q2. Maximum Likelihood classifier

$$y = c(x, w) + v$$

$$v \sim N(0, \sigma^2)$$

$$p(y_i | x_i, w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y_i - c(x_i, w))^2}{2\sigma^2}}$$

taking log

$$\log p(y_i | x_i, w) = \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y_i - c(x_i, w))^2}{2\sigma^2}} \right)$$

$$= \frac{-1}{2} \log(2\pi\sigma^2) - \frac{(y_i - c(x_i, w))^2}{2\sigma^2}$$

($\log(2\pi\sigma^2)$: constant wrt to w)

$$= \frac{-1}{2\sigma^2} (y_i - c(x_i, w))^2$$

$$p(D | w) = \prod_{i=1}^{N} p(y_i | x_i, w)$$

$$\log p(D | w) = \sum_{i=1}^{N} \log p(y_i | x_i, w)$$

$$\text{Maximum Likelihood} = \underset{w}{\text{argmax}} \sum_{i=1}^{N} \log p(y_i | x_i, w)$$

$$= \underset{w}{\text{argmax}} \sum_{i=1}^{N} \frac{-1}{2\sigma^2} (y_i - c(x_i, w))^2$$

turning into min

$$= \underset{w}{\text{argmin}} \sum_{i=1}^{N} \frac{1}{N} (y_i - c(x_i, w))^2$$

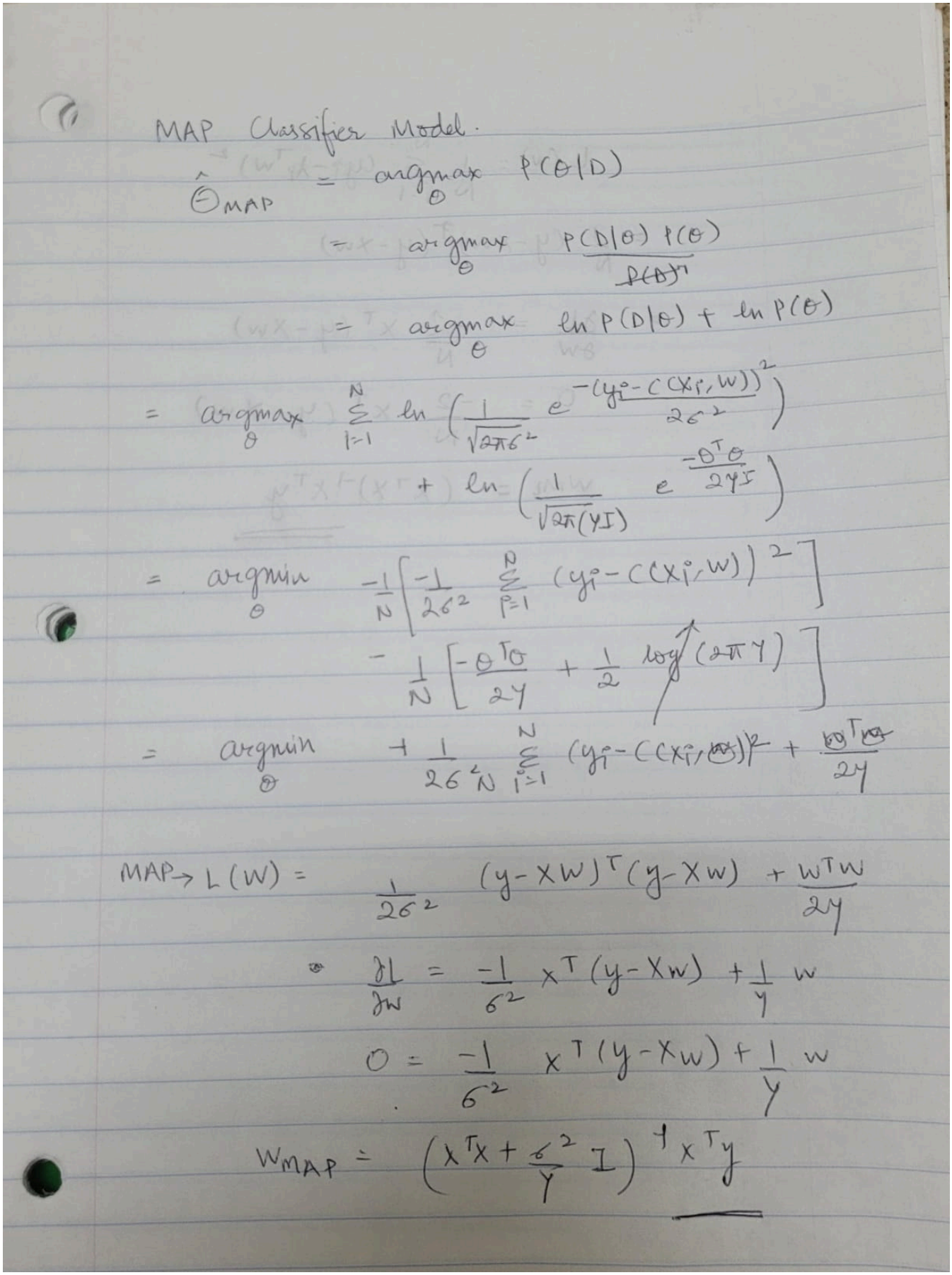
$$L(w) = \frac{1}{N} \sum_{i=1}^{N} (y_i - x_i^T w)^2$$

$$= \frac{1}{N} (y - Xw)^T (y - Xw)$$

$$\frac{\partial L}{\partial w} = \frac{-2}{N} X^T (y - Xw)$$

$$0 = \frac{-2}{N} X^T (y - Xw)$$

$$w_{ML} = (X^T X)^{-1} X^T y$$



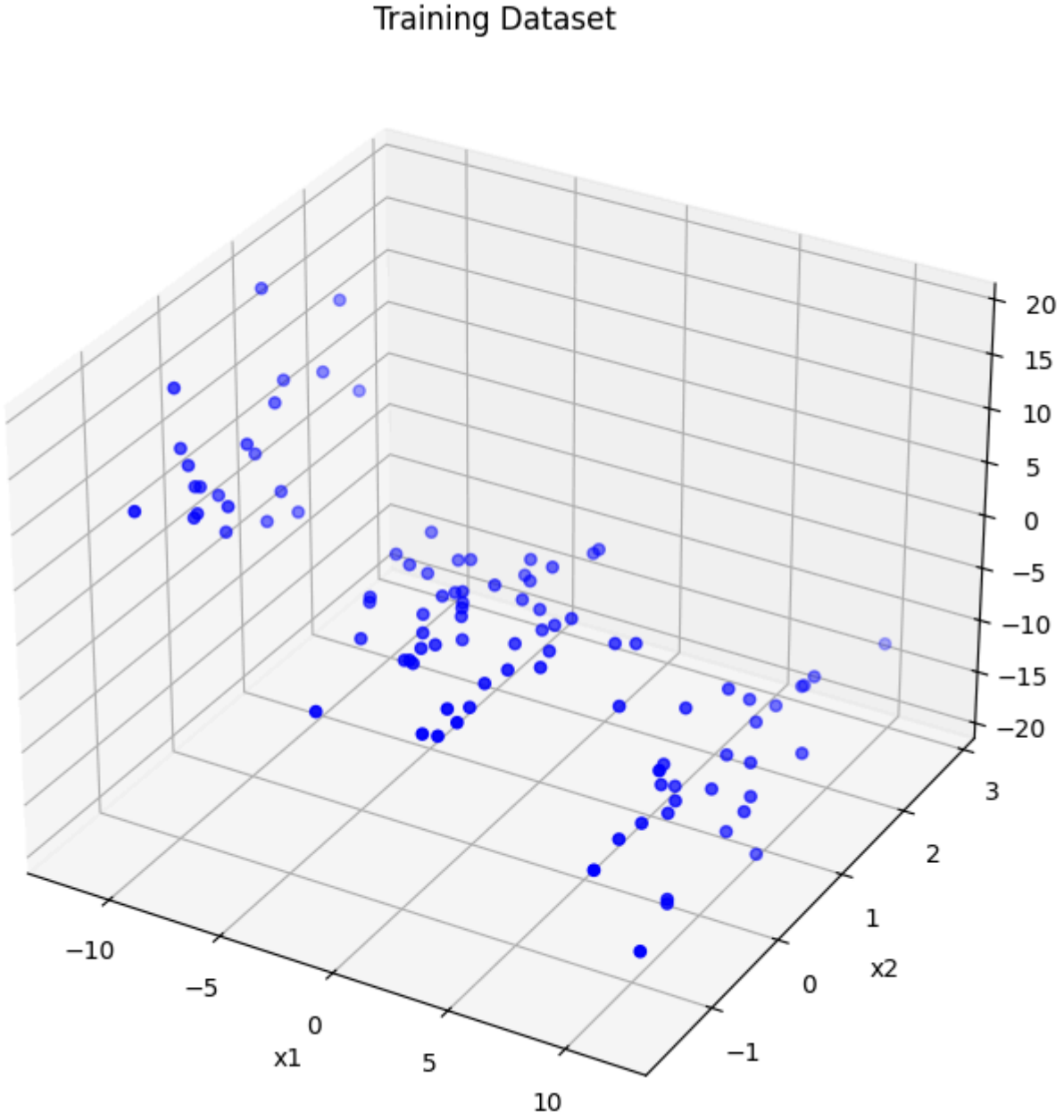
In [ ]:
```
def c(x, w):
    x1, x2 = x[0], x[1]
    return w[0] + w[1]*x1 + w[2]*x1**2 + w[3]*x1**3 + w[4]*x2 + w[5]*x2**2 + w[6]*x2**3

def likelihood(w, x_data, y_data, sigma):
```

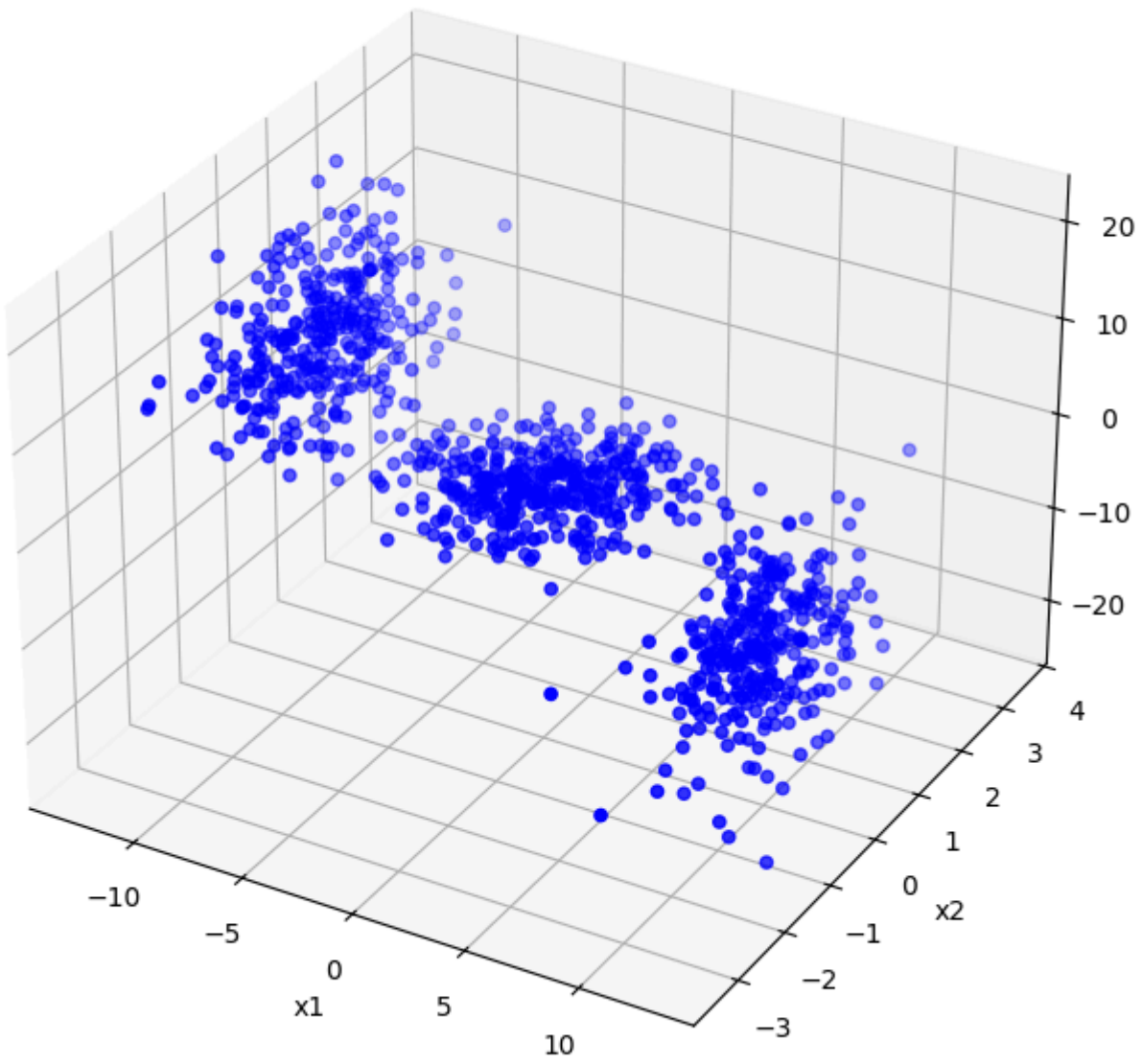
```
    predictions = np.array([c(x, w) for x in x_data.T])
    return 0.5 * np.sum(((y_data - predictions)**2) / sigma**2)

def prior(w, gamma):
    return 0.5 * np.sum(w**2) / gamma

def loss(w, x_data, y_data, sigma, gamma):
    if gamma == np.inf:  # ML case
        return likelihood(w, x_data, y_data, sigma)
    else:  # MAP case
        return likelihood(w, x_data, y_data, sigma) + prior(w, gamma)

def optimize_weights(x_data, y_data, initial_w, sigma, gamma, method):
    result = minimize(loss, initial_w, args=(x_data, y_data, sigma, gamma),
                      method=method)
    return result.x
```

In [ ]:

```
xTrain,yTrain,xValidate,yValidate = hw2q2()
initial_w = np.zeros(7)
sigma = 1.0
gamma_values = np.logspace(-6, 6, 25)

w_ml = optimize_weights(xTrain, yTrain, initial_w, sigma, np.inf, "Nelder-Mead")
ml_mse = np.mean((yValidate - np.array([c(x, w_ml) for x in xValidate.T]))**2)

w_map_values = []
mse_values = []

for gamma in gamma_values:
    w_map = optimize_weights(xTrain, yTrain, initial_w, sigma, gamma, "Nelder-Mead")
    w_map_values.append(w_map)

    predictions = np.array([c(x, w_map) for x in xValidate.T])
    mse = np.mean((yValidate - predictions)**2)
    mse_values.append(mse)
```

```
/var/folders/hh/njmxn3hn0qgb96zkxs4bc_pr0000gn/T/ipykernel_5910/2714086052.py:44: Deprecationwarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
  indl = np.where(u <= float(thresholds[:,l]))
```


Training Dataset
20
15
10
5
0
−5
−10
−15
−20
3
2
1
0
−1
x2
−10
−5
0
5
10
x1



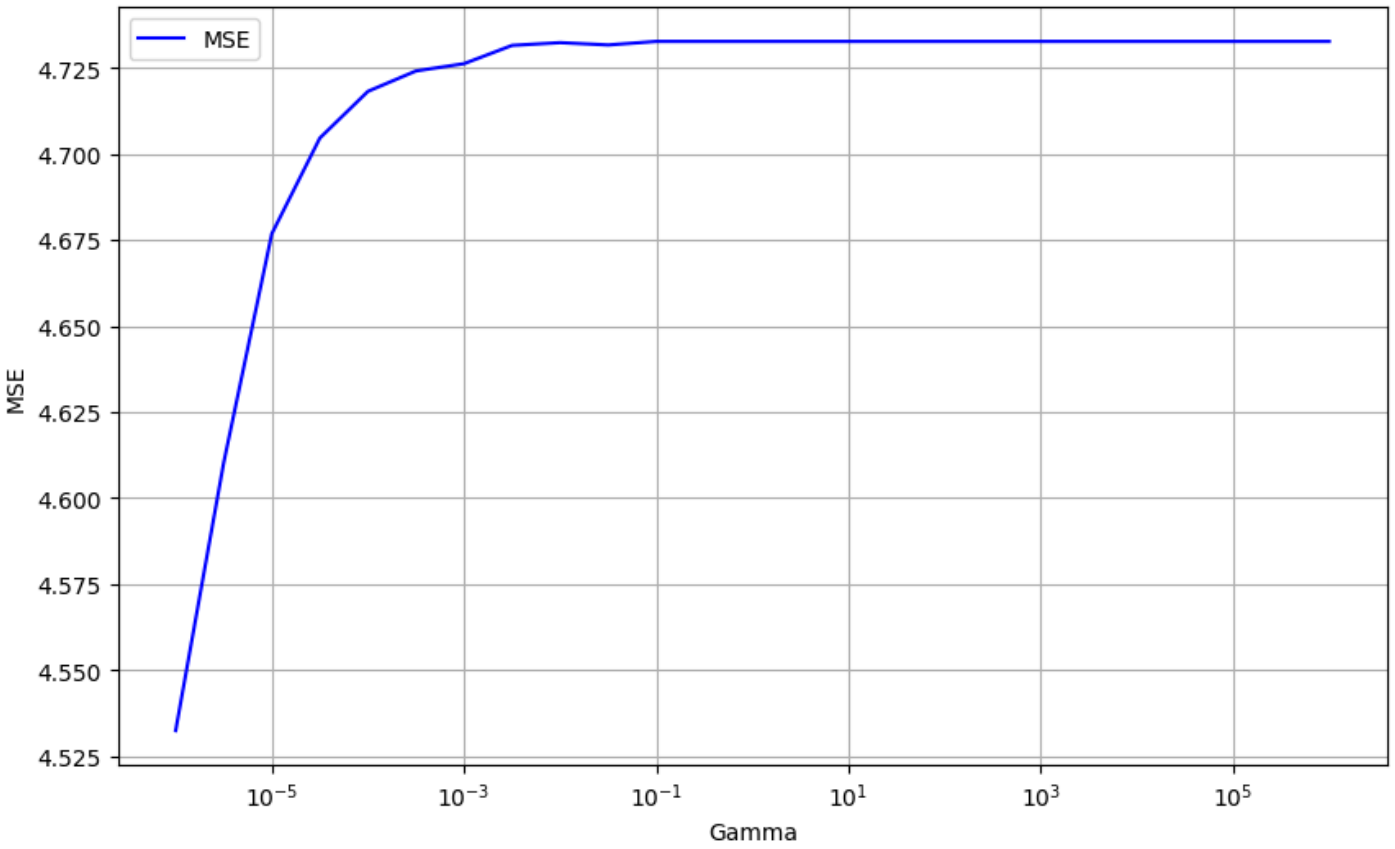
In [ ]:
```
print("ML Estimator MSE:", ml_mse)
print("Best weights:", w_ml)

print("\nBest MAP Result:")
best_idx = np.argmin(mse_values)
print(f"Best gamma: {gamma_values[best_idx]}")
print(f"Best MSE: {mse_values[best_idx]}")
print("Best weights:", w_map_values[best_idx])
```

```
ML Estimator MSE: 4.732640959794644
Best weights: [-0.00808432 -0.00364884 -0.0053063  -0.01016564  0.00572839  0.0142112
  0.00217902]

Best MAP Result:
Best gamma: 1e-06
Best MSE: 4.532484113247529
Best weights: [-1.10025155e-04  3.67557607e-04 -2.10456603e-03 -1.00414793e-02
  3.69224912e-05  7.22961845e-06  8.74016703e-05]
```

In [ ]:
```
print(f"MSE at {gamma_values[0]}, 10^-6: {mse_values[0]}\nMSE at {gamma_values[-1]}, 10^
```

```
MSE at 1e-06, 10^-6: 4.532484113247529
MSE at 1000000.0, 10^+6: 4.732640959794644
```

In [ ]:
```python
# Plot results
plt.figure(figsize=(10, 6))
plt.semilogx(gamma_values, mse_values, 'b-', label='MSE')
plt.grid(True)
plt.xlabel('Gamma')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



Using prior information in MAP estimation can enhance model compared to ML alone, when you fine-tune the regularization parameter.

- ML Estimator: This method gave us a Mean Squared Error (MSE) of about 4.73.
- MAP Estimator: By adding some prior knowledge, the MAP estimator achieved a slightly better MSE of about 4.53 when the gamma was set to a very small value (10^{-6}). This indicates that a little regularization can help improve the model.
- Regularization Effects: Gamma values were ranging from 10^{-6} to 10^{6}. The results showed that the best model was at the lowest gamma value.

Assignment 2 - Question 3

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import minimize
```

Question 3 (20%)

A vehicle at true position $[x_T, y_T]^{\mathrm{T}}$ in 2-dimensional space is to be localized using distance (range) measurements to K reference (landmark) coordinates $\{[x_1, y_1]^{\mathrm{T}}, \ldots, [x_i, y_i]^{\mathrm{T}}, \ldots, [x_K, y_K]^{\mathrm{T}}\}$. These range measurements are $r_i = d_{Ti} + n_i$ for $i \in \{1, \ldots, K\}$, where $d_{Ti} = \|[x_T, y_T]^{\mathrm{T}} - [x_i, y_i]^{\mathrm{T}}\|$ is the true distance between the vehicle and the i^{th} reference point, and n_i is a zero mean Gaussian distributed measurement noise with known variance σ_i^2. The noise in each measurement is independent from the others.

Assume that we have the following prior knowledge regarding the position of the vehicle:

$$p\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = (2\pi\sigma_x\sigma_y)^{-1} e^{-\frac{1}{2}\begin{bmatrix} x & y \end{bmatrix}\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1}\begin{bmatrix} x \\ y \end{bmatrix}} \tag{1}$$

where $[x, y]^{\mathrm{T}}$ indicates a candidate position under consideration.

Express the optimization problem that needs to be solved to determine the MAP estimate of the vehicle position. Simplify the objective function so that the exponentials and additive/multiplicative terms that do not impact the determination of the MAP estimate $[x_{MAP}, y_{MAP}]^{\mathrm{T}}$ are removed appropriately from the objective function for computational savings when evaluating the objective.

Implement the following as computer code: Set the true vehicle location to be inside the circle with unit radious centered at the origin. For each $K \in \{1, 2, 3, 4\}$ repeat the following.

Place evenly spaced K landmarks on a circle with unit radius centered at the origin. Set measurement noise standard deviation to 0.3 for all range measurements. Generate K range measurements according to the model specified above (if a range measurement turns out to be negative, reject it and resample; all range measurements need to be nonnegative).

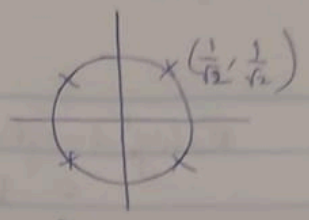
Plot the equilevel contours of the MAP estimation objective for the range of horizontal and vertical coordinates from -2 to 2; superimpose the true location of the vehicle on these equilevel contours (e.g. use a $+$ mark), as well as the landmark locations (e.g. use a o mark for each one).

Provide plots of the MAP objective function contours for each value of K. When preparing your final contour plots for different K values, make sure to plot contours at the same function value across each of the different contour plots for easy visual comparison of the MAP objective landscapes. *Suggestion:* For values of σ_x and σ_y, you could use values around 0.25 and perhaps make them equal to each other. Note that your choice of these indicates how confident the prior is about the origin as the location.

Supplement your plots with a brief description of how your code works. Comment on the behavior of the MAP estimate of position (visually assessed from the contour plots; roughly center of the innermost contour) relative to the true position. Does the MAP estimate get closer to the true position as K increases? Doe is get more certain? Explain how your contours justify your conclusions.

Note: The additive Gaussian distributed noise used in this question is likely not appropriate for a proper distance sensor, since it could lead to negative measurements. However, in this question, we will ignore this issue and proceed with this noise model for illustration. In practice, a multiplicative log-normal distributed noise may be more appropriate than an additive normal distributed noise depending on the measurement mechanism.

Q3.



Range $= d_{T_i} + n_i$

n_i : iid $N(0, \sigma^2)$

$$P(\theta) = (2\pi\sigma_x\sigma_y)^{-1} e^{-1/2 [x\ y] \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}}$$

$$\hat{\theta}_{MAP} = P(D|\theta)\, P(\theta)$$
$$= P(R|\theta)\, P(\theta)$$
$$= \log P(R|\theta)\, P(\theta)$$
$$= \log \prod_{i=1}^{k} P(R_i|\theta)\, P(\theta)$$
$$= \sum_{i=1}^{k} \log P(R_i|\theta) + \log P(\theta)$$

$$= \sum_{i=1}^{k} \log P(R_i|\theta) + \log \left[(2\pi\sigma_x\sigma_y)^{-1} e^{-1/2 [x\ y] \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}} \right]$$

$$= \sum_{i=1}^{k} \log P(R_i|\theta) + \log \frac{1}{2\pi\sigma_x\sigma_y} - \frac{1}{2} \left[[x\ y] \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right]$$

independent of x, y

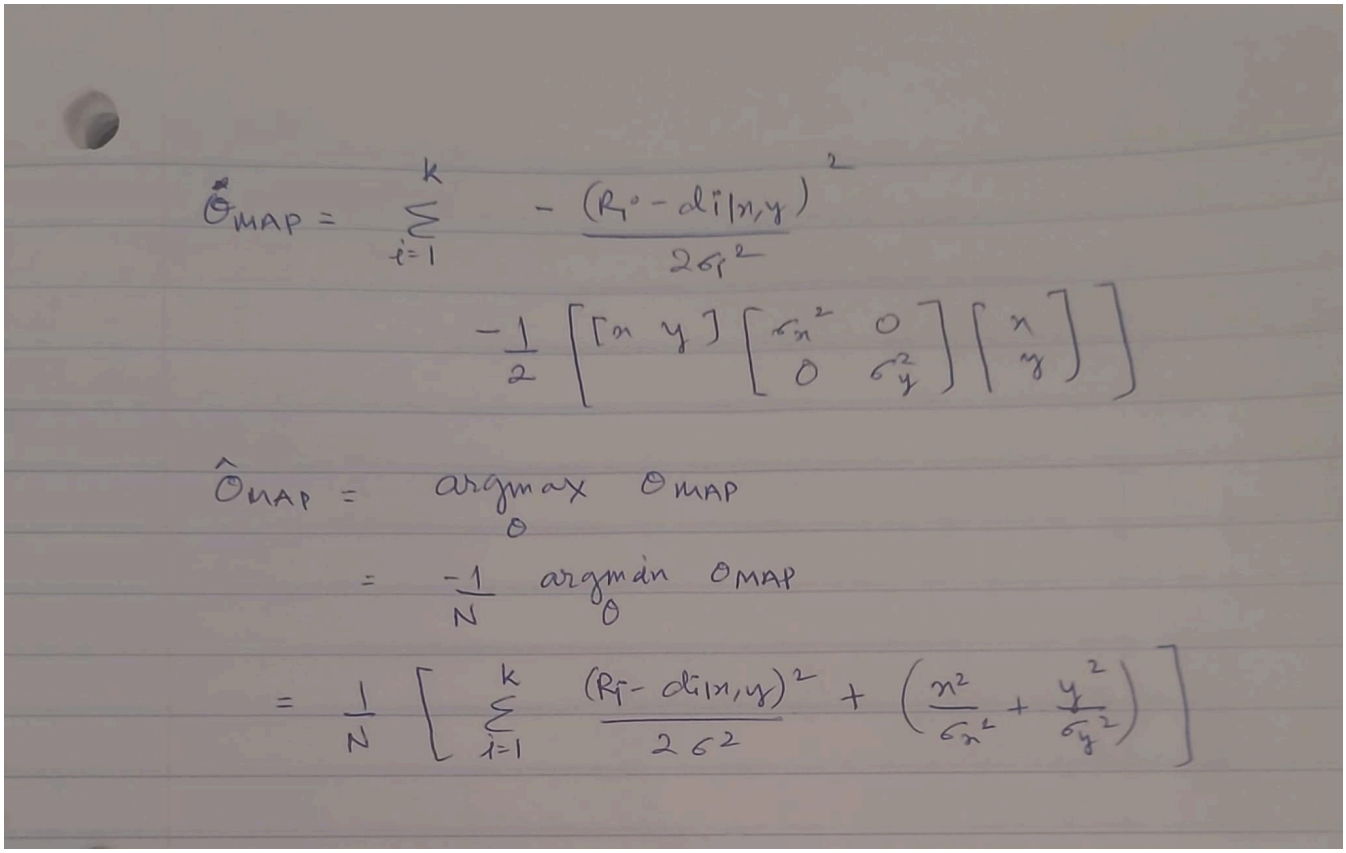
$$R_i = \sqrt{(x_i - x) + (y_i - y)^2} + n_i$$

$$\mu_{R_i|\theta} = \sqrt{(x_i - x) + (y_i - y)} = d_i|x,y$$

$$\sigma_{R_i|\theta} = \sigma_i^2$$

$$P(R_i|\theta) = \log \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(R_i - d_i|x,y)^2}{2\sigma_i^2}}$$

$$= \log \frac{1}{\sqrt{2\pi\sigma_i^2}} \text{ independent of } x,y \quad \frac{(R_i - d_i|x,y)^2}{2\sigma_i^2}$$



In [ ]:
```
np.random.seed(3)
```

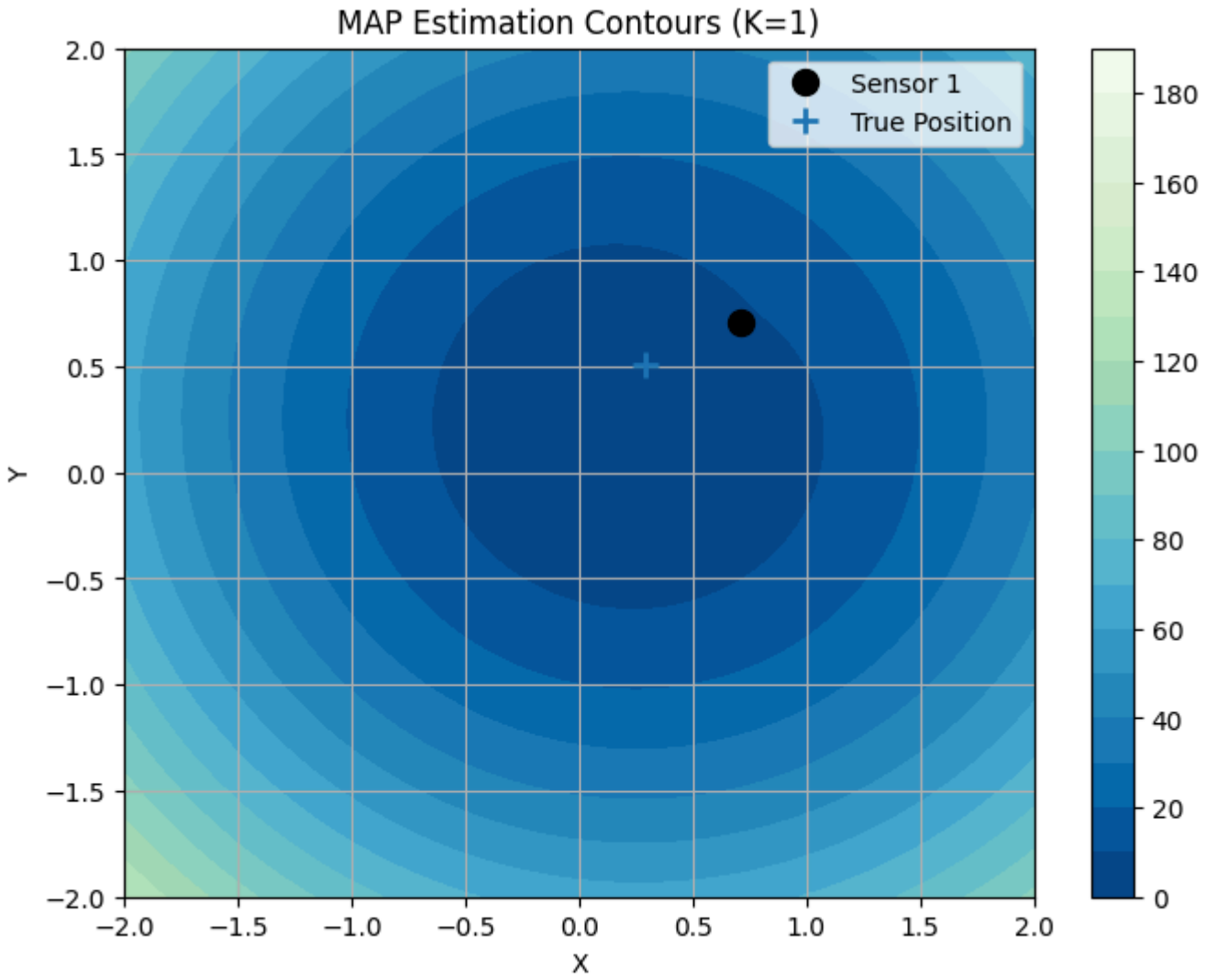
In [ ]:
```
sensor_pos_1 = (1 / np.sqrt(2), 1 / np.sqrt(2))
sensor_pos_2 = (-1 / np.sqrt(2), 1 / np.sqrt(2))
sensor_pos_3 = (-1 / np.sqrt(2), -1 / np.sqrt(2))
sensor_pos_4 = (1 / np.sqrt(2), -1 / np.sqrt(2))
landmarks = np.array([sensor_pos_1, sensor_pos_2, sensor_pos_3, sensor_pos_4])
```

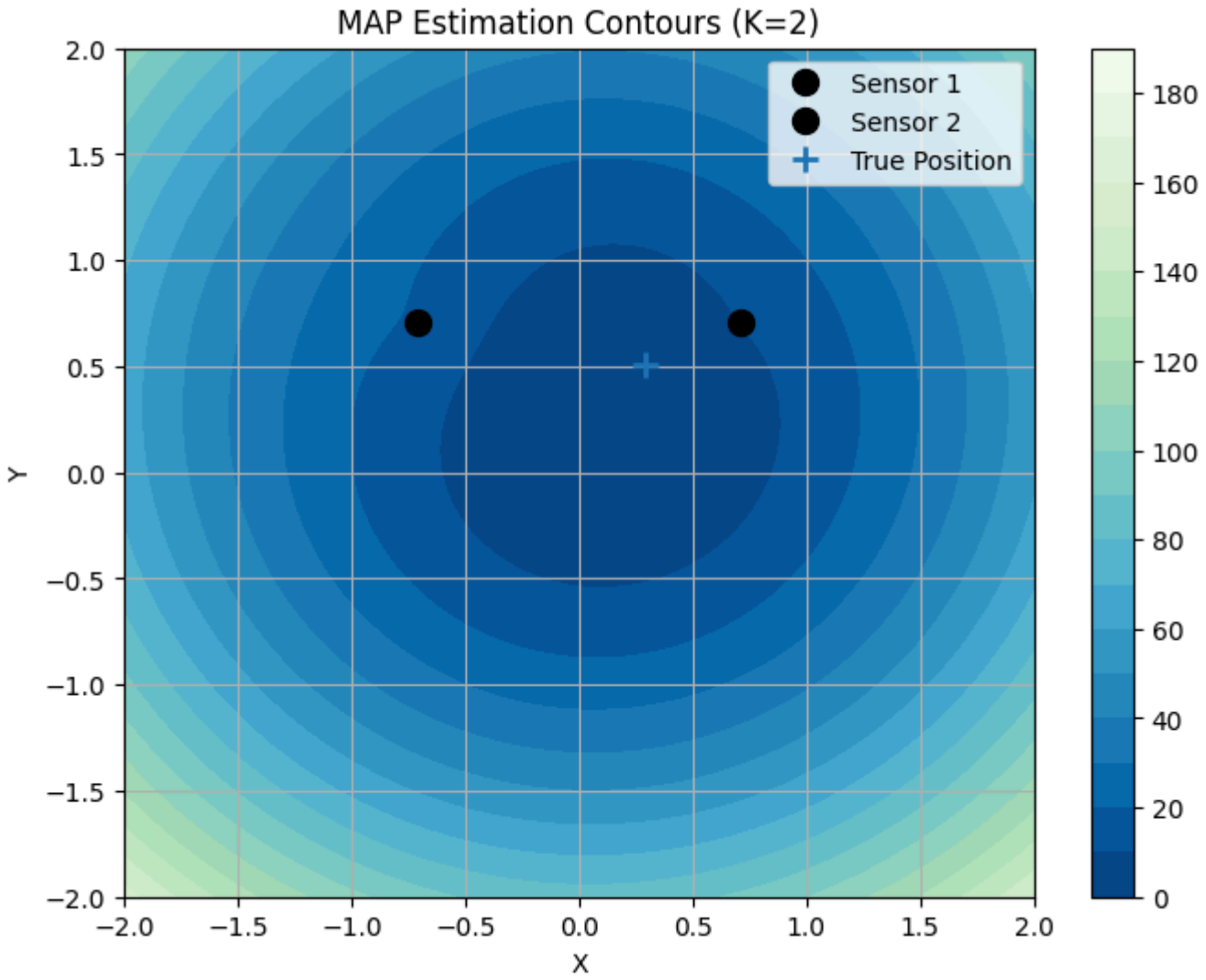
In [ ]:
```
pos = np.random.rand(2, 1)
```

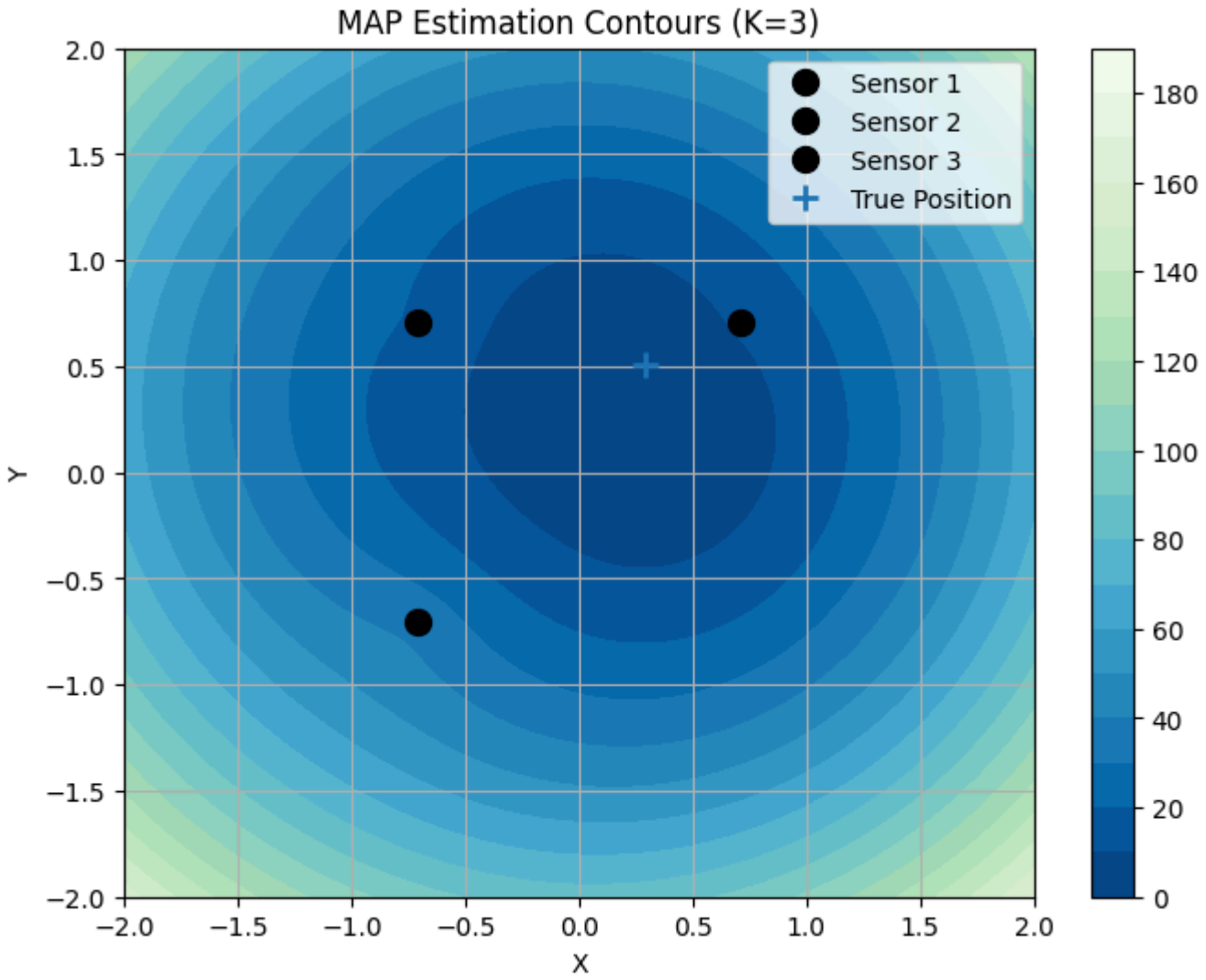
In [ ]:
```
sigma_x = 0.25
sigma_y = 0.25
sigma_i = 0.3
```

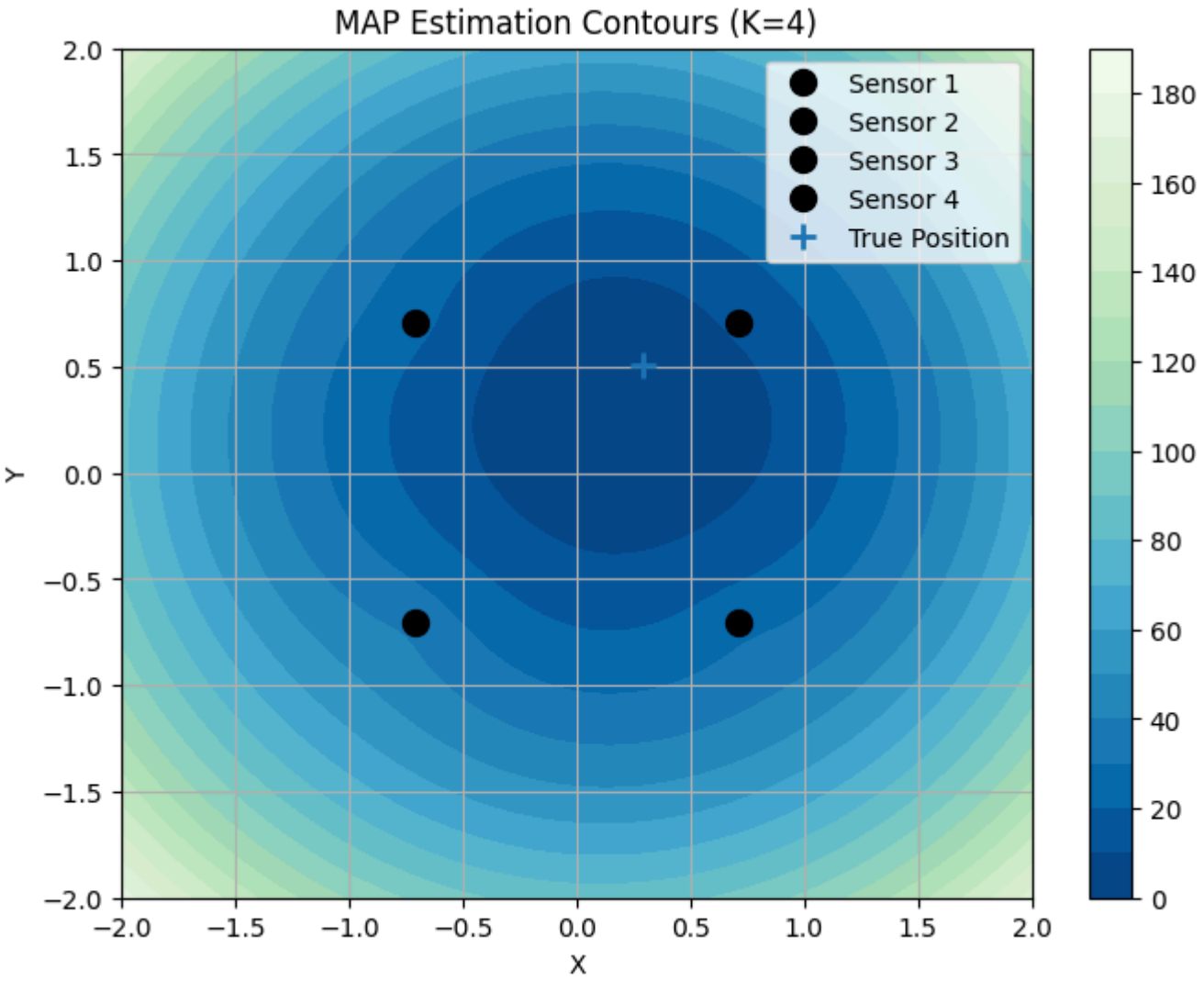
In [ ]:
```
contour_levels = np.arange(0, 200, 10)


def generate_measurements(true_pos, landmarks, noise_std=sigma_i):
    distances = np.sqrt(
        (landmarks[:, 0] - true_pos[0]) ** 2 + (landmarks[:, 1] - true_pos[1]) ** 2
    )
    noisy_measurements = distances + np.random.normal(0, noise_std, len(distances))
    return noisy_measurements


def compute_map_objective_k(
    x,
    y,
    measurements,
    landmarks,
    k,
    sigma_meas=sigma_i,
```

```
    sigma_x=sigma_x,
    sigma_y=sigma_y,
):
    prior_term = (x**2) / (2 * sigma_x**2) + (y**2) / (2 * sigma_y**2)
    sensor_term = 0

    for i in range(k):
        di = np.sqrt((landmarks[i, 0] - x) ** 2 + (landmarks[i, 1] - y) ** 2)
        sensor_term += (np.square(measurements[i] - di)) / (2 * sigma_meas**2)

    return sensor_term + prior_term


def plot_map_estimation():
    true_pos = np.random.rand(2, 1)
    measurements = generate_measurements(true_pos, landmarks)

    x = np.linspace(-2, 2, 200)
    y = np.linspace(-2, 2, 200)
    X, Y = np.meshgrid(x, y)

    for k in range(1, 5):
        plt.figure(figsize=(8, 6))

        Z = np.zeros_like(X)
        for i in range(len(x)):
            for j in range(len(y)):
                Z[j, i] = compute_map_objective_k(
                    X[j, i], Y[j, i], measurements, landmarks, k
                )

        plt.contourf(X, Y, Z, levels=contour_levels, cmap="GnBu_r")

        for i in range(k):
            plt.plot(
                landmarks[i, 0],
                landmarks[i, 1],
                "ko",
                markersize=10,
                label=f"Sensor {i+1}",
            )

        plt.plot(
            true_pos[0], true_pos[1], "+", markersize=10, label="True Position", mew=2
        )

        plt.colorbar()
        plt.grid(True)
        plt.xlim([-2, 2])
        plt.ylim([-2, 2])
        plt.title(f"MAP Estimation Contours (K={k})")
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.legend()
        plt.show()


# Run the plot function
plot_map_estimation()
```


MAP Estimation Contours (K=1)
Sensor 1
True Position
X
Y
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
180
160
140
120
100
80
60
40
20
0


MAP Estimation Contours (K=2)
Sensor 1
Sensor 2
True Position
X
Y
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
180
160
140
120
100
80
60
40
20
0


MAP Estimation Contours (K=3)
Sensor 1
Sensor 2
Sensor 3
True Position
X
Y
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
180
160
140
120
100
80
60
40
20
0



The code calculates the value of the MAP objective function across a grid of points, aiming to estimate the most probable position of a true location based on noisy distance measurements from multiple landmarks. For each point on the grid, it balances two factors: the prior term and the range term. Lower MAP values on the contour plot indicate areas where the estimated position is likely closer to the true position. The function then visualizes these scores as contour plots, with each plot showing increasingly accurate estimates as more landmarks are considered. The true position is marked, and the contours indicate areas of high likelihood for the object's location.

It can be concluded that increasing the number of landmarks significantly enhances the localization accuracy of the vehicle. As more reference points are introduced, the contours converge around the true position of the car, indicating reduced uncertainty and increased confidence in the localization estimates. This demonstrates the effectiveness of the MAP estimation method in achieving precise vehicle localization through improved data aggregation.

Assignment 2 - Question 4

Question 4 (20%)

Problem 2.13 from Duda-Hart-Stork textbook:

Section 2.4

13. In many pattern classification problems one has the option either to assign the pattern to one of c classes, or to *reject* it as being unrecognizable. If the cost for rejects is not too high, rejection may be a desirable action. Let

$$\lambda(\alpha_i|\omega_j) = \begin{cases} 0 & i = j \qquad i, j = 1, \ldots, c \\ \lambda_r & i = c + 1 \\ \lambda_s & \text{otherwise,} \end{cases}$$

where λ_r is the loss incurred for choosing the $(c + 1)$th action, rejection, and λ_s is the loss incurred for making any substitution error. Show that the minimum risk is obtained if we decide ω_i if $P(\omega_i|\mathbf{x}) \geq P(\omega_j|\mathbf{x})$ for all j and if $P(\omega_i|\mathbf{x}) \geq 1 - \lambda_r/\lambda_s$, and reject otherwise. What happens if $\lambda_r = 0$? What happens if $\lambda_r > \lambda_s$?

The loss matrix can be defined as

.	1	2	3	...	n
1	0	λ_s	λ_s	...	λ_s
2	λ_s	0	λ_s	...	λ_s
3	λ_s	λ_s	0	...	λ_s
.	.	.	.	.	.
n	λ_s	λ_s	λ_s	...	0
n+1	λ_r	λ_r	λ_r	...	λ_r

$$R(D=i|x) = \sum_{i=1}^{c} \lambda(\alpha_i|j)\, P(w_j|x)$$

$$R(\alpha_i|x) = \sum_{i=1}^{c} \lambda(\alpha_i|w_j)\, P(w_j|x)$$

for $i=j$, $\lambda_{ij} = 0$

$$\therefore \quad R(\alpha_i|x) = \sum_{i \neq j}^{c} \lambda_s\, P(w_j|x)$$

$$\sum_{j=1}^{c} P(\alpha_j|x) = 1$$

$$\sum_{j=1}^{c} P(w_j|x) = 1 - P(w_i|x)$$

$$R(\alpha_i|x) = \lambda_s\,(1 - P(w_i|x))$$

and for $c+1$ case,

$$R(\alpha_{c+1}|x) = \sum_{j=1}^{c} P(w_j|x)\,\lambda_r$$

$$= \lambda_r$$

In choosing a class i we choose it when the risk is min.

So, when we have 2 classes i, j (reject class) we choose i when $R(D=i|x) \leq R(D=j|x)$

$$\lambda_s\,(1 - P(w_i|x)) \leq \lambda_r$$

$$1 - P(w_i|x) \leq \frac{\lambda_r}{\lambda_s}$$

$$P(w_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s} \quad \text{---(1)}$$

Case 1: $\lambda_r = 0$

$$P(w_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$$

$$\lambda_r = 0$$

$$P(w_i|x) \geq 1 - \frac{0}{\lambda_s}$$

$$P(w_i|x) \geq 1$$

Decision Rule will ~~always~~ mostly decide the reject class & only decide the class i in case the posterior ≥ 1 which generally is not the case.

Case 2: $\lambda_r > \lambda_s$

$$P(w_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s}$$

if $\frac{\lambda_r}{\lambda_s} > 1$ as $\lambda_r > \lambda_s$

$$P(w_i|x) \geq 1 - (\text{something greater than } 1)$$

which is false as $P(x) > 0$

Decision Rule will always choose some class and never the reject class as the cost of rejecting is high.

Assignment 2 - Question 5

Question 5 (20%)

Let Z be drawn from a categorical distribution (takes discrete values) with K possible outcomes/states and parameter θ, represented by $Cat(\Theta)$. Describe the value/state using a 1-of-K scheme for $\mathbf{z} = [z_1, \ldots, z_K]^T$ where $z_k = 1$ if variable is in state k and $z_k = 0$ otherwise. Let the parameter vector for the pdf be $\Theta = [\theta_1, \ldots, \theta_K]^T$, where $P(z_k = 1) = \theta_k$, for $k \in \{1, \ldots, K\}$.

Given $D\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ with iid samples $\mathbf{z}_n \sim Cat(\Theta)$ for $n \in \{1, \ldots, N\}$:

- What is the ML estimator for Θ?
- Assuming that the prior $p(\Theta)$ for the parameters is a Dirichlet distribution with hyperparameter α, what is the MAP estimator for Θ?

Hint: The Dirichlet distribution with parameter α is

$$p(\Theta|\alpha) = \frac{1}{B(\alpha)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \quad \text{where the normalization constant is} \quad B(\alpha) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^{K} \alpha_k)}$$

$z = \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix}$ $z_k = 1$ if k else 0

$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_k \end{bmatrix}$ we need to estimate parameter vector θ

$P(z_k = 1) = \theta_k$ for $k \in \{1, \ldots k\}$

$D \{z_1, \ldots z_n\}$ are iid samples
$z_n \sim$ ~~Cat (θ)~~ Cat (θ) for $n \{1, \ldots n\}$

$\theta \geq 0$ as these are probabilities
$\sum_{i=1}^{k} \theta_i = 1$ as these are probabilities
$\theta^T \cdot 1 = 1$

① Max likelihood

$\theta_{ML} = \underset{\theta}{\text{argmax}} \; P(D|\theta)$

converting into minization & taking log

$\theta_{ML} = \frac{1}{N} \underset{\theta}{\text{argmin}} \; \log P(D|\theta)$

Using lagrangian optimization method, we can maximize log likelihood with a constraint $\theta^T \cdot 1 = 1$ & take the partial derivative

$$L(\theta, \lambda) = \underset{\theta}{\operatorname{argmin}} \frac{-1}{N} \sum_{i=1}^{N} \log P(z_i|\theta) + \lambda(\theta^T 1 - 1)$$

$$\frac{\partial L(\theta, \lambda)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}\left(\frac{-1}{N} \sum_{i=1}^{N} \log P(z_i|\theta) + \lambda(\theta^T 1 - 1)\right)$$

$$P(z_i|\theta) = \prod_{k=1}^{K} \theta_k^{z_{ik}}$$

$$\log P(z_i|\theta) = \log \prod_{k=1}^{K} \theta_k^{z_{ik}}$$

$$= \sum_{k=1}^{K} z_{ik} \log \theta_k$$

$$\frac{\partial L(\theta, \lambda)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}\left[\left(\frac{-1}{N} \sum_{i=1}^{10} \sum_{k=1}^{K} z_{ik} \log \theta_k\right) + \lambda(\theta^T 1 - 1)\right]$$

$$\frac{\partial L}{\partial \theta_k} = \frac{-N_k}{N\theta_k} + \lambda$$

$$0 = \frac{-N_k}{N\theta_k} + \lambda$$

$$0 = -N_k + N\theta_k \lambda$$

~~$0 = N\theta_k + N_k$~~

$$\theta_k = \frac{N_k}{N\lambda}$$

$$\sum_{k=1}^{K} \theta_k = \sum_{1}^{N} \frac{N_k}{N\lambda}$$

$$\sum_{i=1}^{k} \theta_k = 1$$

$$1 = \sum_{i}^{N} \frac{N_k}{N\lambda}$$

$$1 = \frac{1}{N\lambda} \sum_{i}^{N} N_k$$

$$\sum_{i}^{N} N_k = N$$

$$N\lambda = N$$

$$\lambda = 1$$

$$\therefore \quad \theta_k = \frac{N_k}{N}$$

(2) MAP Estimation.

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

$$= \underset{\theta}{\operatorname{argmax}}\ P(D|\theta) \cdot P(\theta)$$

$P(D)$

taking log

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

$$= \underset{\theta}{\operatorname{argmax}}\ \frac{1}{N} \sum_{i=1}^{N} \ln P(z_i|\theta) + \ln \prod_{i=1}^{k} \theta_k^{(\alpha_k - 1)}$$

$$- \ln B(\alpha)$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \ln \theta_k + \sum_{k=1}^{K} (\alpha_k - 1) \ln \theta_k$$

$$= \sum_{k=1}^{K} \left(\frac{N_k}{N} + \alpha_k - 1 \right) \ln \theta_k$$

$N_k = \sum_{i=1}^{N} z_{ik}$ from previous solution

$$\sum_{k=1}^{K} \theta_k = 1$$

$$= \sum_{k=1}^{K} \left(\frac{N_k}{N} + \alpha_k - 1 \right) \ln \theta_k + \lambda \left(\sum_{k=1}^{K} \theta_k - 1 \right)$$

$$\frac{\partial L}{\partial \theta_k} = \frac{N_k}{N\theta_k} + \frac{\alpha_k - 1}{\theta_k} + \lambda$$

$$0 = \frac{N_k}{N\theta_k} + \frac{(\alpha_k - 1)}{\theta_k} + \lambda$$

$$-\lambda = \frac{N_k + N(\alpha_k - 1)}{N\theta_k}$$

$$\theta_k = -\left[\frac{N_k + N(\alpha_k - 1)}{N\lambda} \right]$$

$$\sum_{k=1}^{K} \theta_k = \sum_{k=1}^{K} -\left[\frac{N_k + N(\alpha_k - 1)}{N\lambda} \right]$$

$$1 = \frac{-1}{\lambda} \sum_{k=1}^{K} \left[\frac{N_k + N(\alpha_k - 1)}{N} \right]$$

$$-\lambda = \sum_{k=1}^{k} \frac{N_k}{N} + \sum_{k=1}^{k} \frac{(\alpha_k - 1)}{N}$$

$$\sum_{k=1}^{k} N_k = N$$

$$-\lambda = \frac{N}{N} + \sum_{k=1}^{k} (\alpha_k - 1)$$

$$-\lambda = 1 + \sum_{k=1}^{k} \alpha_k - k$$

$$\theta_k = \frac{N_k + N(\alpha_k - 1)}{-N\lambda}$$

$$\theta_k = \frac{N_k + N(\alpha_k - 1)}{N\left(1 + \sum_{k=1}^{N} \alpha_k - k\right)}$$

$$A = \sum_{k=}^{N} \alpha_k - k$$

$$\theta_k = \frac{N_k + N(\alpha_k - 1)}{N(1 + A)}$$

$$= \frac{N_k + N(\alpha_k - 1)}{N + A}$$

$$\theta_k = \frac{N_k + \alpha_k - 1}{N + \sum_{k=1}^{k} \alpha_k - k}$$

References

1. Lecture Notes

2. Code shared as part of the Google Drive

3. Pattern Classification by Duda, Hart and Stork

4. StatsQuest - https://statquest.org/