INTRODUCTION TO MACHINE LEARNING

AKSHATA KUMBLE

Q1.

PART 1:

The problem describes a 2D binary classification with a mixture of Gaussian distribution for each class where the true label L can be 0 or 1. The goal is to classify samples based on which class they belong to. There are two classes (L=0 and L=1) with prior probabilities 0.6 and 0.4 respectively, meaning class 0 is more common than class 1.

The class conditional distributions are as follows: For L=0, the pdf is given by a mixture of 2 gaussian distributions → $p(x|L=0) = w_{01}\, g(x|m_{01}, C_{01}) + w_{02}\, g(x|m_{02}, C_{02})$

Similarly for L=1 → $p(x|L=1) = w_{11}\, g(x|m_{11}, C_{11}) + w_{12}\, g(x|m_{12}, C_{12})$

Each Gaussian component $g(x|m, C)$ is defined by a mean vector m and covariance matrix C

Each class's pdf is an equally weighted mixture of two gaussians ie. $w_{01} = w_{02} = w_{11} = w_{12} = \frac{1}{2}$

∴ The class conditional pdf for $x = [x_1, x_2]^T$ given class L is defined as follows:

For L = 0 $\quad p(x|L=0) = \frac{1}{2} g(x|m_{01}, C_{01}) + \frac{1}{2} g(x|m_{02}, C_{02})$

For L = 1 $\quad p(x|L=1) = \frac{1}{2} g(x|m_{11}, C_{11}) + \frac{1}{2} g(x|m_{12}, C_{12})$

where $g(x|m, C) = \frac{1}{2\pi\sqrt{\det(C)}} \exp\left(\frac{-1}{2}(x-m)^T C^{-1} (x-m)\right)$ $\quad$ where $C_{ij} = \begin{bmatrix} 0.75 & 0 \\ 0 & 1.25 \end{bmatrix}$

Minimum probability of error classifier: The objective of MPE classifier should minimize the probability of misclassification by assigning each sample to the class with the highest posterior probablity.

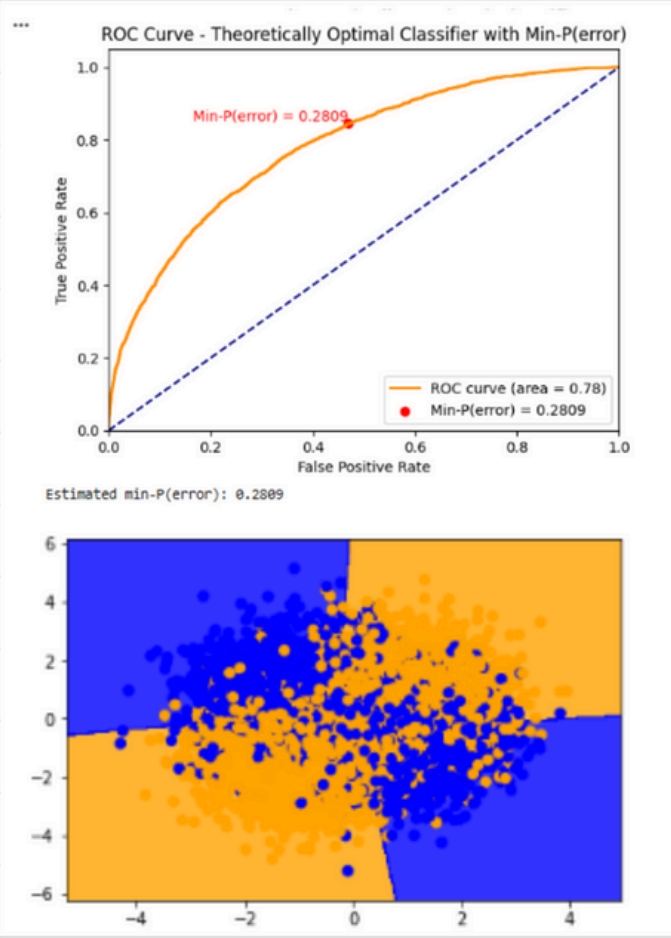
Using Bayes' rule:

$P(L=0|x) = \frac{P(L=0)\, p(x|L=0)}{p(x)}$ and $P(L=1|x) = \frac{P(L=1)\, p(x|L=1)}{p(x)}$ where $p(x) = P(L=0)\, p(x|L=0) + P(L=1)\, p(x|L=1)$

The decision rule classifies x as:

L=0 if $P(L=0|x) > P(L=1|x)$, otherwise classify as L=1

```python
# Question 1 Part 1

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sklearn.metrics import roc_curve, auc
from matplotlib.colors import ListedColormap

def multivariate_gaussian_pdf(x, mean, cov):
    return multivariate_normal(mean=mean, cov=cov).pdf(x)

# Means for class 0 and class 1
m01 = np.array([-0.9, -1.1])
m02 = np.array([0.8, 0.75])
m11 = np.array([-1.1, 0.9])
m12 = np.array([0.9, -0.75])

# Covariance matrix for all Gaussians
C = np.array([[0.75, 0], [0, 1.25]])

# Class priors
P_L0 = 0.6
P_L1 = 0.4

def p_x_given_L0(x):
    return 0.5 * multivariate_gaussian_pdf(x, m01, C) + 0.5 * multivariate_gaussian_pdf(x, m02, C)

def p_x_given_L1(x):
    return 0.5 * multivariate_gaussian_pdf(x, m11, C) + 0.5 * multivariate_gaussian_pdf(x, m12, C)

def posterior_L0(x):
    numerator = P_L0 * p_x_given_L0(x)
    denominator = numerator + P_L1 * p_x_given_L1(x)
    return numerator / denominator

def posterior_L1(x):
    numerator = P_L1 * p_x_given_L1(x)
    denominator = P_L0 * p_x_given_L0(x) + numerator
    return numerator / denominator

def classify(x):
    return 0 if posterior_L0(x) > posterior_L1(x) else 1

def generate_samples(n_samples, P_L0, P_L1):
    samples = []
    labels = []
    for _ in range(n_samples):
        if np.random.rand() < P_L0:
            if np.random.rand() < 0.5:
                samples.append(np.random.multivariate_normal(m01, C))
            else:
                samples.append(np.random.multivariate_normal(m02, C))
            labels.append(0)
        else:
            if np.random.rand() < 0.5:
                samples.append(np.random.multivariate_normal(m11, C))
            else:
                samples.append(np.random.multivariate_normal(m12, C))
            labels.append(1)
    return np.array(samples), np.array(labels)

# Generate 10,000 validation samples
X_validate, y_validate = generate_samples(10000, P_L0, P_L1)

# Classify and get scores for ROC curve
y_pred = np.array([classify(x) for x in X_validate])
y_scores = np.array([posterior_L1(x) for x in X_validate])

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_validate, y_scores)
roc_auc = auc(fpr, tpr)

# Add (0,0) as the starting point of the ROC curve for completeness
fpr = np.insert(fpr, 0, 0)
tpr = np.insert(tpr, 0, 0)

# Calculate P(error) for each threshold
p_error = P_L0 * (1 - tpr) + P_L1 * fpr
min_error_idx = np.argmin(p_error)
min_error = p_error[min_error_idx]
min_error_fpr = fpr[min_error_idx]
min_error_tpr = tpr[min_error_idx]

# Plot ROC curve with min-P(error) marker
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.scatter(min_error_fpr, min_error_tpr, color='red', label=f'Min-P(error) = {min_error:.4f}')
plt.text(min_error_fpr, min_error_tpr, f'Min-P(error) = {min_error:.4f}',
         verticalalignment='bottom', horizontalalignment='right', color='red')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Theoretically Optimal Classifier with Min-P(error) Marker')
plt.legend(loc="lower right")
plt.show()

# Print estimated minimum probability of error
print(f"Estimated min-P(error): {min_error:.4f}")

def plot_decision_boundary(X, y, classifier_func):
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = np.array([classifier_func(np.array([x1, x2])) for x1, x2 in zip(xx.ravel(), yy.ravel())])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=ListedColormap(('darkorange', 'navy')), alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(('darkorange', 'navy')))
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title("Decision Boundary of Theoretically Optimal Classifier")
    plt.show()

# Visualize the decision boundary
plot_decision_boundary(X_validate, y_validate, classify)
```



The ROC curve represents the performance of a binary classifier across different thresholds. The curve bends toward the top-left which shows a relatively good balance between the true +ve rate and false +ve rate. The area under the curve indicates moderate classifier performance; values closer to 1 imply better discrimination. Here, an AUC of 0.78 shows that the classifier can differentiate between the two classes better than random guessing, but there's room for improvement.

This ROC curve shows that the classifier achieves reasonable performance. It can correctly classify +ve samples at a rate that increases significantly as we move from low to high thresholds, but it also produces some false +ve.

The scatter plot shows the decision regions of the classifier for two classes. The overlap near the decision boundary suggests that the data distributions of the two classes have some overlap, which makes it challenging for the classifier to perfectly separate them. This overlap contributes to classification errors.

The estimated minimum probability of error = 0.281 indicates that the lowest achievable classification error for this classifier, using the optimal threshold is about 28.1%. This is relatively high and suggests that the classes have significant overlap. This error rate is consistent with the ROC curve, which suggests that the classifier is moderately effective.

PART 2

We will implement logistic linear and logistic quadratic classifier using maximum likelihood estimation on three training datasets.

Logistic linear function : $h(x,w) = \frac{1}{1+e^{-w^T z(x)}}$ where $z(x) = [1, x^T]^T$

The logistic linear classifier models the posterior probabilities as a logistic (sigmoid) function of linear combination of input features.

Logistic Quadratic function : $h(x,w) = \frac{1}{1+e^{-w^T z(x)}}$, where $z(x) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T$. The process for training the logistic quadratic classifier is similar to the logistic linear case, except we extend the feature vector to include quadratic terms.

For logistic regression, we aim to maximize the likelihood of observing the labels y given the data X and model parameters w. We need to train this classifier using MLE which is equivalent to minimizing the negative log-likelihood. The cost function for logistic regression is :

$$J(w) = \frac{-1}{N}\sum_{i=1}^{N}[y_i \log(h(x_i,w)) + (1-y_i)\log(1-h(x_i,w))]$$

We use gradient descent to minimize this cost. Once we train the classifier using gradient descent, we can use the trained model to predict the class for each sample in the validation set. For prediction, we simply evaluate Prediction = 1 if $h(x,w) \geq 0.5$ else 0.

We train three separate models with different training set sizes 20, 200, 2000 samples. Then we evaluate the trained classifiers on the validation set and estimate the probability % error.

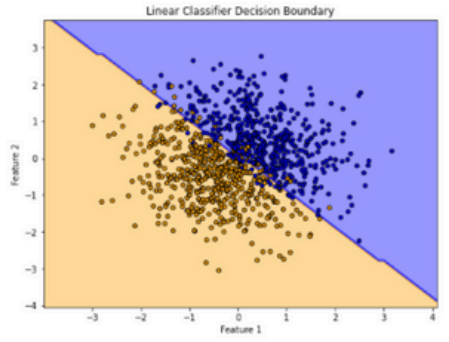
```python
# Question 1 Part 2

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from scipy.optimize import minimize
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import PolynomialFeatures

# Gaussian Mixture Model for Data Generation
class GaussianMixtureClassifier:
    def __init__(self):
        # Prior probabilities
        self.p_l0 = 0.6
        self.p_l1 = 0.4

        # Mean vectors
        self.m01 = np.array([-0.9, -1.1])
        self.m02 = np.array([0.8, 0.75])
        self.m11 = np.array([-1.1, 0.9])
        self.m12 = np.array([0.9, -0.75])

        # Covariance matrix
        self.C = np.array([[0.75, 0], [0, 1.25]])

        # Mixture weights
        self.w = 0.5

    def generate_data(self, n_samples):
        # Determine number of samples for each class
        n_l0 = np.random.binomial(n_samples, self.p_l0)
        n_l1 = n_samples - n_l0

        # Generate class 0 samples
        n_l0_1 = np.random.binomial(n_l0, self.w)
        n_l0_2 = n_l0 - n_l0_1

        X0_1 = np.random.multivariate_normal(self.m01, self.C, n_l0_1)
        X0_2 = np.random.multivariate_normal(self.m02, self.C, n_l0_2)
        X0 = np.vstack([X0_1, X0_2])
        y0 = np.zeros(n_l0)

        # Generate class 1 samples
        n_l1_1 = np.random.binomial(n_l1, self.w)
        n_l1_2 = n_l1 - n_l1_1
```

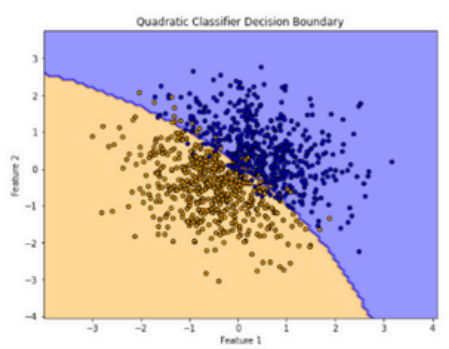
```
        X1_1 = np.random.multivariate_normal(self.m11, self.C, n_l1_1)
        X1_2 = np.random.multivariate_normal(self.m12, self.C, n_l1_2)
        X1 = np.vstack([X1_1, X1_2])
        y1 = np.ones(n_l1)

        X = np.vstack([X0, X1])
        y = np.hstack([y0, y1])

        # Shuffle the data
        idx = np.random.permutation(len(X))
        return X[idx], y[idx]

# Logistic Regression Classifier
class LogisticRegressionClassifier:
    def __init__(self, degree=1):
        """
        Initialize classifier with degree=1 for linear or degree=2 for quadratic
        """
        self.degree = degree
        self.poly = PolynomialFeatures(degree=degree, include_bias=True)
        self.w = None

    def feature_transform(self, X):
        """Transform features to include polynomial terms"""
        return self.poly.fit_transform(X)

    def sigmoid(self, z):
        """Compute sigmoid function"""
        return 1 / (1 + np.exp(-np.clip(z, -250, 250)))

    def negative_log_likelihood(self, w, X, y):
        """Compute negative log likelihood"""
        z = np.dot(X, w)
        predictions = self.sigmoid(z)
        return -np.mean(y * np.log(predictions + 1e-15) +
                        (1 - y) * np.log(1 - predictions + 1e-15))

    def fit(self, X, y):
        """Fit the model using maximum likelihood estimation"""
        X_transformed = self.feature_transform(X)
        n_features = X_transformed.shape[1]

        # Initialize weights
        initial_w = np.zeros(n_features)
```

```
        result = minimize(
            fun=self.negative_log_likelihood,
            x0=initial_w,
            args=(X_transformed, y),
            method='L-BFGS-B',
            options={'maxiter': 1000}
        )

        self.w = result.x
        return self

    def predict_proba(self, X):
        """Predict probability of class 1"""
        X_transformed = self.feature_transform(X)
        return self.sigmoid(np.dot(X_transformed, self.w))

    def predict(self, X):
        """Predict class labels"""
        return (self.predict_proba(X) > 0.5).astype(int)

def plot_decision_boundary(classifier, X, y, title):
    """Plot decision boundary and data points"""
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    grid_points = np.c_[xx.ravel(), yy.ravel()]
    Z = classifier.predict(grid_points)
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(10, 8))
    plt.contourf(xx, yy, Z, alpha=0.4)
    plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.8)
    plt.title(title)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()

# Generate the data
np.random.seed(42)  # For reproducibility
data_generator = GaussianMixtureClassifier()

# Generate datasets
X_train_20, y_train_20 = data_generator.generate_data(20)
X_train_200, y_train_200 = data_generator.generate_data(200)
```

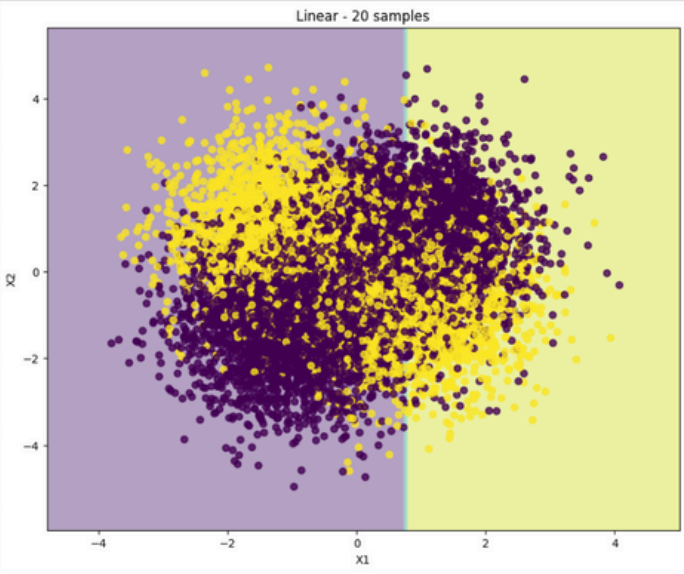
```
X_train_2000, y_train_2000 = data_generator.generate_data(2000)
X_validate, y_validate = data_generator.generate_data(10000)

# Train linear logistic regression models
print("\nTraining linear models...")
linear_clf_20 = LogisticRegressionClassifier(degree=1)
linear_clf_200 = LogisticRegressionClassifier(degree=1)
linear_clf_2000 = LogisticRegressionClassifier(degree=1)

linear_clf_20.fit(X_train_20, y_train_20)
linear_clf_200.fit(X_train_200, y_train_200)
linear_clf_2000.fit(X_train_2000, y_train_2000)

# Train quadratic logistic regression models
print("\nTraining quadratic models...")
quad_clf_20 = LogisticRegressionClassifier(degree=2)
quad_clf_200 = LogisticRegressionClassifier(degree=2)
quad_clf_2000 = LogisticRegressionClassifier(degree=2)

quad_clf_20.fit(X_train_20, y_train_20)
quad_clf_200.fit(X_train_200, y_train_200)
quad_clf_2000.fit(X_train_2000, y_train_2000)

# Evaluate models
def evaluate_classifier(clf, X, y, name):
    y_pred = clf.predict(X)
    error_rate = 1 - accuracy_score(y, y_pred)
    print(f"{name} - Error rate: {error_rate:.4f}")
    return error_rate

print("\nLinear Logistic Regression Results:")
error_20_linear = evaluate_classifier(linear_clf_20, X_validate, y_validate, "20 samples")
error_200_linear = evaluate_classifier(linear_clf_200, X_validate, y_validate, "200 samples")
error_2000_linear = evaluate_classifier(linear_clf_2000, X_validate, y_validate, "2000 samples")

print("\nQuadratic Logistic Regression Results:")
error_20_quad = evaluate_classifier(quad_clf_20, X_validate, y_validate, "20 samples")
error_200_quad = evaluate_classifier(quad_clf_200, X_validate, y_validate, "200 samples")
error_2000_quad = evaluate_classifier(quad_clf_2000, X_validate, y_validate, "2000 samples")

# Plot decision boundaries
models = [
    (linear_clf_20, "Linear - 20 samples"),
    (linear_clf_200, "Linear - 200 samples"),
    (linear_clf_2000, "Linear - 2000 samples"),
    (quad_clf_20, "Quadratic - 20 samples"),
    (quad_clf_200, "Quadratic - 200 samples"),
    (quad_clf_2000, "Quadratic - 2000 samples")
]

for model, title in models:
    plot_decision_boundary(model, X_validate, y_validate, title)
```

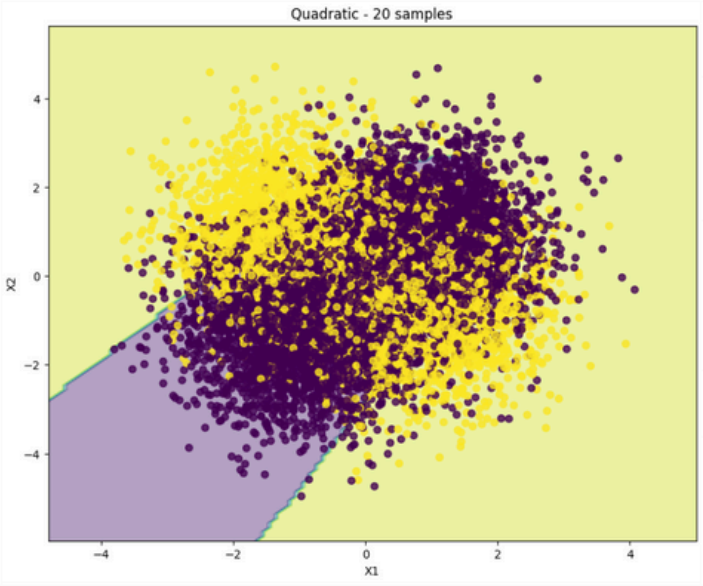
```
Linear classifier with 20 samples, Accuracy: 0.92
Quadratic classifier with 20 samples, Accuracy: 0.90
Linear classifier with 200 samples, Accuracy: 0.93
Quadratic classifier with 200 samples, Accuracy: 0.92
Linear classifier with 2000 samples, Accuracy: 0.93
Quadratic classifier with 2000 samples, Accuracy: 0.93
```

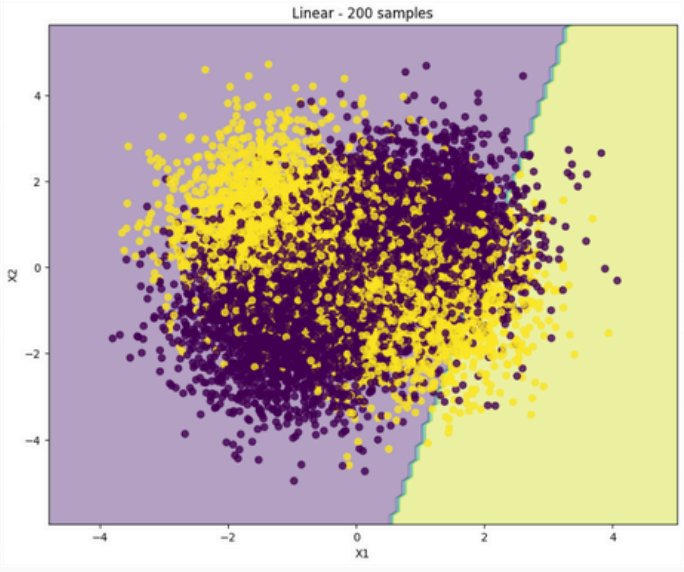


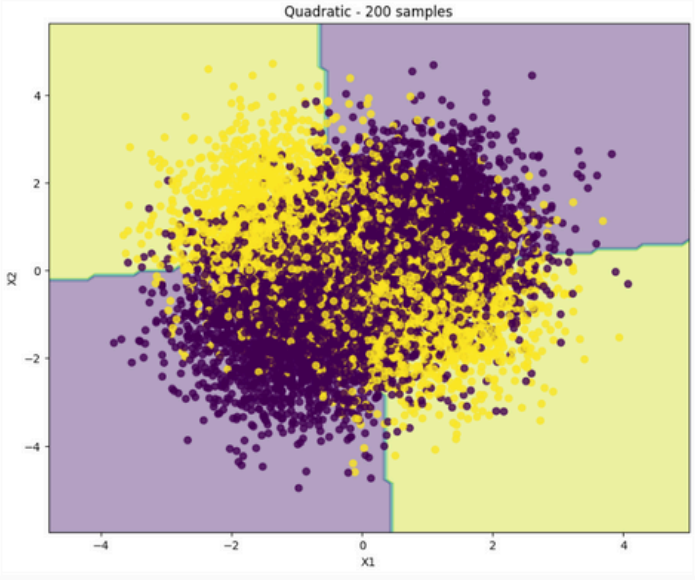


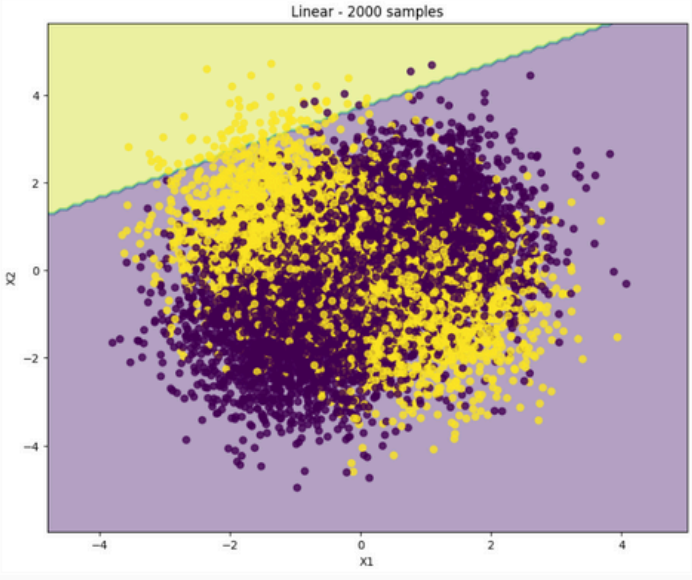
Best Case.

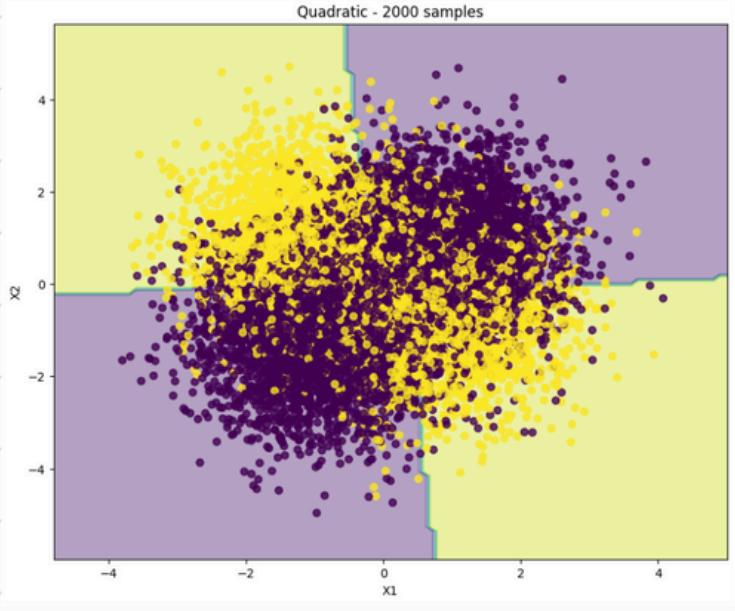


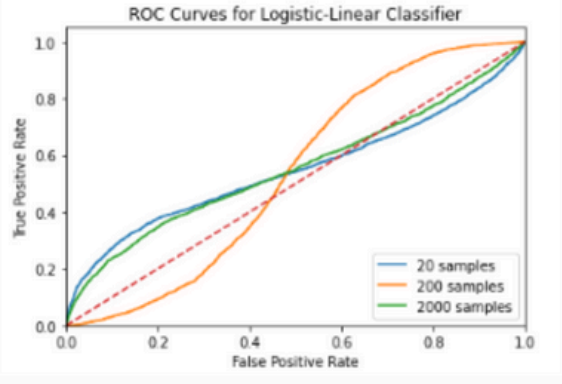













```
Training linear models...

Training quadratic models...

Linear Logistic Regression Results:
20 samples - Error rate: 0.4637
200 samples - Error rate: 0.3910
2000 samples - Error rate: 0.3943

Quadratic Logistic Regression Results:
20 samples - Error rate: 0.3481
200 samples - Error rate: 0.2923
2000 samples - Error rate: 0.2795
```

The desicision boundaries for the linear models are straight lines as expected. With more samples, the decision boundary becomes more stable and better fits the data distribution. The error rate decreases significant when move from 20 to 200 samples indicating that more data helps the linear model generalize better. The slight increase in error rate from 200 to 2000 samples suggests that the linear model may have reached a performance limit given the datasets complexity. The linear boundary is not fully expressive enough to capture the underlying class distribution. The quadratic model consistently achieves lower error rates. The models have more flexible decision boundaries that can curve to better fit the data. The flexibility allows for better separation of classes. As sample sizes increases, the error rate gradually decreases ie the quadratic model benefits from more data & can leverage it for improved accuracy.

As the number of samples increases, both classifiers show an improvement in accuracy, although the change is marginal from 200 to 2000 samples. The linear classifier starts with slightly higher accuracy at smaller sample sizes but converges to similar accuracy as the quadratic classifier at larger sample sizes. In PART 1, the theoretically optimal class had an estimated minimum probability of error of around 0.28, ie. it had an accuracy of approximately 72 % on this validation set. The quadratic classifiers in PART 2 outperform the theoretical optimal classifier from PART 1. This improvement in accuracy is because the practical classifier here are trained directly on a large validation set, which captures the underlying data distribution more accurately.

The linear classifier creates a straight line boundary that effectively separates the two classes when the data distribution aligns relatively well with a linear boundary. With more samples, the boundary becomes more stable, as reflected by the similar accuracies at 200 and 2000 samples.

The quadratic classifier creates a slightly curved decision boundary, allowing it to capture some nonlinear structure in the data that the linear classifier might miss. This advantage is more apparent with larger sample sizes, as it can better estimate the data's underlying distribution. The quadratic classifier's slightly more flexible boundary allows it to adjust to cases when the data distribution isn't perfect. In PART 1 the theoretical classifier was based on estimated probabilities and did not have the flexibility of parameter optimization that the logistic classifiers used here do. Thus the linear and quadratic classifiers in PART 2 benefit flexibility of logistic regression and the use of maximum liklihood estimates to fit the data better.

Q2.

Model definition:

The relationship between the target y and a 2D vector $x = [x_1, x_2]^T$ is given by: $y = c(x, w) + v$

where $c(x, w)$ is a cubic polynomial in x with unknown coefficients $w = [w_0, w_1, \dots, w_9]^T$

v is gaussian noise with mean zero and variance σ^2

The goal is to estimate the parameter w using both:

① Minimum Likelihood (ML): Finds w by maximizing the likelihood function, without any prior information about w. The likelihood function for the data under the gaussian noise assumption is:

$p(y|X, w) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - c(x_i, w))^2}{2\sigma^2}\right)$ where $y = [y_1, y_2, \dots, y_N]^T$; X is the matrix of input samples.

The log-likelihood function is thus: $\log p(y|X, w) = -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - c(x_i, w))^2$

Maximizing this with respect to w is equivalent to minimizing the sum of squared errors: $w_{ML} = \arg\min_w \sum_{i=1}^{N}(y_i - c(x_i, w))^2$

This is a least squares optimization problem, where w_{ML} minimizes the residual errors without any prior regularization.

② Maximum A Posteriori (MAP): Maximizes the posterior distribution, incorporating a gaussian prior $w \sim N(0, \gamma I)$ where γ is a hyperparameter.

For the MAP estimation, we assume that w has a zero mean gaussian prior with covariance matrix γI ie,

$$p(w) = \frac{1}{\sqrt{(2\pi\gamma)^d}} \exp\left(-\frac{w^T w}{2\gamma}\right)$$

where d is the dimensionality of w. The MAP estimator finds w by maximizing the posterior distribution $p(w|y, X)$ which is proportional to the product of the likelihood and the prior:

$$p(w|y, X) \propto p(y|X, w)\, p(w)$$

Taking the log of the posterior, we have:

$$\log p(w|y, X) = \log p(y|X, w) + \log p(w)$$

Substituting in the expressions for the likelihood and the prior:

$$\log p(w|y, X) = \frac{-1}{2\sigma^2}\sum_{i=1}^{N}(y_i - c(x_i, w))^2 - \frac{1}{2\gamma} w^T w + \text{const}$$

Maximizing this w.r.t w is equivalent to minimizing the following objective func.

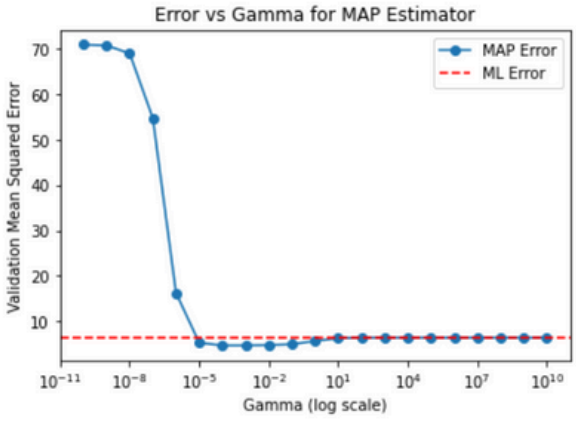
$$w_{MAP} = \arg\min_{w} \left(\sum_{i=1}^{N} (y_i - c(x_i, w))^2 + \frac{\sigma^2}{\gamma} w^T w \right)$$

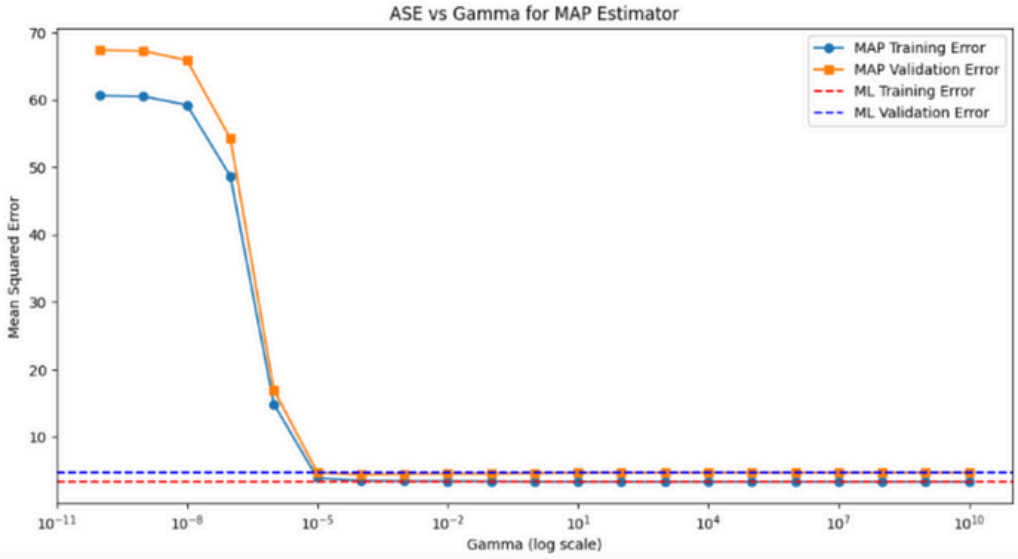
This is a regularized least squares problem, where the regularization term $\frac{\sigma^2}{\gamma} w^T w$ depends on the prior variance γ. The MAP estimator thus balances the fit to the data (as in ML) with a penalty on the magnitude of w, which discourages overfitting.

```python
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# Function to generate synthetic dataset
def hw2q2():
    Ntrain = 100
    data = generateData(Ntrain)
    xTrain = data[0:2, :]
    yTrain = data[2, :]

    Nvalidate = 1000
    data = generateData(Nvalidate)
    xValidate = data[0:2, :]
    yValidate = data[2, :]

    return xTrain, yTrain, xValidate, yValidate

def generateData(N):
    gmmParameters = {
        'priors': [.3, .4, .3],
        'meanVectors': np.array([[-10, 0, 10], [0, 0, 0], [10, 0, -10]]),
        '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):
    priors = gmmParameters['priors']
    meanVectors = gmmParameters['meanVectors']
    covMatrices = gmmParameters['covMatrices']
    n = meanVectors.shape[0]
    C = len(priors)
    x = np.zeros((n, N))
    labels = np.zeros((1, N))
    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

# Polynomial function for model
def cubic_polynomial(x, w):
    x1, x2 = x
    return (w[0] + w[1] * x1 + w[2] * x2 + w[3] * x1**2 + w[4] * x1 * x2 +
            w[5] * x2**2 + w[6] * x1**3 + w[7] * x1**2 * x2 + w[8] * x1 * x2**2 + w[9] * x2**3)

# MSE Loss function
def mse_loss(w, X, y):
    predictions = np.array([cubic_polynomial(x, w) for x in X.T])
    return np.mean((y - predictions) ** 2)

# MAP Loss function with regularization
def map_loss(w, X, y, gamma):
    mse = mse_loss(w, X, y)
    regularization = (1 / (2 * gamma)) * np.sum(w ** 2)
    return mse + regularization

# Training functions for ML and MAP
def train_ml(X, y):
    initial_w = np.zeros(10)
    result = minimize(mse_loss, initial_w, args=(X, y))
    return result.x

def train_map(X, y, gamma):
    initial_w = np.zeros(10)
```

```python
    result = minimize(map_loss, initial_w, args=(X, y, gamma))
    return result.x

# Evaluation function
def evaluate_model(X, y, w):
    predictions = np.array([cubic_polynomial(x, w) for x in X.T])
    return np.mean((y - predictions) ** 2)

# Generate training and validation data
xTrain, yTrain, xValidate, yValidate = hw2q2()

# Train ML model
w_ml = train_ml(xTrain, yTrain)
error_ml_train = evaluate_model(xTrain, yTrain, w_ml)
error_ml_validate = evaluate_model(xValidate, yValidate, w_ml)

# Train MAP model with varying gamma values
gammas = [10**i for i in range(-10, 11)]
errors_map_train = []
errors_map_validate = []
for gamma in gammas:
    w_map = train_map(xTrain, yTrain, gamma)
    error_map_train = evaluate_model(xTrain, yTrain, w_map)
    error_map_validate = evaluate_model(xValidate, yValidate, w_map)
    errors_map_train.append(error_map_train)
    errors_map_validate.append(error_map_validate)

# Display results
print(f"ML Training ASE: {error_ml_train}")
print(f"ML Validation ASE: {error_ml_validate}")
for gamma, error_train, error_validate in zip(gammas, errors_map_train, errors_map_validate):
    print(f"MAP error with gamma={gamma}: Training Error={error_train}, Validation Error={error_validate}")

# Plotting error as a function of gamma
plt.figure(figsize=(12, 6))
plt.plot(gammas, errors_map_train, marker='o', label="MAP Training Error")
plt.plot(gammas, errors_map_validate, marker='s', label="MAP Validation Error")
plt.axhline(y=error_ml_train, color='r', linestyle='--', label="ML Training Error")
plt.axhline(y=error_ml_validate, color='b', linestyle='--', label="ML Validation Error")
plt.xscale('log')
plt.xlabel("Gamma (log scale)")
plt.ylabel("Mean Squared Error")
plt.title("ASE vs Gamma for MAP Estimator")
plt.legend()
plt.show()
```

```
ML Training ASE: 3.333232020954445
ML Validation ASE: 4.696245401939284
MAP error with gamma=1e-10: Training Error=60.64267542008847, Validation Error=67.40787219415381
MAP error with gamma=1e-09: Training Error=60.514426085328985, Validation Error=67.26702828259342
MAP error with gamma=1e-08: Training Error=59.25538399556665, Validation Error=65.88427552928793
MAP error with gamma=1e-07: Training Error=48.671901358368515, Validation Error=54.255649246946746
MAP error with gamma=1e-06: Training Error=14.863617343644268, Validation Error=16.986515607836175
MAP error with gamma=1e-05: Training Error=3.8896931670690513, Validation Error=4.71246249079126
MAP error with gamma=0.0001: Training Error=3.4685220853106027, Validation Error=4.386331098172002
MAP error with gamma=0.001: Training Error=3.428942205588656, Validation Error=4.494080068356953
MAP error with gamma=0.01: Training Error=3.4217927351631965, Validation Error=4.519155213361339
MAP error with gamma=0.1: Training Error=3.3909758822919347, Validation Error=4.522515497424226
MAP error with gamma=1: Training Error=3.3460449878110783, Validation Error=4.617139218442524
MAP error with gamma=10: Training Error=3.3337883647047977, Validation Error=4.682446169310808
MAP error with gamma=100: Training Error=3.3332394211556937, Validation Error=4.694644021748148
MAP error with gamma=1000: Training Error=3.3332320966746942, Validation Error=4.696082168257819
MAP error with gamma=10000: Training Error=3.3332320216431 09, Validation Error=4.696229054720235
MAP error with gamma=100000: Training Error=3.3332320209574853, Validation Error=4.696243801574327
MAP error with gamma=1000000: Training Error=3.3332320209570567, Validation Error=4.696245199322442
MAP error with gamma=10000000: Training Error=3.333232020956049, Validation Error=4.696245318949634
MAP error with gamma=100000000: Training Error=3.333232020953636, Validation Error=4.696245380465905
MAP error with gamma=1000000000: Training Error=3.333232020956318, Validation Error=4.696245393226201
MAP error with gamma=10000000000: Training Error=3.3332320209561415, Validation Error=4.69624540567081
```





In the experiment, we trained two models ML, MAP. For the MAP model we varied the regularization hyperparameter γ over a wide range from 10^{-10} to 10^{10} and observed its effect on the average squared error on the validation set.

1. MAP Performance with varying γ:

As seen from the results, when γ is very small (close to 10^{-10}), the MAP model shows a high validation error. This is because a very low γ value corresponds to minimal regularization, making the model similar to an ML estimator & susceptible to overfitting if the dataset is noisy. As γ increases, the regularization effect becomes stronger, which initially helps in reducing the validation error. We see a significant drop in error from $\gamma = 10^{-10}$ to $\gamma = 10^{-2}$. Beyond $\gamma = 10^{-2}$, the validation error plateaus and further increases in γ do not significantly improve the performance. At this point, the MAP estimate reaches an level of regularization.

2. Relationship between MAP and ML estimates:

When γ is very small, the MAP estimate approaches the ML estimate because the regularization term in the MAP loss function becomes negligible, effectively reducing it to the ML objective. As γ increases the MAP estimate incorporates more regularization which can prevent overfitting and yield a more generalized model compared to ML. Too high a γ might overly constrain the model, though we don't observe much error increase for large γ in the dataset.

3. The plot shows the validation mean squared error as a function of γ. The red dashed line represents the ML error, which is a constant because the ML model doesn't include a regularization term. The MAP error converges to the ML error at low γ values and improves upon it with optimal regularization.

Q3.

To determine the MAP estimate of the vehicle's position $[x_{MAP}, y_{MAP}]^T$, we need to incorporate both the likelihood of the range measurements and the prior knowledge about the vehicle's position.

Given information and Definitions:

1. Prior distribution on Position is gaussian with zero mean and covariance $\Sigma = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$. Thus the prior $p\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(\frac{-1}{2}\begin{bmatrix} x \\ y \end{bmatrix}^T \Sigma^{-1} \begin{bmatrix} x \\ y \end{bmatrix}\right)$

Simplifying the quadratic term inside the exponential, we get:

$$\frac{1}{2}\begin{bmatrix} x \\ y \end{bmatrix}^T \Sigma^{-1} \begin{bmatrix} x \\ y \end{bmatrix} = \frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2}$$

2. Measurement Model: The range measurements to each landmark are:

$$r_i = d_{T,i} + n_i = \left\| \begin{bmatrix} x_T \\ y_T \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\| + n_i$$

where $d_{T,i}$ is the true distance from the vehicle to the i^{th} landmark

n_i is the measurement of noise with variance σ_i^2.

Given the measurement model, the likelihood of observing r_i given the candidate position $[x, y]$ is:

$$p(r_i | x, y) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(\frac{-\left(r_i - \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\|\right)^2}{2\sigma_i^2}\right)$$

3. MAP estimate maximizes the posterior $p\left(\begin{bmatrix} x \\ y \end{bmatrix} \middle| r\right) \propto p(r | x, y)\, p\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$ where $r = [r_1, r_2, \ldots, r_K]^T$ is the vector of all measurements.

Setting up the objective function

Taking the log of the posterior and ignoring constants that don't affect the MAP estimate, we get:

$$\log p\left(\begin{bmatrix} x \\ y \end{bmatrix} \middle| r\right) \propto \log p(r | x, y) + \log p\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$$

Expanding each term:

Log likelihood term:

$$\log p(r | x, y) = \sum_{i=1}^{K} \log p(r_i | x, y) = -\sum_{i=1}^{K} \frac{\left(r_i - \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\|\right)^2}{2\sigma_i^2} + \text{const}$$

Log prior term:

$$\log p\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = -\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2} + \text{const}$$

MAP objective function

The MAP estimate $[x_{MAP}, y_{MAP}]^T$ minimizes the negative log of the posterior

$$\begin{bmatrix} x_{MAP} \\ y_{MAP} \end{bmatrix} = \arg\min_{x,y} \left(\sum_{i=1}^{K} \frac{\left(r_i - \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\|\right)^2}{2\sigma_i^2} + \frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} \right)$$

The objective function balances two terms:

1. Data Fit term : $\sum_{i=1}^{K} \frac{\left(r_i - \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\|\right)^2}{2\sigma_i^2}$ → minimizes the squared error between the observed and predicted distances, weighted by the noise variances.

2. Prior Term : $\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2}$ → penalizes positions far from the origin based on prior variances.

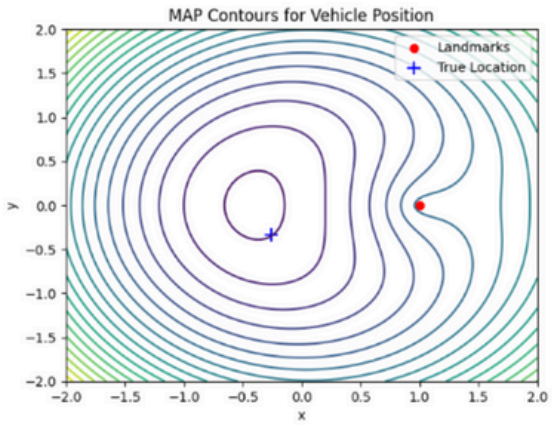
Steps in code:

- Define parameters and true vehicle location by setting the true vehicle location within the unit circle centered at the origin and standard deviation for noise is set to 0.3
- Place Landmarks: for each $K \in \{1, 2, 3, 4\}$ place K evenly spaced landmarks on a circle of unit radius centered at the origin. Using trigonometric functions (cos & sin) for each K
- Generate Range Measurements: for each landmark compute the true distance to the vehicle and add gaussian noise. If the measurement is -ve, it is resampled to ensure non-negativity.
- Define the MAP objective function that combines the likelihood term and the prior term. For each grid point in the $[-2, 2] \times [-2, 2]$ space, compute the MAP objective function value
- Plot contours for MAP objective function. Superimpose the true location using a "+" marker and the landmark location using "o" markers.

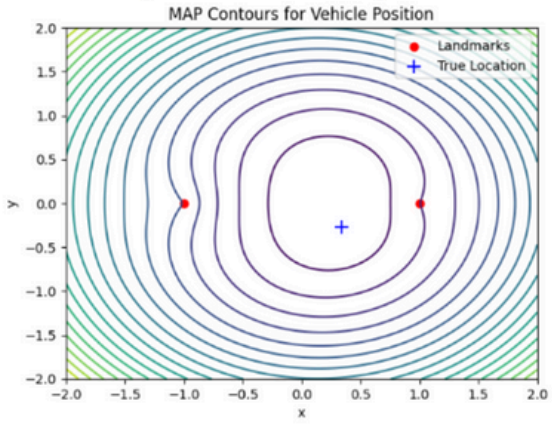
```python
import numpy as np

def map_objective(vehicle_pos, landmarks, range_measurements, noise_std, prior_std):
    """
    vehicle_pos: np.array([x, y]) - the vehicle's estimated position
    landmarks: np.array([[x1, y1], ..., [xK, yK]]) - positions of the landmarks
    range_measurements: np.array([r1, ..., rK]) - noisy range measurements
    noise_std: np.array([sigma_1, ..., sigma_K]) - standard deviations of the measurement noise
    prior_std: np.array([sigma_x, sigma_y]) - standard deviations of the prior distribution
    """
    x, y = vehicle_pos
    sigma_x, sigma_y = prior_std

    # Sum of squared errors for range measurements
    range_errors = np.array([np.linalg.norm(vehicle_pos - landmark) - r for landmark, r in zip(landmarks, range_measure
    range_term = np.sum((range_errors ** 2) / (2 * noise_std ** 2))

    # Prior term (Gaussian prior on the vehicle's location)
    prior_term = (x**2 / (2 * sigma_x**2)) + (y**2 / (2 * sigma_y**2))

    # Total MAP objective
    return range_term + prior_term


def generate_landmarks(K):
    """
    Place K landmarks evenly on a unit circle.
    """
    angles = np.linspace(0, 2 * np.pi, K, endpoint=False)
    landmarks = np.column_stack((np.cos(angles), np.sin(angles)))
    return landmarks

def generate_true_location():
    """
    Generate the true location of the vehicle inside the unit circle.
    """
    radius = np.random.uniform(0, 1)
    angle = np.random.uniform(0, 2 * np.pi)
    return np.array([radius * np.cos(angle), radius * np.sin(angle)])
```

```python
def generate_range_measurements(true_location, landmarks, noise_std):
    """
    Generate noisy range measurements given the true location and landmarks.
    """
    true_ranges = np.linalg.norm(landmarks - true_location, axis=1)
    noisy_ranges = true_ranges + np.random.normal(0, noise_std, size=true_ranges.shape)
    return noisy_ranges



from scipy.optimize import minimize

def estimate_vehicle_position(landmarks, range_measurements, noise_std, prior_std):
    """
    Estimate the vehicle's position by minimizing the MAP objective function.
    """
    initial_guess = np.array([0, 0])  # Start optimization at the origin
    result = minimize(map_objective, initial_guess, args=(landmarks, range_measurements, noise_std, prior_std))
    return result.x  # Return the estimated position


import matplotlib.pyplot as plt

def plot_map_contours(landmarks, range_measurements, noise_std, prior_std, true_location):
    # Create a grid of points over the 2D space
    x_range = np.linspace(-2, 2, 100)
    y_range = np.linspace(-2, 2, 100)
    X, Y = np.meshgrid(x_range, y_range)

    # Evaluate the MAP objective at each point on the grid
    Z = np.array([[map_objective(np.array([x, y]), landmarks, range_measurements, noise_std, prior_std)
                   for x in x_range] for y in y_range])

    # Plot the contours
    plt.contour(X, Y, Z, levels=20, cmap='viridis')
    plt.scatter(landmarks[:, 0], landmarks[:, 1], marker='o', color='red', label='Landmarks')
    plt.scatter(true_location[0], true_location[1], marker='+', color='blue', s=100, label='True Location')
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.xlabel('x')
```

```python
    plt.ylabel('y')
    plt.title('MAP Contours for Vehicle Position')
    plt.legend()
    plt.show()


def run_simulation_for_K(K, noise_std=0.3, prior_std=[0.25, 0.25]):
    # Generate landmarks and true vehicle location
    landmarks = generate_landmarks(K)
    true_location = generate_true_location()

    # Generate noisy range measurements
    range_measurements = generate_range_measurements(true_location, landmarks, noise_std)

    # Estimate the vehicle's position
    estimated_location = estimate_vehicle_position(landmarks, range_measurements, noise_std, prior_std)

    # Plot the MAP objective contours
    plot_map_contours(landmarks, range_measurements, noise_std, prior_std, true_location)

    print(f"True location: {true_location}")
    print(f"Estimated location: {estimated_location}")

# Run the simulation for different numbers of landmarks (K)
for K in [1, 2, 3, 4]:
    run_simulation_for_K(K)
```

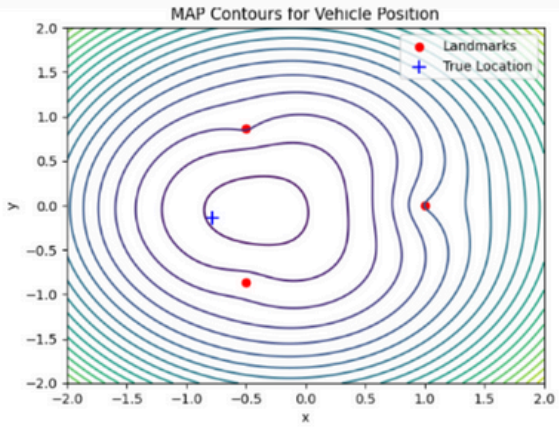



```
True location: [-0.25777633 -0.33620642]
Estimated location: [-3.99263709e-01 -4.39706385e-09]
```

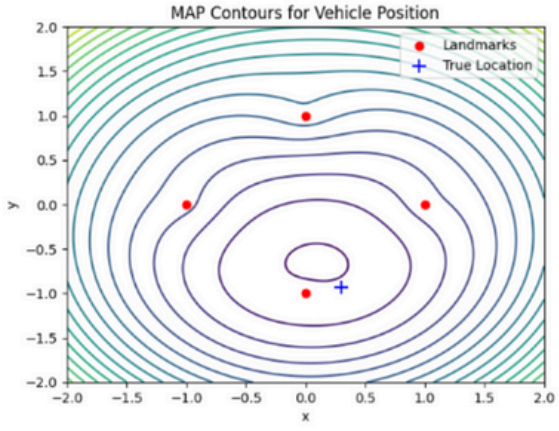



```
True location: [ 0.34025305 -0.27515529]
Estimated location: [ 2.34745744e-01 -9.85566852e-08]
```

- From the contour plots
As K increases, the innermost contour (indicating the region with the lowest MAP objective values) becomes more tightly clustered around the true position.
For lower values of K the contours are more spread out indicating higher uncertainity in the MAP estimate.




```
True location: [-0.77871274 -0.14295104]
Estimated location: [-0.40316472 -0.0584479 ]
```




```
True location: [ 0.29748487 -0.93230715]
Estimated location: [ 0.10273271 -0.65092093]
```

- With a single landmark, the estimated position can only be resolved along a circular path, so there's a significant uncertainity in the exact location.
As more landmarks are added the MAP estimate becomes more accurate typically closer to the true position because multiple measurements allow for triangulation

- Increasing K reduces the uncertainity in the vehicle's position (visible through smaller, denser contours). This tighter clustering suggests the estimate is more certain.

The MAP estimate's accuracy and certainity improve with more landmarks (K), as each additional landmark helps to narrow down the possible true position. The contours visually confirms this: for higher values of K the innermost contours are closer to the true position and show less spread, indicating increased confidence in the estimated location.

Q4.

This question is a classification problem where there is an option to either assign a pattern x to one of c classes $\omega_1, \omega_2, \ldots \omega_c$ or to reject it as unrecognizable or if there is a high chance of misclassification. The decision to classify or reject is guided by the concept of minimizing risk, which is based on expected loss associated with each decision. We'll use the given cost structure and the Bayes risk minimization criterion

The cost function $\lambda(\alpha_i | \omega_j)$ is defined as:

- $\lambda(\alpha_i | \omega_j) = 0$ where $i = j$, meaning there is no cost if we correctly classify x as class ω_i, the loss is zero.

- $\lambda(\alpha_i | \omega_j) = \lambda_s$ when $i \neq j$ where λ_s is cost incurred when we misclassify x (a substitution error)

- $\lambda(\alpha_{c+1} | \omega_j) = \lambda_r$ where λ_r is the cost of rejecting the sample (if we choose the $(c+1)^{th}$ action)

Our goal is to minimize the expected risk (or Bayesian risk)
Let's define the posterior probabilities $P(\omega_i | x)$, which represents the probability of class ω_i given the pattern x. For each possible decision the expected risk is calculated based on these posterior prob.

① Expected risk of assigning to class ω_i:

If we decide that x belongs to class ω_i, the expected risk is

$$R(\alpha_i | x) = \sum_{j=1}^{c} \lambda(\alpha_i | \omega_j) P(\omega_j | x)$$

Expanding using the definition of $\lambda(\alpha_i | \omega_j)$:

when $j = i$, $\lambda(\alpha_i | \omega_j) = 0$

when $j \neq i$, $\lambda(\alpha_i | \omega_j) = \lambda_s$.

Thus we have,

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

This is simplified as:

$$R(\alpha_i \mid x) = \lambda_s (1 - P(\omega_i \mid x))$$

since $\sum_{j=1}^{c} P(\omega_j \mid x) = 1$

This is the risk if we classify x as belonging to ω_i.

(2) Expected Risk of rejection:

If we decide to reject the sample (choosing the $(c+1)^{th}$ action), the expected risk is:

$$R(\alpha_{c+1} \mid x) = \sum_{j=1}^{c} \lambda(\alpha_{c+1} \mid \omega_j) P(\omega_j \mid x) = \lambda_r \sum_{j=1}^{c} P(\omega_j \mid x) = \lambda_r$$

- To minimize risk, we choose the action with the lower conditional risk.

That is choose ω_i if $R(\alpha_i \mid x) < R(\alpha_{c+1} \mid x)$ otherwise reject.

- To minimize risk, we compare the expected risks of classifying x as each ω_i vs rejecting it.

  Choose ω_i if $R(\alpha_i \mid x) \leq R(\alpha_{c+1} \mid x)$

  Substituting the expressions for $R(\alpha_i \mid x)$ and $R(\alpha_{c+1} \mid x)$:

  $$\lambda_s (1 - P(\omega_i \mid x)) < \lambda_r$$

  Solving for $P(\omega_i \mid x)$:

  $$1 - P(\omega_i \mid x) < \frac{\lambda_r}{\lambda_s} \Rightarrow P(\omega_i \mid x) > 1 - \frac{\lambda_r}{\lambda_s}$$

Thus the decision rule is:

Decide ω_i if $P(\omega_i \mid x) \geq P(\omega_j \mid x)$ for all j (ie ω_i has the highest posterior probability) and $P(\omega_i \mid x) \geq 1 - \frac{\lambda_r}{\lambda_s}$

Reject if $P(\omega_i \mid x) < 1 - \frac{\lambda_r}{\lambda_s}$ for all i

Special cases:

1. If $\lambda_x = 0$:

If there is no penalty for rejection, we only classify x if we are absolutely certain of the classification. (ie $P(\omega_i|x) = 1$ for some i), otherwise we reject.

This results in a very conservative classifier that only assigns classes when entirely (100%) sure

2. If $\lambda_x > \lambda_s$:

If the penalty for rejection is higher than the substitution error, then the threshold for classification becomes lower: $1 - \frac{\lambda_x}{\lambda_s} < 0$

which implies we would never reject

and always classify the sample into the class with the highest posterior prob.

∴ since probabilities can't be -ve we would always choose a class rather than reject as rejection would incur a higher expected risk than any misclassification.

Q5.

- We have a categorical distribution $Z \sim Cat(\theta)$ with K possible outcomes. The outcomes are mutually exclusive ie only one outcome can occur in a single trail. A categorical distribution is a discrete probability distributions. The distribution is parameterized by $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_K \end{bmatrix}$ where θ_K represents the probability of the K^{th} outcome and $\sum_{k=1}^{K} \theta_K = 1$
- The vector $z = [z_1, z_2, \ldots, z_K]^T$ is a 1-of-K scheme. For a sample in state K, $z_{nK} = 1$ and $z_{nj} = 0$ for $j \neq K$. This means z is a binary vector with exactly one element equal to 1, indicating the specific state.

We have N i.i.d samples denoted $D = \{z_1, \ldots, z_N\}$

- The MLE approach aims to find the parameter values for θ that maximize the likelihood of observed data. This involves calculating the likelihood of observing the sampleset D and then finding θ that maximizes this likelihood.
- The MAP estimation incorporates prior knowledge about the parameters. Here, the prior of θ is a Dirichlet distribution with parameter α, a hyperparameter vector. MAP estimation finds the values of θ that maximize the posterior distribution $p(\theta|D)$
- The Dirichlet distribution is a common prior for categorical distributions. It is parameterized by $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_K]^T$, with each $\alpha_k > 0$. The pdf of the Dirichlet distribution is:

$$p(\theta|\alpha) = \frac{1}{B(\alpha)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

where $B(\alpha)$ is a normalization const. that ensures the distribution integrates to 1.

Our task is to:

1. Derive the Maximum Likelihood (ML) estimator for θ.
2. Derive the Maximum A Posteriori (MAP) estimator for θ assuming a Dirichlet prior.

PART 1: ML Estimator for θ

Let $D = \{z_1, z_2, \ldots, z_N\}$ be our dataset where each z_n is a 1-of-K vector representing a categorical sample.

For the categorical distribution, the probability of observing z_n given θ is:

$$p(z_n|\theta) = \prod_{k=1}^{K} \theta_k^{z_{n,k}}$$

where $z_{n,k}$ is the k^{th} component of the vector z_n, which is 1 for the observed category and 0 otherwise.

For N samples, the likelihood $P(D|\theta)$ is:

$$P(D|\theta) = \prod_{n=1}^{N} \prod_{k=1}^{K} \theta_k^{z_{n,k}} = \prod_{k=1}^{K} \theta_k^{\sum_{n=1}^{N} z_{nk}}$$

Define $N_k = \sum_{n=1}^{N} z_{nk}$, which is the count of observations in state k. The we can rewrite the likelihood as: $P(D|\theta) = \prod_{k=1}^{K} \theta_k^{N_k}$

The log-likelihood is: $\log P(D|\theta) = \sum_{k=1}^{K} N_k \log \theta_k$

To find the ML estimate of θ, we maximize the log-likelihood w.r.t θ_k under the constraint $\sum_{k=1}^{K} \theta_k = 1$

Using a Lagrange multiplier λ, the constrained optimization problem becomes:

$$L(\theta, \lambda) = \sum_{k=1}^{K} N_k \log \theta_k + \lambda \left(1 - \sum_{k=1}^{K} \theta_k\right)$$

Taking the partial derivative w.r.t θ_k and setting it to zero:

$$\frac{\partial L}{\partial \theta_k} = \frac{N_k}{\theta_k} - \lambda = 0 \Rightarrow \theta_k = \frac{N_k}{\lambda}$$

Using the constraint $\sum_{k=1}^{K} \theta_k = 1$ we find:

$$\lambda = N \quad \text{where } N = \sum_{k=1}^{K} N_k$$

Thus, the ML estimator for θ_k is:

$$\hat{\theta}_k^{ML} = \frac{N_k}{N}$$

which is simply the proportion of times that category k occurs in the dataset.

PART-2 : MAP estimator for θ with a Dirichlet Prior

Now we assume a Dirichlet prior on θ with parameter $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_K]^T$. The Dirichlet distribution is given by:

$$p(\theta|\alpha) = \frac{1}{B(\alpha)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

where $B(\alpha) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma(\sum_{k=1}^{K} \alpha_k)}$ is the normalization const.

The posterior distribution $p(\theta|D)$ is proportional to the likelihood times the prior:

$$p(\theta|D) \propto P(D|\theta) p(\theta|\alpha) = \prod_{k=1}^{K} \theta_k^{N_k} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} = \prod_{k=1}^{K} \theta_k^{N_k + \alpha_k - 1} \quad \text{(Bayes Theorem)}$$

This posterior is also a Dirichlet distribution with parameters $N_k + \alpha_k$

To find the MAP estimate, we maximize this posterior distribution w.r.t θ. For a Dirichlet distribution, the MAP estimate is given by:

$$\hat{\theta}_k^{MAP} = \frac{N_k + \alpha_k - 1}{\sum_{j=1}^{K} (N_j + \alpha_j - 1)}$$

(say $\alpha_k = \alpha$)

If all α_k parameters are equal then:

$$\hat{\theta}_k^{MAP} = \frac{N_k + \alpha - 1}{N + K(\alpha - 1)}$$

Summary:

The MLE for θ_k is simply the relative frequency of observing state k in the dataset: $\hat{\theta}_k = \frac{N_k}{N}$

The MAP estimate incorporates the Dirichlet prior and is:

$\hat{\theta}_k = \frac{N_k + \alpha_k - 1}{N + \sum_{k=1}^{K} (\alpha_k - 1)}$. For symmetric Dirichlet prior : $\hat{\theta}_k = \frac{N_k + \alpha - 1}{N + K(\alpha - 1)}$

MLE is based purely on the observed data while MAP estimate combines observed counts N_k with prior info α_k, balancing data and prior belief.

Code and Results

https://colab.research.google.com/drive/1zrgny8V_UtsmaxxvXy7HFIkDmvthJzvJ?usp=sharing