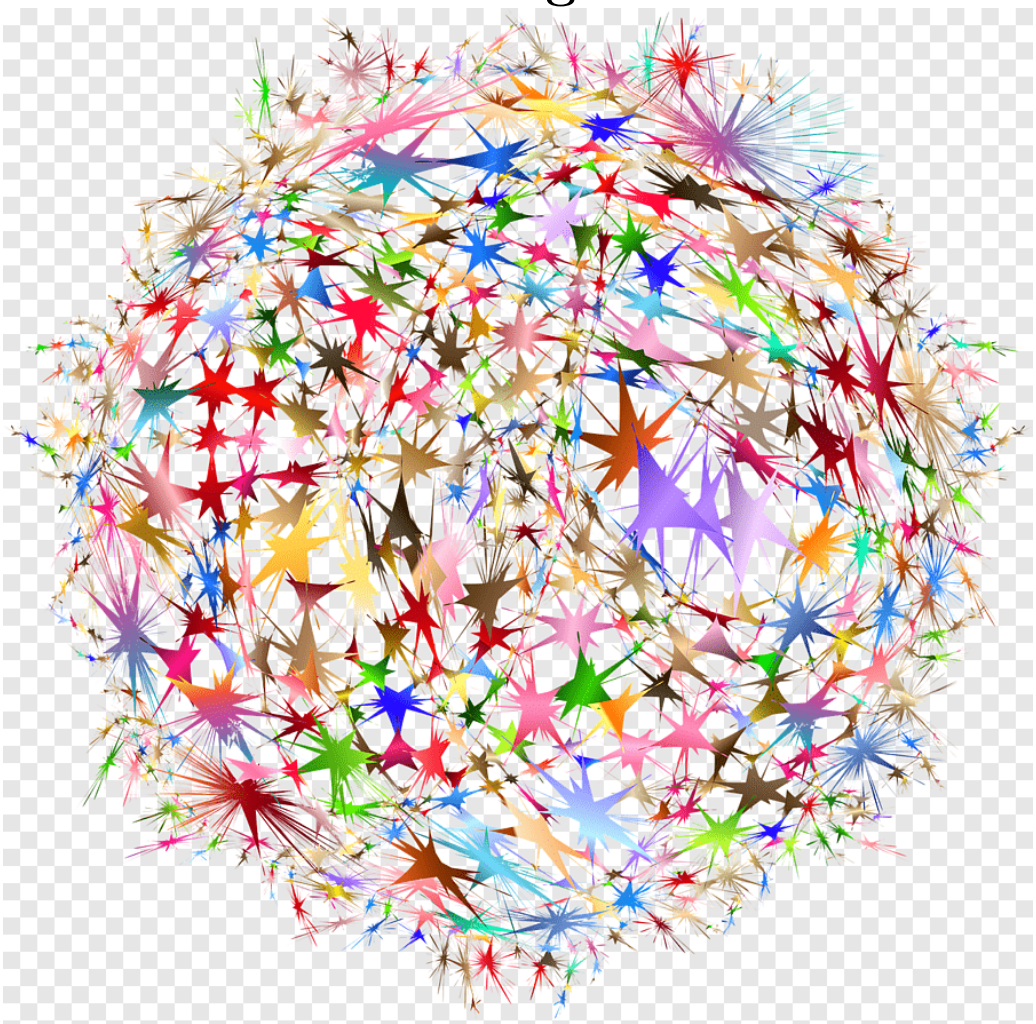
Northeastern University

Department of Electrical and Computer Engineering

EECE 5644: Machine Learning and Pattern Recognition

Homework Assignment-2



Baibhav Kumar Pathak
NUID: 002295510

Github Link: https://github.com/pathakbaibhav/ML-and-Pattern-Recognition

Question 1: 20%

The probability density function (pdf) for a 2-dimensional real-valued random vector X is as follows:

$$p(x) = P(L = 0)p(x|L = 0) + P(L = 1)p(x|L = 1)$$

Here, L is the true class label that indicates which class-label-conditioned pdf generates the data. The class priors are:

$$P(L = 0) = 0.6, \quad P(L = 1) = 0.4$$

The class-conditional pdfs are given by:

$$p(x|L = 0) = w_{01}g(x|m_{01}, C_{01}) + w_{02}g(x|m_{02}, C_{02})$$

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

where $g(x|m, C)$ is a multivariate Gaussian probability density function with mean vector m and covariance matrix C. The parameters of the class-conditional Gaussian pdfs are:

$$w_{i1} = w_{i2} = \frac{1}{2}, \quad \text{for } i \in \{0, 1\}$$

The mean vectors are:

$$m_{01} = \begin{bmatrix} -0.9 \\ -1.1 \end{bmatrix}, \quad m_{02} = \begin{bmatrix} 0.8 \\ 0.75 \end{bmatrix}$$

$$m_{11} = \begin{bmatrix} -1.1 \\ 0.9 \end{bmatrix}, \quad m_{12} = \begin{bmatrix} 0.9 \\ -0.75 \end{bmatrix}$$

The covariance matrices are identical for all pairs $\{ij\}$:

$$C_{ij} = \begin{bmatrix} 0.75 & 0 \\ 0 & 1.25 \end{bmatrix}, \quad \text{for all } \{ij\} \text{ pairs.}$$

For numerical results, generate the following independent datasets, each consisting of i.i.d. samples from the specified data distribution. Make sure to include the true class label for each sample.

- D_{20}^{train}: 20 samples and their labels for training;
- D_{200}^{train}: 200 samples and their labels for training;
- D_{2000}^{train}: 2000 samples and their labels for training;
- $D_{10K}^{\text{validate}}$: 10000 samples and their labels for validation.

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 $D_{10K}^{\text{validate}}$. 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. 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 $D_{10K}^{\text{validate}}$.

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.

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_{20}^{train}, D_{200}^{train}, and D_{2000}^{train}. 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 $D_{10K}^{\text{validate}}$, 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 2(a) using a logistic-quadratic-function-based approximation of class label posterior functions given a sample.

Discussion (2%)

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

Answer

```
# Enable inline plotting in Google Colab
%matplotlib inline

# Import necessary libraries
from sys import float_info  # Threshold smallest positive floating value
from math import ceil, floor
import matplotlib.pyplot as plt  # For general plotting
import numpy as np

# Import specific functions and classes
from scipy.optimize import minimize
from scipy.stats import multivariate_normal as mvn
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import roc_curve
```

```
# Set options for numpy to display numbers without scientific notation
np.set_printoptions(suppress=True)

# Set a random seed for reproducibility
np.random.seed(7)

# Set global plotting configurations for better visualization
# Font size of the title
plt.rc('axes', titlesize=20)
# Font size for text
plt.rc('font', size=24)
# Font size for the x and y labels
plt.rc('axes', labelsize=18)
# Font size of the x axes tick labels
plt.rc('xtick', labelsize=16)
# Font size of the y axes tick labels
plt.rc('ytick', labelsize=16)
# Legend fontsize
plt.rc('legend', fontsize=14)
# Font size of the figure title
plt.rc('figure', titlesize=24)
```

- `generate_q1_data` generates synthetic samples from a Gaussian Mixture Model (GMM) with two classes. The inputs are `num_samples` and `pdf_params`, defining means (`mu`), covariances (`Sigma`), mixture coefficients (`gmm_a`), and class priors (`priors`). Samples are allocated based on thresholds set by the priors, and labels are assigned based on class probabilities.

```
[ ] def generate_q1_data(num_samples, pdf_params):
        # Determine dimensionality from PDF parameters
        num_dimensions = pdf_params['mu'].shape[1]

        # Decide randomly which samples will come from each Gaussian: u_i ~ Uniform(0, 1) for i in {1, ..., N}
        u = np.random.rand(num_samples)

        # Determine thresholds based on the mixture weights/priors for the GMM, which need to sum to 1
        # Samples from class 0 are weighted according to 'a' and class prior probabilities contribute to labeling
        thresholds = np.cumsum(np.append(pdf_params['gmm_a'].dot(pdf_params['priors'][0]), pdf_params['priors'][1]))
        thresholds = np.insert(thresholds, 0, 0)  # For interval of classes

        # For 2 classes, use a single boolean condition to label samples
        labels = u >= pdf_params['priors'][0]
        samples = np.zeros((num_samples, num_dimensions))

        num_gaussians = len(pdf_params['mu'])
        for i in range(1, num_gaussians):
            # Get randomly sampled indices for this Gaussian between thresholds based on class priors
            indices = np.argwhere((thresholds[i - 1] <= u) & (u <= thresholds[i]))[:, 0]
            samples[indices, :] = mvn.rvs(pdf_params['mu'][i - 1], pdf_params['Sigma'][i - 1], len(indices))

        return samples, labels
```

- `estimate_roc` computes ROC curve data for a classifier. It takes `discriminant_score`, `labels`, and `N_labels` as inputs, and returns false positive rates (FPR) and true positive rates (TPR) for various thresholds (`gammas`).

```
# Generate ROC curve samples
def estimate_roc(discriminant_score, labels, N_labels):
    # Sorting necessary so the resulting FPR and TPR axes plot threshold probabilities in order as a line
    sorted_score = sorted(discriminant_score)

    # Use gamma values to account for every possible classification split
    # Epsilon handles the extremes of the ROC curve (TPR=FPR=0 and TPR=FPR=1)
    gammas = ([sorted_score[0] - float_info.epsilon]
              + sorted_score +
              [sorted_score[-1] + float_info.epsilon])

    # Calculate decision labels for each observation for every gamma
    decisions = [discriminant_score >= g for g in gammas]

    # Retrieve indices where false positives (FP) occur
    ind10 = [np.argwhere((d == 1) & (labels == 0)) for d in decisions]
    # Compute false positive rates (FPR) as a fraction of total negative class samples
    p10 = [len(inds) / N_labels[0] for inds in ind10]

    # Retrieve indices where true positives (TP) occur
    ind11 = [np.argwhere((d == 1) & (labels == 1)) for d in decisions]
    # Compute true positive rates (TPR) as a fraction of total positive class samples
    p11 = [len(inds) / N_labels[1] for inds in ind11]

    # Store the results in a dictionary for convenience
    roc = {}
    roc['p10'] = np.array(p10)  # FPR
    roc['p11'] = np.array(p11)  # TPR

    # Return the ROC data and gamma thresholds
    return roc, gammas
```

- `get_binary_classification_metrics` computes binary classification metrics (TNR, FPR, FNR, TPR) based on `predictions` and `labels`. Returns a dictionary of metrics.

```
def get_binary_classification_metrics(predictions, labels, N_labels):
    # Initialize a dictionary to store classification metrics
    class_metrics = {}

    # True Negative Rate (TNR)
    class_metrics['TN'] = np.argwhere((predictions == 0) & (labels == 0))
    class_metrics['TNR'] = len(class_metrics['TN']) / N_labels[0]

    # False Positive Rate (FPR)
    class_metrics['FP'] = np.argwhere((predictions == 1) & (labels == 0))
    class_metrics['FPR'] = len(class_metrics['FP']) / N_labels[0]

    # False Negative Rate (FNR)
    class_metrics['FN'] = np.argwhere((predictions == 0) & (labels == 1))
    class_metrics['FNR'] = len(class_metrics['FN']) / N_labels[1]

    # True Positive Rate (TPR)
    class_metrics['TP'] = np.argwhere((predictions == 1) & (labels == 1))
    class_metrics['TPR'] = len(class_metrics['TP']) / N_labels[1]

    # Return the dictionary containing all classification metrics
    return class_metrics
```

- `create_prediction_score_grid` generates a grid within specified bounds, applies an optional transformation, and calculates prediction scores. Returns grid coordinates and scores.

```
def create_prediction_score_grid(bounds_X, bounds_Y, prediction_function, phi=None, num_coords=100):
    # Create a meshgrid within the specified bounds
    xx, yy = np.meshgrid(
        np.linspace(bounds_X[0], bounds_X[1], num_coords),
        np.linspace(bounds_Y[0], bounds_Y[1], num_coords)
    )
    grid = np.c_[xx.ravel(), yy.ravel()]  # Flatten the grid coordinates

    # Apply the phi transformation if provided
    if phi is not None:
        grid = phi.transform(grid)

    # Apply the prediction function to each point on the grid
    Z = np.array([prediction_function(point) for point in grid])

    # Reshape Z to match the shape of the meshgrid
    Z = Z.reshape(xx.shape)
    return xx, yy, Z
```

- `discriminant_score_erm` calculates the log discriminant score for classifying a point using `pdf_params`. Returns the log of the score difference between classes.

```
def discriminant_score_erm(point):
    # Compute the score for class 0 and class 1 based on pdf_params
    class_0_score = (
        pdf_params['gmm_a'][0] * mvn.pdf(point, pdf_params['mu'][0], pdf_params['Sigma'][0]) +
        pdf_params['gmm_a'][1] * mvn.pdf(point, pdf_params['mu'][1], pdf_params['Sigma'][1])
    )
    class_1_score = mvn.pdf(point, pdf_params['mu'][2], pdf_params['Sigma'][2])

    # Return the log of the discriminant score
    return np.log(class_1_score) - np.log(class_0_score)
```

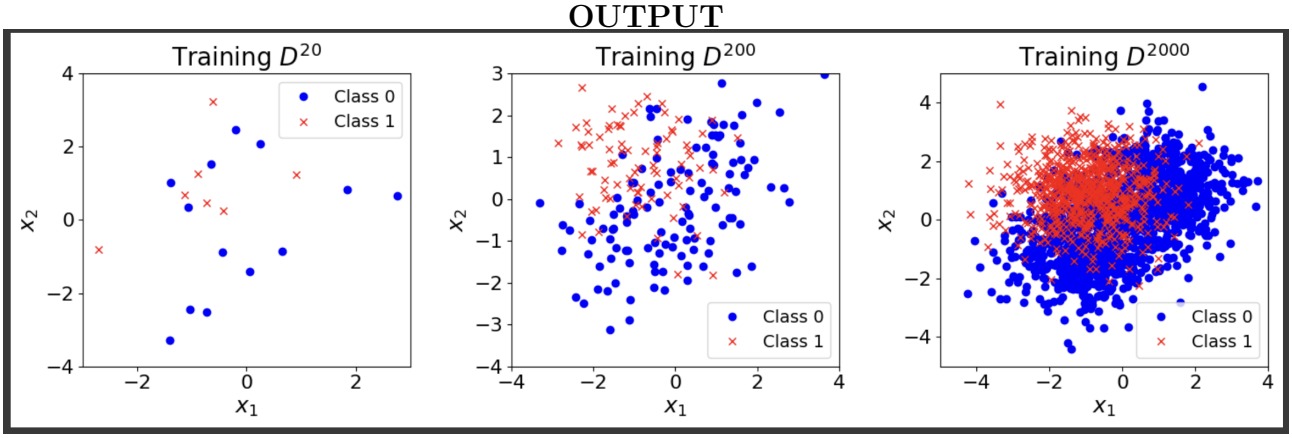
- The following code establishes distribution parameters for a Gaussian Mixture Model (GMM) and creates synthetic training and validation datasets of various sizes. The GMM distribution parameters include priors, mixture weights, means, and covariances.

- Training datasets of three sizes (20, 200, and 2000 samples) are generated using a specified data generation function, `generate_q1_data`. These datasets are then displayed in separate subplots, with points colored according to their true class labels.

- Additionally, a larger validation dataset (10,000 samples) is produced, and its class distribution is depicted in an individual subplot. The axis limits across subplots are adjusted based on the validation set to maintain a uniform view, providing a visual comparison of the synthetic datasets across different training sizes.

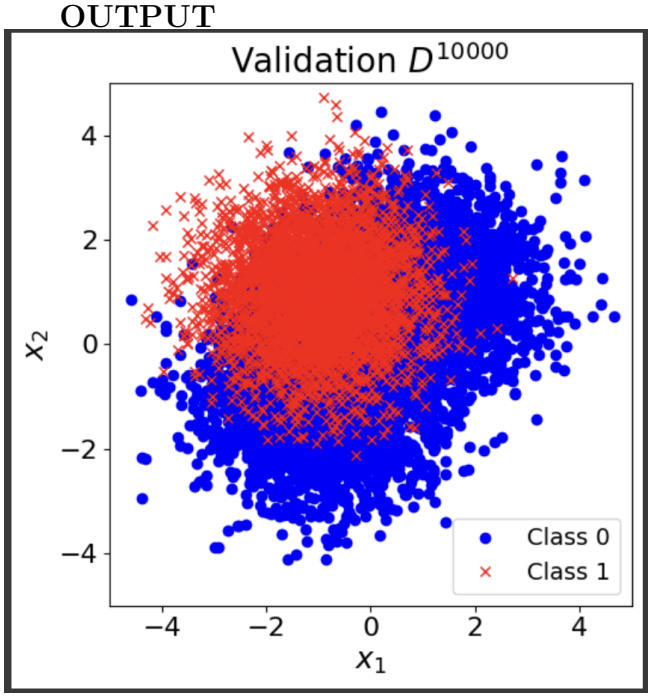
```python
# Define distribution parameters: priors, gmm_a, mu, and Sigma
pdf_params = {
    'priors': np.array([0.6, 0.4]),
    'gmm_a': np.array([0.5, 0.5]),
    'mu': np.array([[-1, -1], [1, 1], [-1, 1], [1, -1]]),
    'Sigma': np.array([
        [[1, 0], [0, 1]],
        [[1, 0], [0, 1]],
        [[1, 0], [0, 1]],
        [[1, 0], [0, 1]]
    ])
}

# Number of training input samples for experiments
N_train = [20, 200, 2000]

# Create the figure and subplots with 1 row and 3 columns (since 1 subplot was empty)
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

# Lists to store the generated samples and their labels
X_train = []
labels_train = []
N_labels_train = []

# Generate training data and plot it for different sizes
for t, N_t in enumerate(N_train):
    # Generate data for the current size
    X_t, labels_t = generate_q1_data(N_t, pdf_params)
    X_train.append(X_t)
    labels_train.append(labels_t)
    N_labels_train.append(np.array((sum(labels_t == 0), sum(labels_t == 1))))

    # Plot the data on the respective subplot
    ax[t].set_title(r"Training $D^{%d}$" % N_t)
    ax[t].plot(X_t[labels_t == 0, 0], X_t[labels_t == 0, 1], 'bo', label="Class 0")
    ax[t].plot(X_t[labels_t == 1, 0], X_t[labels_t == 1, 1], 'rx', label="Class 1")
    ax[t].set_xlabel(r"$x_1$")
    ax[t].set_ylabel(r"$x_2$")
    ax[t].legend()

    # Adjust axis limits to avoid overlap between smaller and larger datasets
    x1_lim = (floor(np.min(X_t[:, 0])), ceil(np.max(X_t[:, 0])))
    x2_lim = (floor(np.min(X_t[:, 1])), ceil(np.max(X_t[:, 1])))
    ax[t].set_xlim(x1_lim)
    ax[t].set_ylim(x2_lim)

# Adjust layout to prevent overlap
plt.tight_layout()

# Display the plots
plt.show()
```

OUTPUT



```python
# Number of Validation Samples for experiment
N_valid = 10000

# Generate validation data
X_valid, labels_valid = generate_q1_data(N_valid, pdf_params)

# Count the number of samples per class in the validation set
N1_valid = np.array((sum(labels_valid == 0), sum(labels_valid == 1)))

# Create a single plot (no subplots) for the validation data
fig, ax = plt.subplots(figsize=(8, 6))

# Plot the validation data
ax.set_title(r"Validation $D^{%d}$" % N_valid)
ax.plot(
    X_valid[labels_valid == 0, 0], X_valid[labels_valid == 0, 1], 'bo', label="Class 0"
)
ax.plot(
    X_valid[labels_valid == 1, 0], X_valid[labels_valid == 1, 1], 'rx', label="Class 1"
)
ax.set_xlabel(r"$x_1$")
ax.set_ylabel(r"$x_2$")
ax.legend()

# Set axis limits based on the data
x1_valid_lim = (floor(np.min(X_valid[:, 0])), ceil(np.max(X_valid[:, 0])))
x2_valid_lim = (floor(np.min(X_valid[:, 1])), ceil(np.max(X_valid[:, 1])))
ax.set_xlim(x1_valid_lim)
ax.set_ylim(x2_valid_lim)

# Keep axis proportions equal
ax.set_aspect('equal', adjustable='box')

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```

OUTPUT



Part A

- To achieve the theoretically optimal classifier that minimizes error probability, the Maximum A Posteriori (MAP) rule can be applied. The MAP classifier reduces misclassification risk by assigning each sample to the class with the highest posterior probability based on the observed data.

- The MAP classifier's discriminant score is computed, and decisions are made by comparing this score against a threshold determined by the logarithm of the class priors ratio (the decision boundary). The mathematical form of the MAP classifier's decision rule is:

$$\text{Decision} = \frac{p(D = 1 \mid L = 1)}{p(D = 1 \mid L = 0)} \geq \frac{P(L = 0)}{P(L = 1)}$$

- This rule minimizes the probability of error by taking into account the class priors and the likelihood of the observed data for each class. The implementation of this classifier and its application to the validation set $D_{10k}^{validate}$ is included in the code.

- The ROC curve is then constructed from the discriminant scores of the MAP classifier on the validation set. A special marker on the ROC curve indicates the point of minimum error probability for the classifier, which represents the theoretically optimal point. This minimum error probability is estimated by analyzing decision-truth label counts on the validation set.

- The following code calculates and displays the Receiver Operating Characteristic (ROC) curve for an Empirical Risk Minimization (ERM) classifier.

- ERM scores for a given validation dataset (X_{valid}) are calculated using a function, `calculate_erm_sco` based on the specified probability distribution parameters.

- Next, the code plots the ROC curve using False Positive Rate (FPR) and True Positive Rate (TPR) values derived from the empirical ERM scores.

- Points corresponding to the minimum probability of error are identified and marked on the ROC curve, showing both empirical and theoretical minimum error rates from the MAP classification rule.

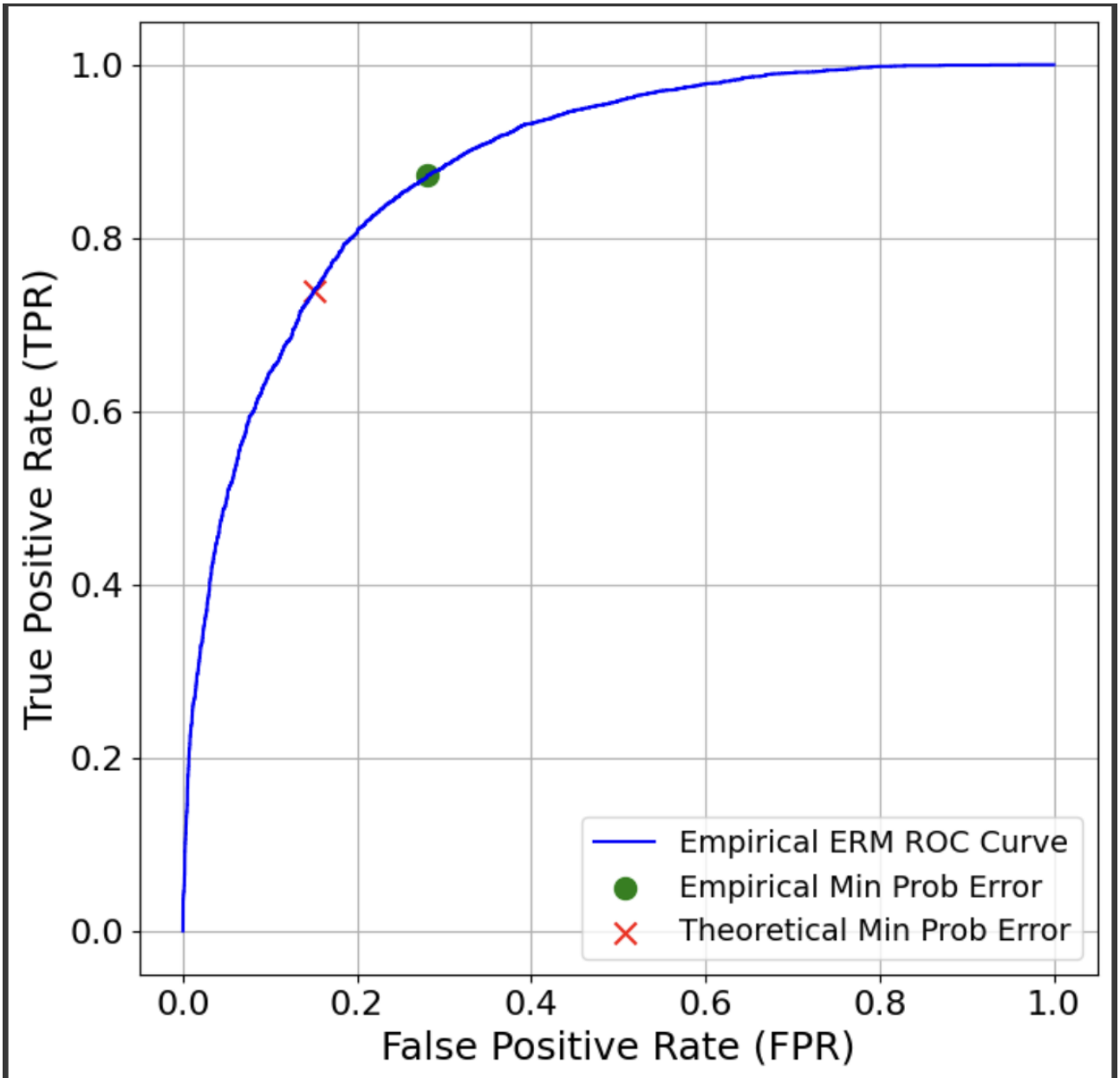
- The resulting plot offers insights into the classifier's performance, showing the trade-off between true positive and false positive rates, as well as the minimum error points from both empirical and theoretical perspectives.

- The results displayed include the empirical and theoretical minimum error probabilities along with the associated gamma values.

```python
# Function to calculate ERM scores for given data and distribution parameters
def calculate_erm_scores(X, dist_params):
    class_0 = dist_params['gmm_a'][0] * mvn.pdf(X, dist_params['mu'][0], dist_params['Sigma'][0]) + \
              dist_params['gmm_a'][1] * mvn.pdf(X, dist_params['mu'][1], dist_params['Sigma'][1])
    class_1 = mvn.pdf(X, dist_params['mu'][2], dist_params['Sigma'][2])
    return np.log(class_1 + 1e-10) - np.log(class_0 + 1e-10)  # Adding tolerance for numerical stability

# Generate ERM scores for the validation dataset
erm_scores = calculate_erm_scores(X_valid, pdf_params)

# Calculate predictions based on ERM scores
predictions = (erm_scores >= 0).astype(int)

# Generate ROC curve data using validation labels and ERM scores
fpr, tpr, thresholds = roc_curve(labels_valid, erm_scores)
roc_erm = {'p10': fpr, 'p11': tpr}

# Calculate empirical probability of error based on ROC points and class priors
prob_error_empirical = roc_erm['p10'] * pdf_params['priors'][1] + \
                       (1 - roc_erm['p11']) * pdf_params['priors'][0]

# Find the optimal threshold by minimizing classification error
threshold_tuning = np.linspace(erm_scores.min(), erm_scores.max(), 500)
best_threshold = min(threshold_tuning, key=lambda t: np.mean((erm_scores >= t).astype(int) != labels_valid))

# Calculate the empirical gamma by normalizing the best threshold
mean_score, std_score = np.mean(erm_scores), np.std(erm_scores)
min_gamma_empirical = np.exp((best_threshold - mean_score) / std_score)

# Calculate the theoretical gamma based on class priors
theoretical_gamma = pdf_params['priors'][0] / pdf_params['priors'][1]
theoretical_threshold = np.log(theoretical_gamma)

# Find the closest threshold on the ROC curve to the theoretical gamma
closest_threshold_index = np.argmin(np.abs(thresholds - theoretical_threshold))

# Calculate the theoretical minimum probability of error
theoretical_error = (pdf_params['priors'][0] * (1 - tpr[closest_threshold_index]) +
                     pdf_params['priors'][1] * fpr[closest_threshold_index])

# Plot the ROC curve along with empirical and theoretical min error points
fig, ax = plt.subplots(figsize=(8, 8))
ax.plot(fpr, tpr, label="Empirical ERM ROC Curve", color='blue')
ax.scatter(fpr[np.argmin(prob_error_empirical)], tpr[np.argmin(prob_error_empirical)], color='green', label="Empirical Min Prob Error", s=100, marker='o')
ax.scatter(fpr[closest_threshold_index], tpr[closest_threshold_index], color='red', label="Theoretical Min Prob Error", s=100, marker='x')
ax.set_xlabel("False Positive Rate (FPR)")
ax.set_ylabel("True Positive Rate (TPR)")
plt.grid(True)
plt.legend()
plt.show()
```

```python
# Display final results for empirical and theoretical minimum errors and gamma
print(f"Empirical Min Prob Error for ERM = {np.min(prob_error_empirical):.4f}")
print(f"Empirical Min Gamma = {min_gamma_empirical:.3f}")
print(f"Theoretical Min Prob Error for ERM = {theoretical_error:.4f}")
print(f"Theoretical Gamma = {theoretical_gamma:.3f}")
```

OUTPUT




```
Empirical Min Prob Error for ERM = 0.1890
Empirical Min Gamma = 1.244
Theoretical Min Prob Error for ERM = 0.2166
Theoretical Gamma = 1.500
```

Part B

- To develop logistic-linear-function-based approximations of posterior functions for class labels using maximum likelihood estimation, we can frame the logistic regression task as minimizing the negative log-likelihood of the training data. The logistic function is commonly employed to model the posterior probability of a sample belonging to the positive class in binary classification. The logistic function is expressed as:

$$P(Y = 1 \mid X) = \frac{1}{1 + e^{-(w \cdot X + b)}}$$

  where w is the weight vector, X represents the input feature vector, b is the bias term, and e is the base of the natural logarithm.

- The negative log-likelihood function for logistic regression is defined by:

$$-\log L(w,b) = -\sum_{i=1}^{N}\left[y_i \log\left(\frac{1}{1+e^{-(w\cdot x_i+b)}}\right) + (1-y_i)\log\left(1-\frac{1}{1+e^{-(w\cdot x_i+b)}}\right)\right]$$

where N represents the number of samples in the training dataset, and y_i is the actual class label of the i-th sample.

- The objective of the optimization process is to find the values of w and b that minimize the negative log-likelihood. Numerical optimization methods, such as gradient descent, can be employed for this purpose.
- After training the logistic regression models on three different training datasets (D^{20}_{train}, D^{200}_{train}, D^{2000}_{train}), the approximations of class-label-posteriors are utilized to classify samples in the validation dataset $D^{10k}_{\text{validate}}$. The classification rule for minimizing the probability of error typically involves using a threshold probability (often set at 0.5) and assigning the positive class if the estimated posterior probability meets or exceeds this threshold.
- To estimate the classification error, predictions on the validation set are compared to the true labels, and the error rate is calculated as the proportion of misclassified samples. The three logistic-linear-function-based approximations are independently evaluated to determine their performance on the validation set.

.

- `sigmoid` computes the logistic sigmoid function for a given input, mapping values to the (0, 1) range.
- `logistic_prediction_prob` calculates predicted probabilities for class 1 using the sigmoid function on input features `X` and weights `w`.
- `nll` computes the binary cross-entropy loss (Negative Log-Likelihood) between true `labels` and predicted `predictions`, with `np.clip` to avoid numerical instability.

```python
# Epsilon: Smallest positive floating value to avoid numerical instability
epsilon = 1e-7

# Define the logistic/sigmoid function
def sigmoid(z):
    return 1.0 / (1 + np.exp(-z))

# Define the prediction function: L = 1 / (1 + np.exp(-X * w))
# X.dot(w) serves as inputs to the sigmoid, referred to as logits
def logistic_prediction_prob(X, w):
    logits = X.dot(w)  # Compute the logits
    return sigmoid(logits)  # Apply the sigmoid function

# Binary cross-entropy, equivalent to NLL (Negative Log-Likelihood)
def nll(labels, predictions):
    # Epsilon adjustment to prevent underflow or overflow
    predictions = np.clip(predictions, epsilon, 1 - epsilon)

    # Log probability for the negative class: log P(L=0 | x; theta)
    log_p0 = (1 - labels) * np.log(1 - predictions + epsilon)

    # Log probability for the positive class: log P(L=1 | x; theta)
    log_p1 = labels * np.log(predictions + epsilon)

    # NLL: Mean of the per-sample negative log-likelihoods
    return -np.mean(log_p0 + log_p1, axis=0)
```

- The function `compute_logistic_params` optimizes parameters for logistic regression by maximizing the likelihood of observed binary labels.
- It initializes the parameter vector θ_0 randomly, with a size based on the number of features in the input samples.
- Defines the cost function (`cost_fun`) as the negative log-likelihood using binary cross-entropy loss (`nll`) and logistic predictions (`logistic_prediction_prob`).
- Uses the `minimize` function (likely from SciPy) to optimize θ, minimizing `cost_fun` and returning the optimal parameter vector that maximizes label likelihood.

```
[ ]  def compute_logistic_params(samples, labels):
         # Initialize theta0 with the correct dimensionality (including bias term)
         theta0 = np.random.rand(samples.shape[1])

         # Define the cost function to minimize NLL
         cost_fun = lambda w: nll(labels, logistic_prediction_prob(samples, w))

         # Use minimize function from SciPy to solve the optimization problem
         res = minimize(cost_fun, theta0, tol=1e-6)

         # Return the optimized parameter vector (theta)
         return res.x
```

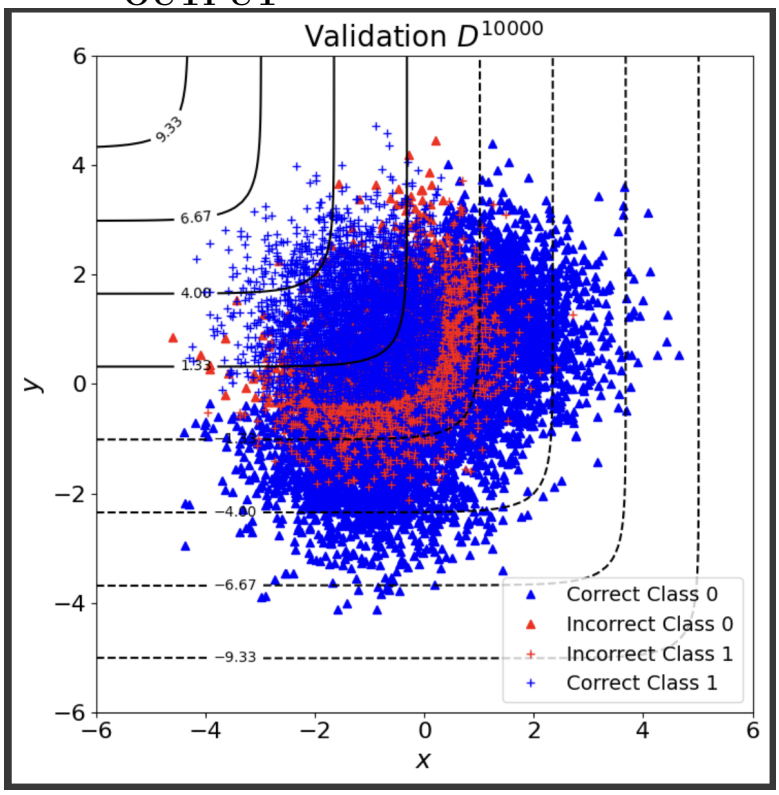
- The function `report_logistic_classifier_results` takes a dataset, logistic classifier weights, true labels, and label counts as input.
- It calculates the probability of error for the classifier, plots the data with markers indicating correct and incorrect classifications, and overlays decision contours based on the model's predictions. An optional ϕ-transformation can be applied to the input data.
- Error probability is computed using false positive rate (FPR) and false negative rate (FNR), and these metrics help highlight misclassified points on the plot. Decision boundaries are based on the ϕ-transformation, and the function returns the error probability.

```
def compute_logistic_params(samples, labels):
    # Initialize theta0 with the correct dimensionality (including bias term)
    theta0 = np.random.rand(samples.shape[1])

    # Define the cost function to minimize NLL
    cost_fun = lambda w: nll(labels, logistic_prediction_prob(samples, w))

    # Use minimize function from SciPy to solve the optimization problem
    res = minimize(cost_fun, theta0, tol=1e-6)

    # Return the optimized parameter vector (theta)
    return res.x
```

Optional: Part A

```python
def plot_erm_decision_boundaries(ax, X, prediction_function, phi=None):
    """
    Plots ERM decision boundaries using a contour plot.
    """
    # Define axis bounds based on the input data
    bounds_X = (floor(np.min(X[:, 0])) - 1, ceil(np.max(X[:, 0])) + 1)
    bounds_Y = (floor(np.min(X[:, 1])) - 1, ceil(np.max(X[:, 1])) + 1)

    # Generate grid and corresponding discriminant scores
    xx, yy, Z = create_prediction_score_grid(bounds_X, bounds_Y, prediction_function, phi)

    # Define contour levels based on Z values
    contour_levels = np.linspace(np.min(Z), np.max(Z), 10)

    # Plot the decision boundaries with contour levels
    cs = ax.contour(xx, yy, Z, levels=contour_levels, colors='k')
    ax.clabel(cs, fontsize=10, inline=1)

# Ensure that class metrics are correctly computed and stored
class_metrics_map = get_binary_classification_metrics(predictions, labels_valid, N1_valid)

# Define a figure for the validation plot with decision boundaries
fig_disc_grid, ax_disc = plt.subplots(figsize=(8, 8))
ax_disc.set_title(r"Validation $D^{%d}$" % N_valid)

# Plot correct and incorrect predictions
ax_disc.plot(X_valid[class_metrics_map['TN'], 0], X_valid[class_metrics_map['TN'], 1], '^b', label="Correct Class 0")
ax_disc.plot(X_valid[class_metrics_map['FP'], 0], X_valid[class_metrics_map['FP'], 1], '^r', label="Incorrect Class 0")
ax_disc.plot(X_valid[class_metrics_map['FN'], 0], X_valid[class_metrics_map['FN'], 1], '+r', label="Incorrect Class 1")
ax_disc.plot(X_valid[class_metrics_map['TP'], 0], X_valid[class_metrics_map['TP'], 1], '+b', label="Correct Class 1")

# Set axis labels
ax_disc.set_xlabel(r"$x$")
ax_disc.set_ylabel(r"$y$")

# Call the plot function to draw decision boundaries
plot_erm_decision_boundaries(
    ax_disc,
    X_valid,
    discriminant_score_erm
)

# Set the aspect ratio and adjust axis limits to prevent skew
ax_disc.set_aspect('equal', adjustable='box')
ax_disc.legend()

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```

OUTPUT

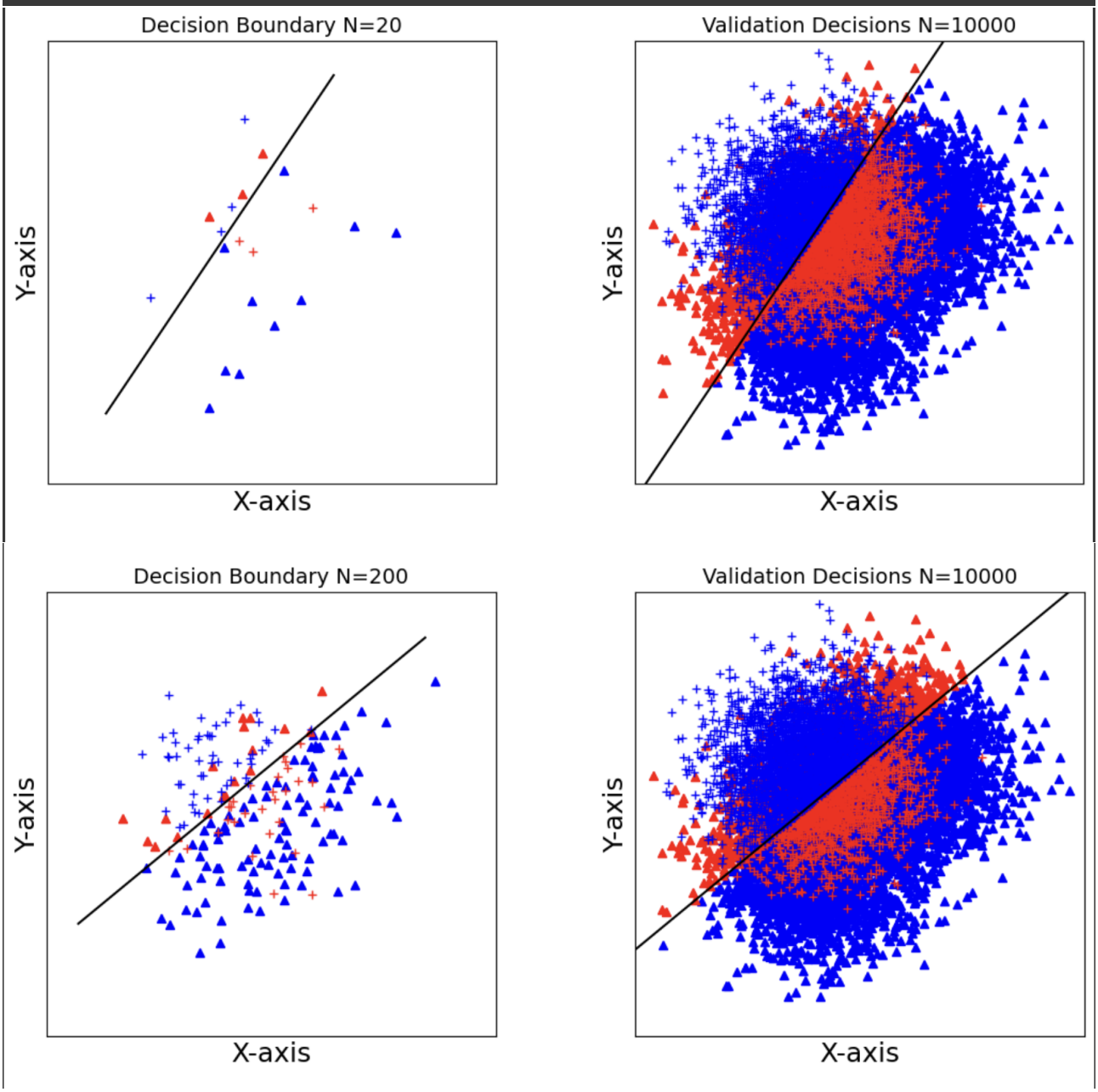


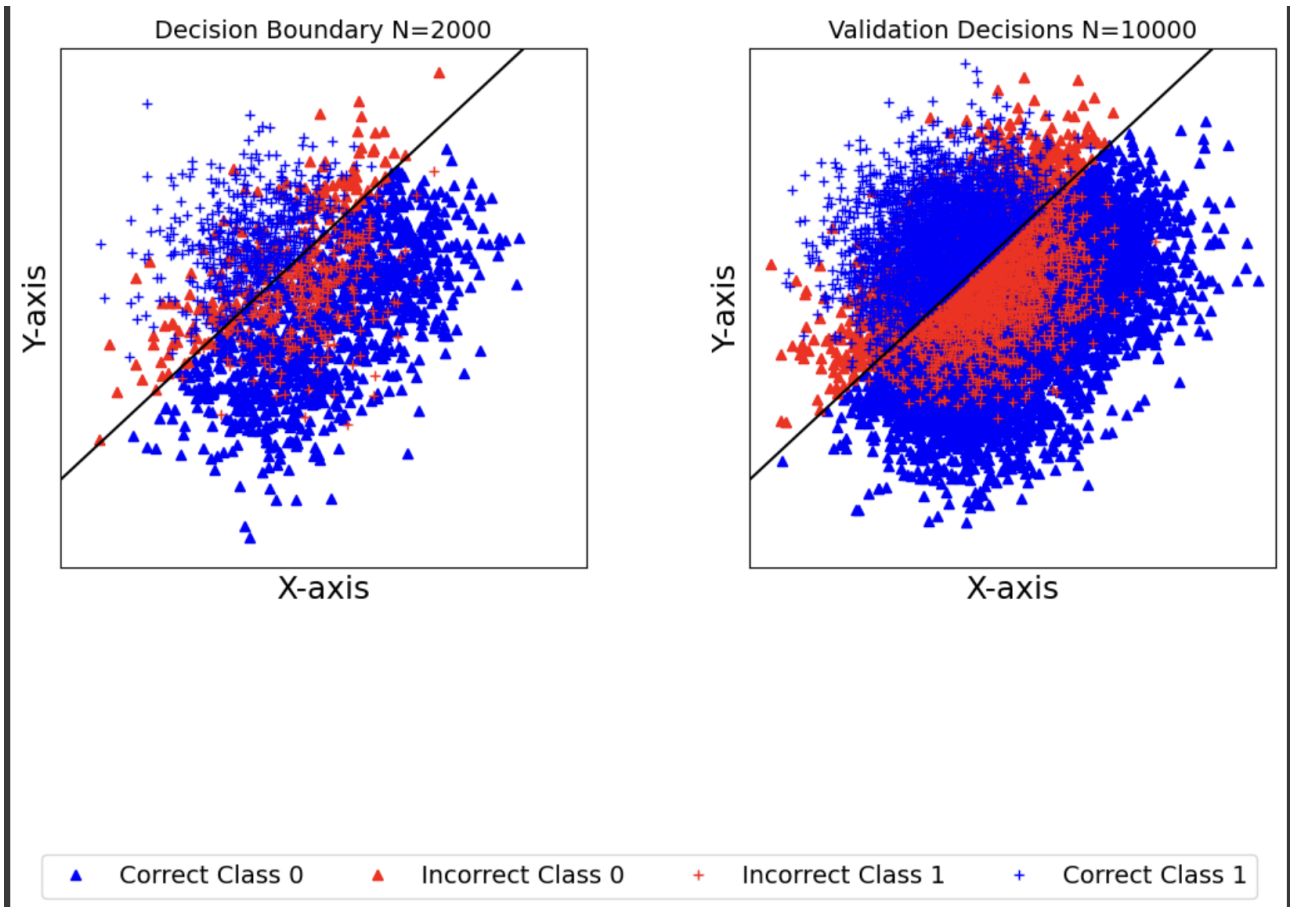
- The code generates subplots for a logistic-linear model trained on different subset sizes (`N_train`). It iterates over each subset, computes the Maximum Likelihood Estimates (MLE) for model parameters, and displays the values.
- For each subset, it plots the decision boundary on the training set, marking correct and incorrect classifications, and reports the training error.
- Additionally, it evaluates classifier performance on the validation set, showing validation errors with decision boundaries. X and Y limits are set based on validation data for consistency.
- The purpose is to compare the model's performance across varying training subset sizes, with a shared legend and layout adjustments for clarity.

```
# Calculate the number of samples per class in the validation set
N_labels_valid = np.array([sum(labels_valid == 0), sum(labels_valid == 1)])

# Create subplots with increased figure size for better visibility
fig_linear, ax_linear = plt.subplots(3, 2, figsize=(12, 18))  # Adjusted size for better spacing

# Define the phi transformation to only include a bias column (degree 1 polynomial)
phi = PolynomialFeatures(degree=1)

print("Logistic-Linear Model Optimization Results for Different Training Subsets\n")

# Iterate over different training subsets and compute MLE parameters
for i, num_samples in enumerate(N_train):
    # Compute MLE parameters for the logistic-linear model
    w_mle = compute_logistic_params(phi.fit_transform(X_train[i]), labels_train[i])

    # Print the MLE parameters for the current training subset
    print(f"Training Subset Size (N): {num_samples}; MLE for Model Parameters (w): {w_mle}")

    # Report and plot results on the training set
    training_error = report_logistic_classifier_results(
        ax_linear[i, 0], X_train[i], w_mle, labels_train[i], N_labels_train[i], phi
    )
    print(f"Training set error for N={num_samples} classifier is {training_error:.3f}")

    # Set title and remove ticks for the training set plot
    ax_linear[i, 0].set_title(f"Decision Boundary N={num_samples}", fontsize=14)
    ax_linear[i, 0].set_xticks([])
    ax_linear[i, 0].set_yticks([])

    # Report and plot results on the validation set
    prob_error = report_logistic_classifier_results(
        ax_linear[i, 1], X_valid, w_mle, labels_valid, N_labels_valid, phi
    )
    print(f"Validation set error for N={num_samples} classifier is {prob_error:.3f}\n")

    # Set title and remove ticks for the validation set plot
    ax_linear[i, 1].set_title(f"Validation Decisions N={N_valid}", fontsize=14)
    ax_linear[i, 1].set_xticks([])
    ax_linear[i, 1].set_yticks([])

# Set axis limits for all plots based on the validation set limits
plt.setp(ax_linear, xlim=x1_valid_lim, ylim=x2_valid_lim)

# Adjust layout to prevent overlap between subplots
plt.tight_layout(pad=3.0)  # Increased padding between subplots

# Add a shared legend below the plots (outside the figure)
handles, labels = ax_linear[0, 1].get_legend_handles_labels()
fig_linear.legend(handles, labels, loc='upper center', bbox_to_anchor=(0.5, -0.05), ncol=4)

plt.show()
```

OUTPUT

```
Logistic-Linear Model Optimization Results for Different Training Subsets

Training Subset Size (N): 20; MLE for Model Parameters (w): [-1.3116299  -0.91956086  0.6101539 ]
Training set error for N=20 classifier is 0.300
Validation set error for N=20 classifier is 0.245

Training Subset Size (N): 200; MLE for Model Parameters (w): [-1.14618962 -0.86736636  1.03784857]
Training set error for N=200 classifier is 0.235
Validation set error for N=200 classifier is 0.235

Training Subset Size (N): 2000; MLE for Model Parameters (w): [-1.40153152 -0.94160397  0.99618381]
Training set error for N=2000 classifier is 0.227
Validation set error for N=2000 classifier is 0.232
```





Optional: Part B

- For a logistic-quadratic model, the logistic function is defined as:

$$g(w^T\phi(x)) = \frac{1}{1 + e^{-w^T\phi(x)}}$$

  where $\phi(x) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T$. Below is a training procedure applying a quadratic feature transformation to the inputs.

- The code generates subplots to visualize the results of logistic-quadratic model optimization on different training subsets. It applies Maximum Likelihood Estimation (MLE) to calculate model parameters for subsets of varying sizes.

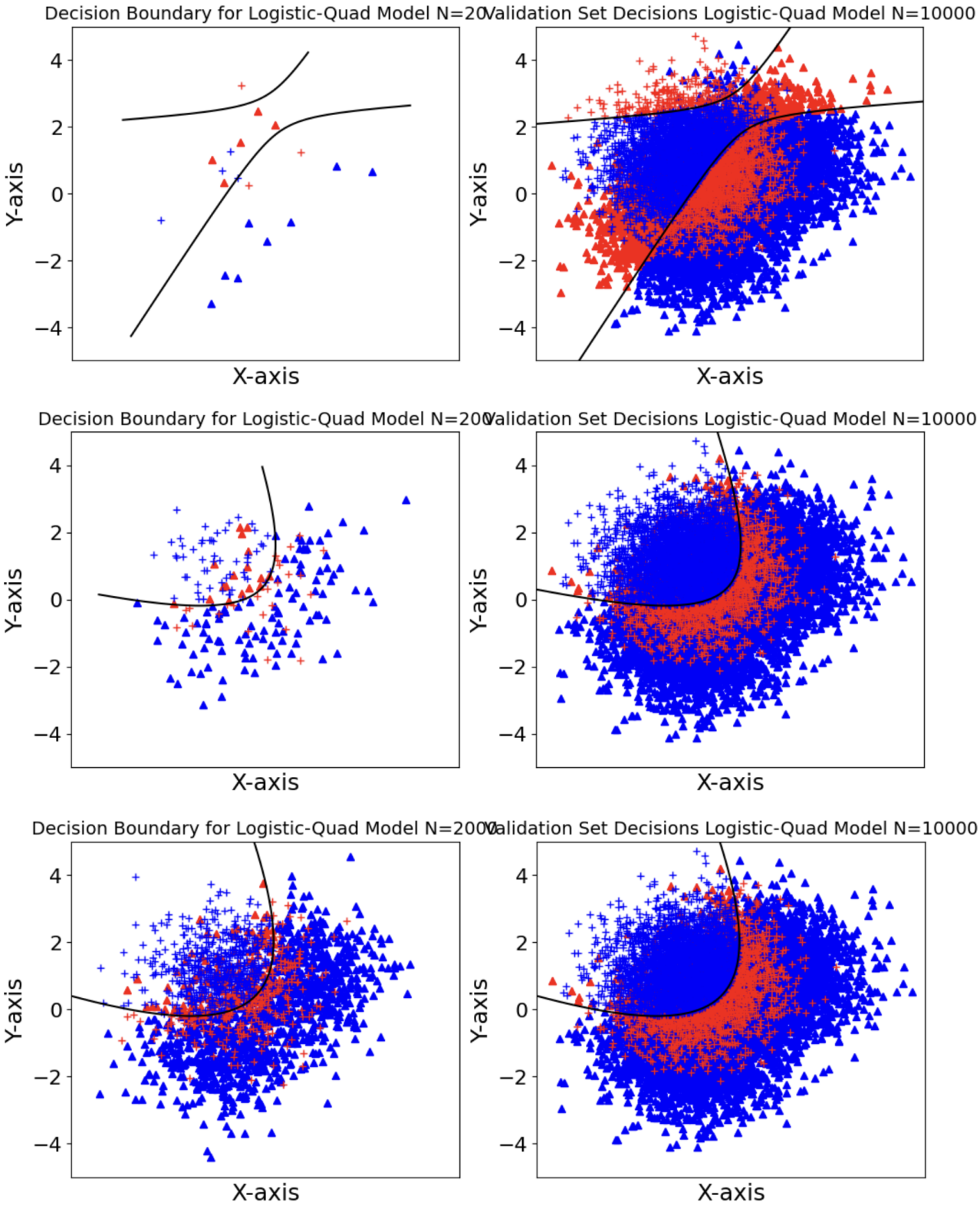
- Decision boundaries and classifier predictions for both the training and validation sets are displayed for each subset. Each plot includes details about the model size (`N`) and corresponding MLE values. Decision boundaries represent class separation, and validation set predictions give insights into model generalization.

- Legends summarize relevant information, ensuring easy interpretation. The code facilitates a comparative analysis of the logistic-quadratic model's performance across different training subset sizes.

```python
# Create subplots for logistic-quadratic model results with increased figure size
fig_quad, ax_quad = plt.subplots(3, 2, figsize=(12, 18))  # Increased size for better spacing

# Define the phi transformation for quadratic terms
phi = PolynomialFeatures(degree=2)

print("Second-order optimization of the NLL loss for a logistic-quadratic model applied to different training subsets\n")

# Iterate over different training subsets
for i, num_samples in enumerate(N_train):
    # Compute MLE for logistic-quadratic model parameters
    w_mle = compute_logistic_params(phi.fit_transform(X_train[i]), labels_train[i])

    # Print the MLE parameters
    print(f"Training Subset Size (N): {num_samples}; MLE for Model Parameters (w): {w_mle}")

    # Report and plot results for the training set
    training_error = report_logistic_classifier_results(
        ax_quad[i, 0], X_train[i], w_mle, labels_train[i], N_labels_train[i], phi
    )
    print(f"Training set error for N={num_samples} classifier is {training_error:.3f}")

    # Set the title for the training plot
    ax_quad[i, 0].set_title(f"Decision Boundary for Logistic-Quad Model N={num_samples}", fontsize=14)
    ax_quad[i, 0].set_xticks([])

    # Report and plot results for the validation set
    prob_error = report_logistic_classifier_results(
        ax_quad[i, 1], X_valid, w_mle, labels_valid, N_labels_valid, phi
    )
    print(f"Validation set error for N={num_samples} classifier is {prob_error:.3f}\n")

    # Set the title for the validation plot
    ax_quad[i, 1].set_title(f"Validation Set Decisions Logistic-Quad Model N={N_valid}", fontsize=14)
    ax_quad[i, 1].set_xticks([])

# Set consistent axis limits across all plots based on the validation set
plt.setp(ax_quad, xlim=x1_valid_lim, ylim=x2_valid_lim)

# Adjust layout to avoid overlap
plt.subplots_adjust(hspace=0.4)  # Increased spacing between plots

# Add a legend for all plots, positioned below the figure
handles, labels = ax_quad[0, 1].get_legend_handles_labels()
fig_quad.legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.05), ncol=4)

# Display the plots
plt.show()
```

OUTPUT

```
Second-order optimization of the NLL loss for a logistic-quadratic model applied to different training subsets

Training Subset Size (N): 20; MLE for Model Parameters (w): [-1.13477422 -1.16536271  1.15080551 -0.03617655  0.48455116 -0.260896  ]
Training set error for N=20 classifier is 0.400
Validation set error for N=20 classifier is 0.279

Training Subset Size (N): 200; MLE for Model Parameters (w): [-0.36386018 -0.76915305  0.91471439 -0.19117076 -0.6910404  -0.22288765]
Training set error for N=200 classifier is 0.210
Validation set error for N=200 classifier is 0.194

Training Subset Size (N): 2000; MLE for Model Parameters (w): [-0.61993337 -0.94406928  0.93077323 -0.21925547 -0.51445597 -0.20085655]
Training set error for N=2000 classifier is 0.202
Validation set error for N=2000 classifier is 0.196
```


Decision Boundary for Logistic-Quad Model N=20
Validation Set Decisions Logistic-Quad Model N=10000
Decision Boundary for Logistic-Quad Model N=200
Validation Set Decisions Logistic-Quad Model N=10000
Decision Boundary for Logistic-Quad Model N=2000
Validation Set Decisions Logistic-Quad Model N=10000
Y-axis
X-axis
4
2
0
−2
−4
Correct Class 0
Incorrect Class 0
Incorrect Class 1
Correct Class 1

Discussion

- The results provide insights into the performance of classifiers on a dataset with overlapping Gaussian distributions. For the logistic-linear regression model, a linear decision boundary is insufficient to effectively separate the classes. The model shows consistent suboptimal performance, with an average error of around 0.25 on the validation set, which is higher than the theoretical optimal classifier's error of 0.2166. This indicates that the logistic-linear model struggles to differentiate between Gaussian class-conditional distributions, regardless of the training subset size.

- In contrast, the logistic-quadratic regression model performs significantly better. With a training set of size D_{train}^{2000}, the model achieves a validation error close to the theoretically optimal classifier's error rate of 0.233. This suggests that the quadratic decision boundary aligns well with the underlying distribution, effectively capturing the nonlinearity in class separation as suggested by the Gaussian class-conditional densities.

- Additionally, the logistic-quadratic classifier demonstrates sensitivity to the training data volume. For a small training set (D_{train}^{20}), the model appears to overfit, with an error of 0.4 on the training set but a higher error of 0.279 on the validation set. As the training set size increases, the generalization gap reduces, resulting in comparable error rates for both training and validation sets with the larger dataset (D_{train}^{2000}), where errors are 0.202 and 0.196, respectively.

.

Question 2 (20%)

Assume that scalar-real y and two-dimensional real vector $\mathbf{x}$ are related to each other according to $y = c(\mathbf{x}, \mathbf{w}) + \nu$, where $c(\cdot, \mathbf{w})$ is a cubic polynomial in $\mathbf{x}$ with coefficients $\mathbf{w}$, and ν is a random Gaussian 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, and 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 them 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 $c(\cdot, \mathbf{w}_{\text{trained}})$. 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.

Answer

```
[ ]  %matplotlib inline

     import numpy as np
     import matplotlib.pyplot as plt
     from sklearn.preprocessing import PolynomialFeatures
     from mpl_toolkits.mplot3d import Axes3D
     from sklearn.preprocessing import StandardScaler
```

```
def generate_gmm_data(num_samples, gmm_pdf):
    """Generate data samples from a specified mixture of Gaussians."""
    u = np.random.random(num_samples)
    thresholds = np.cumsum(gmm_pdf['priors'])
    thresholds = np.insert(thresholds, 0, 0)

    n = gmm_pdf['mu'].shape[1]  # Data dimensionality
    samples = np.zeros((num_samples, n))

    C = len(gmm_pdf['priors'])  # Number of components

    for i in range(1, C + 1):
        indices = np.argwhere((thresholds[i - 1] <= u) & (u < thresholds[i])).flatten()
        samples[indices, :] = np.random.multivariate_normal(
            gmm_pdf['mu'][i - 1], gmm_pdf['Sigma'][i - 1], size=len(indices)
        )

    return samples[:, 0:2], samples[:, 2]
```

```
def generate_q2_data(num_samples, dataset_name):
    """Generate training or validation data using a GMM and plot the data."""
    gmm_pdf = {
        'priors': np.array([0.3, 0.4, 0.3]),
        'mu': np.array([[-10, 0, 10], [0, 0, 0], [10, 0, -10]]),
        'Sigma': np.array([
            [[3, 0, 0], [0, 2, 0], [0, 0, 2]],
            [[7, 0, 0], [0, 3, 0], [0, 0, 3]],
            [[2, 0, 0], [0, 2, 0], [0, 0, 4]]
        ])
    }

    X, y = generate_gmm_data(num_samples, gmm_pdf)

    # Plot the generated data
    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], y, marker='o', color='b', alpha=0.6)

    ax.set_xlabel(r"$x$", fontsize=12)
    ax.set_ylabel(r"$y$", fontsize=12)
    ax.set_zlabel(r"$z$", fontsize=12)
    ax.set_title(f"{dataset_name} Dataset", fontsize=15)
    plt.show()

    return X, y
```

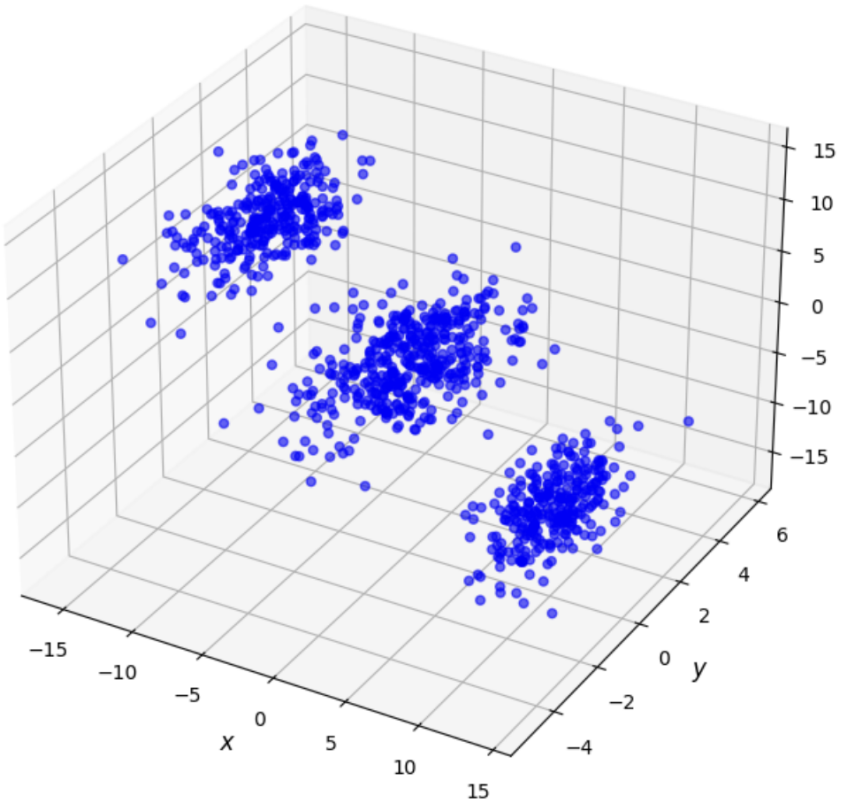
```python
def plot3(x, y, z, title):
    """
    Plot a 3D scatter plot of the generated data with an increased figure size.

    Args:
        x (np.ndarray): X-coordinates.
        y (np.ndarray): Y-coordinates.
        z (np.ndarray): Z-coordinates.
        title (str): Title of the plot.
    """
    # Create the 3D plot with a larger figure size (width, height)
    fig = plt.figure(figsize=(12, 8))  # Properly using a tuple for figsize
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x, y, z, marker='o', color='b')

    # Set labels and title with adjusted font sizes
    ax.set_xlabel('X', fontsize=12)
    ax.set_ylabel('Y', fontsize=12)
    ax.set_zlabel('Z', fontsize=12)
    ax.set_title(title, fontsize=15)

    plt.show()
```

```python
[ ] def create_prediction_score_grid(bounds_X, bounds_Y, prediction_function, phi=None, num_coords=100):
        """
        Generate a grid of prediction scores over the specified x-y bounds.
        """
        # Create a meshgrid within the specified bounds
        xx, yy = np.meshgrid(
            np.linspace(bounds_X[0], bounds_X[1], num_coords),
            np.linspace(bounds_Y[0], bounds_Y[1], num_coords)
        )
        grid = np.c_[xx.ravel(), yy.ravel()]  # Flatten the grid coordinates

        # Apply the phi transformation if provided
        if phi is not None:
            grid = phi.transform(grid)

        # Apply the prediction function to each point on the grid
        Z = np.array([prediction_function(point) for point in grid])

        # Reshape Z to match the shape of the meshgrid
        Z = Z.reshape(xx.shape)
        return xx, yy, Z
```

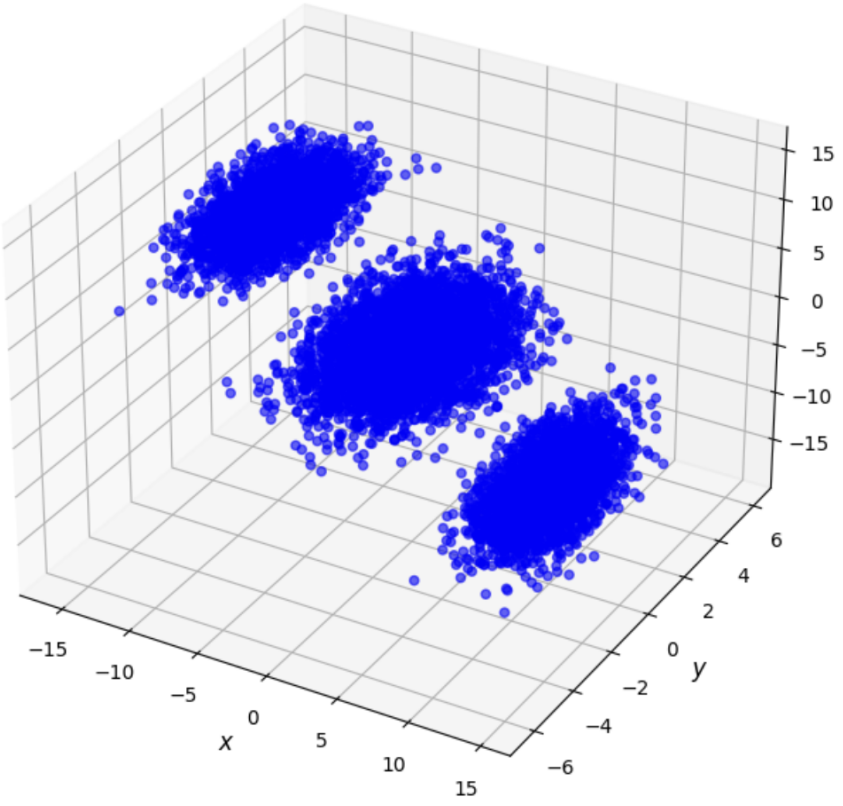
```python
# Example usage to generate training and validation datasets
X_train, y_train = generate_q2_data(1000, "Training")
X_valid, y_valid = generate_q2_data(10000, "Validation")
```

OUTPUT

Training Dataset



Validation Dataset



Maximum Likelihood Estimation (MLE)

The likelihood function for the data

$$D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$$

is given by

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

Assume

$$y_i = c(x_i, \omega) + v_i, \quad v_i \sim \text{Gaussian random scalar}$$

The likelihood function is expressed as

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

Taking the log of the likelihood function,

$$\ln(L(\omega)) = -\frac{N}{2}\ln(2\pi\sigma^2) - \sum_{i=1}^{N} \frac{(y_i - c(x_i, \omega))^2}{2\sigma^2}$$

The ML estimator $\hat{\omega}_{\text{ML}}$ is obtained by maximizing the log likelihood:

$$\hat{\omega}_{\text{ML}} = \arg\max_{\omega} \ln(L(\omega))$$

```python
def ml_solution(X, y):
    """Compute the ML parameter solution using pseudo-inverse."""
    return np.linalg.pinv(X.T @ X) @ X.T @ y
```

Maximum A Posteriori (MAP) Estimation

The MAP estimator incorporates a prior distribution for parameter ω. Given a zero-mean prior with covariance matrix $\sigma^2 I$, the posterior distribution is proportional to the product of the prior and likelihood:

$$p(\omega|D) \propto p(D|\omega)\, p(\omega)$$

$$\ln(p(\omega|D)) \propto \ln(p(D|\omega)) + \ln(p(\omega))$$

For a zero-mean Gaussian prior,

$$\ln(p(\omega)) = -\frac{1}{2\sigma^2}\omega^T\omega$$

Thus,

$$\ln(p(\omega|D)) = \ln(L(\omega)) - \frac{1}{2\sigma^2}\omega^T\omega$$

The MAP estimator $\hat{\omega}_{\text{MAP}}$ is obtained by maximizing the log-posterior:

$$\hat{\omega}_{\text{MAP}} = \arg\max_{\omega} \ln(p(\omega|D))$$

```python
def map_solution(X, y, gamma=0.1):
    """Compute the MAP parameter solution with regularization."""
    I = np.eye(X.shape[1])
    return np.linalg.pinv(X.T @ X + gamma * I) @ X.T @ y
```

```python
def mean_squared_error(y_preds, y_true):
    """Compute the Mean Squared Error (MSE)."""
    error = y_preds - y_true
    return np.mean(error ** 2)
```

ML Estimator Solution

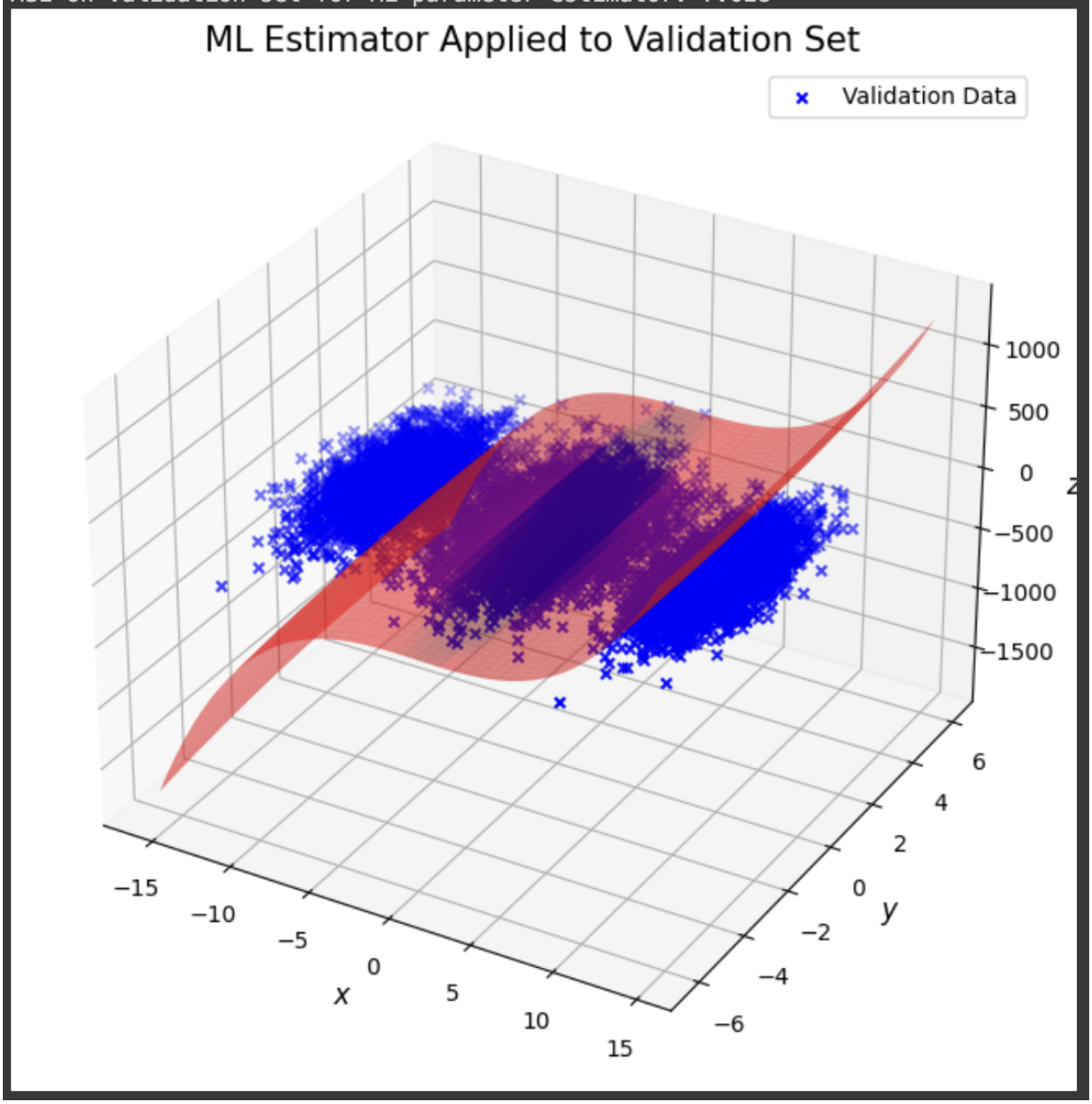
- The cubic transformation is applied to the training dataset ($X_{\text{train\_cubic}}$) using the `PolynomialFeature` class from scikit-learn with a degree of 3. The Maximum Likelihood Estimate (MLE) for the parameters (θ_{mle}) of the cubic model is calculated via the `ml_solution` function. This model is then utilized to make predictions on the validation dataset ($X_{\text{valid\_cubic}}$), resulting in predicted values ($y_{\text{pred\_mle}}$). The Mean Squared Error (MSE) between the predicted and actual values on the validation set is computed and displayed as a metric for evaluation.

- A 3D plot is generated to show the fitted plane on the validation dataset. The blue 'x' markers represent validation samples, while the red transparent surface indicates the fitted cubic plane. This visualization helps to assess how accurately the cubic model fits the data in a three-dimensional space. The axes are labeled, and the aspect ratio of the 3D plot is adjusted to match the actual ranges of the features and target variable. This plot offers an intuitive view of the cubic model's performance on the validation dataset.

```python
# Apply scaling to the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

# Apply cubic transformation to scaled data
phi = PolynomialFeatures(degree=3)
X_train_cubic = phi.fit_transform(X_train_scaled)
X_valid_cubic = phi.transform(X_valid_scaled)

# Derive the ML parameter solution
theta_mle = ml_solution(X_train_cubic, y_train)

# Predict and calculate MSE for ML estimator
y_pred_mle = X_valid_cubic @ theta_mle
mse_mle = mean_squared_error(y_pred_mle, y_valid)
print(f"MSE on Validation set for ML parameter estimator: {mse_mle:.3f}")

# 3D plot for ML estimator
fig_mle = plt.figure(figsize=(12, 8))
ax_mle = fig_mle.add_subplot(111, projection='3d')

ax_mle.scatter(X_valid[:, 0], X_valid[:, 1], y_valid, color='b', marker='x', label='Validation Data')

xx, yy = np.meshgrid(
    np.linspace(X_valid[:, 0].min(), X_valid[:, 0].max(), 100),
    np.linspace(X_valid[:, 1].min(), X_valid[:, 1].max(), 100)
)
Z = phi.transform(np.c_[xx.ravel(), yy.ravel()]) @ theta_mle
Z = Z.reshape(xx.shape)

ax_mle.plot_surface(xx, yy, Z, color='red', alpha=0.5)
ax_mle.set_xlabel(r"$x$", fontsize=12)
ax_mle.set_ylabel(r"$y$", fontsize=12)
ax_mle.set_zlabel(r"$z$", fontsize=12)
ax_mle.set_title("ML Estimator Applied to Validation Set", fontsize=15)
ax_mle.legend()
plt.show()
```

OUTPUT

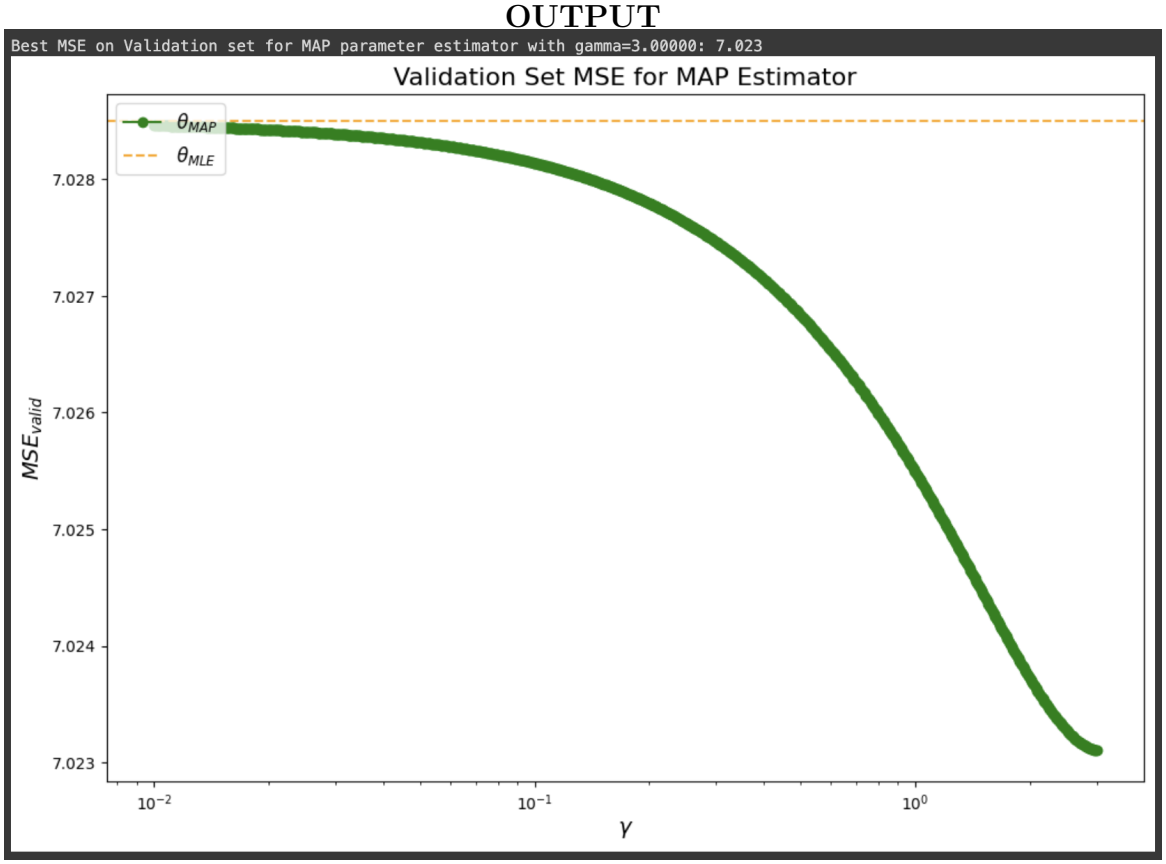
```
MSE on Validation set for ML parameter estimator: 7.028
```



Now repeat the same process for the MAP estimator

```python
# Define gamma values for MAP estimation
num_gammas = 1000
gammas = np.geomspace(1e-2, 3, num=num_gammas)
mse_values_map = np.empty(num_gammas)

# Compute MSE for each gamma
for i, gamma in enumerate(gammas):
    theta_map = map_solution(X_train_cubic, y_train, gamma)
    y_pred_map = X_valid_cubic @ theta_map
    mse_values_map[i] = mean_squared_error(y_pred_map, y_valid)

# Find the best gamma value and corresponding MSE
best_gamma_map = gammas[np.argmin(mse_values_map)]
best_mse_map = np.min(mse_values_map)
print(f"Best MSE on Validation set for MAP parameter estimator with gamma={best_gamma_map:.5f}: {best_mse_map:.3f}")

# Plot MSE vs Gamma
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(gammas, mse_values_map, color='green', label=r"$\theta_{MAP}$", marker='o')
ax.axhline(y=mse_mle, color='orange', linestyle='--', label=r"$\theta_{MLE}$")
ax.set_xscale('log')
ax.set_xlabel(r"$\gamma$", fontsize=14)
ax.set_ylabel(r"$MSE_{valid}$", fontsize=14)
ax.set_title("Validation Set MSE for MAP Estimator", fontsize=16)
ax.legend(loc='upper left', fontsize=12)
plt.show()
```

OUTPUT



- Applying a MAP-trained linear regression model with a zero-mean Gaussian prior on the weights, often termed ridge regression, offers a way to control overfitting issues commonly seen with Maximum Likelihood Estimation (MLE), especially in cases with limited data. In this context, the MAP estimator includes an additional regularization term in the NLL objective, which penalizes large weight values. This regularization term is managed by the parameter γ (or equivalently, $\lambda = 1/\gamma$), and it effectively prevents the weights from reaching overly complex values. The NLL objective function, with regularization, is given as follows:

$$\text{PNLL}(\theta) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - w^T x^{(i)})^2 + \lambda \sum_{j=1}^{n} w_j^2$$

  or equivalently,

$$= \frac{1}{2}(Xw - y)^T(Xw - y) + \lambda w^T w,$$

  where $\lambda = 1/\gamma$ determines the strength of regularization.

- In terms of performance, the MAP estimator shows an advantage over the MLE when an appropriate γ value is chosen. In this experiment, the MLE estimator achieves an MSE of 7.028 on the validation set, while the MAP estimator reaches a slightly better MSE of 7.023 with an optimal $\gamma = 3.000$.

- The MSE vs. γ plot highlights the relationship between the MLE and MAP estimators. As γ grows larger (e.g., $\gamma > 1$), the MAP estimator's performance approaches that of the MLE. This is because, at high γ values, the regularization effect decreases, allowing the MAP estimator to behave similarly to MLE. However, if γ is too small (e.g., $\gamma < 10^{-1}$), the regularization becomes too strong, causing a sharp increase in MSE due to the model overly conforming to the prior and not fitting the data well.

- Although this experiment finds a suitable γ value, in practical situations where validation data may be limited, cross-validation is a valuable alternative for tuning γ. Cross-validation ensures balanced regularization strength, optimizing generalization without sacrificing training set performance.

Question 3 (20%)

A vehicle at true position $[x_T, y_T]^T$ in 2-dimensional space is to be localized using distance (range) measurements to K reference (landmark) coordinates $\{[x_1, y_1]^T, \ldots, [x_i, y_i]^T, \ldots, [x_K, y_K]^T\}$. These range measurements are $r_i = d_{Ti} + n_i \quad \text{for} \quad i \in \{1, \ldots, K\}$, where $d_{Ti} = \|[x_T, y_T]^T - [x_i, y_i]^T\|$ is the true distance between the vehicle and the ith 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:

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

where $[x, y]^T$ 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 $[x_{\text{MAP}}, y_{\text{MAP}}]^T$ 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 radius 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? Does it 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.

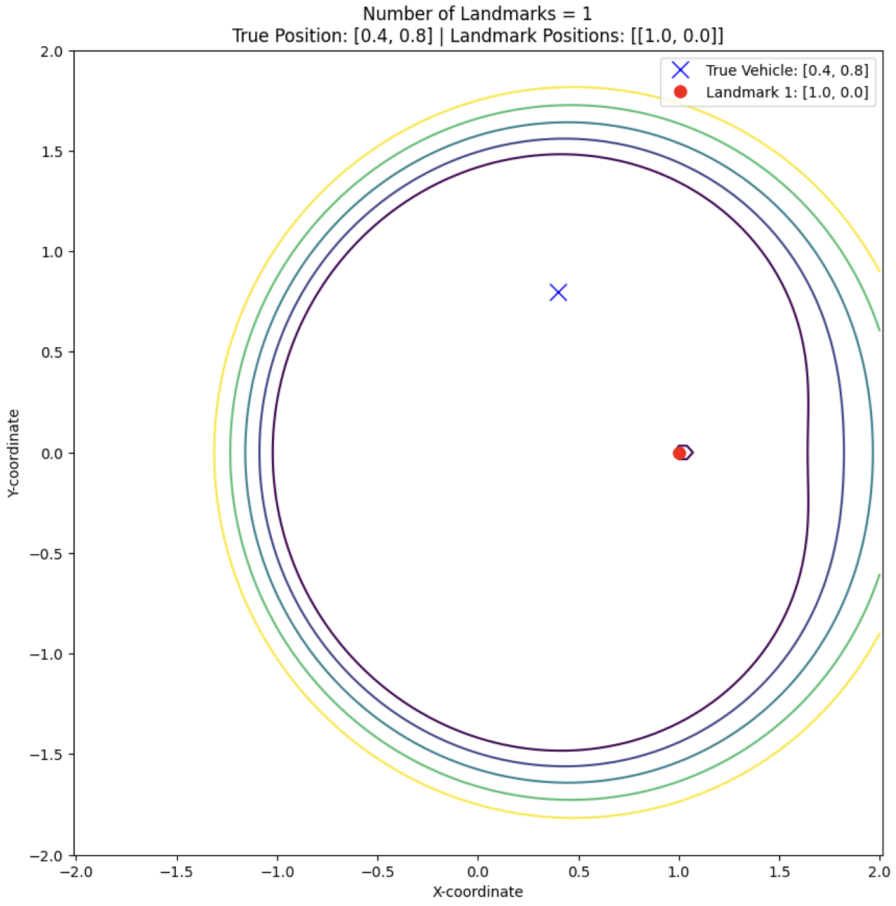
Answer

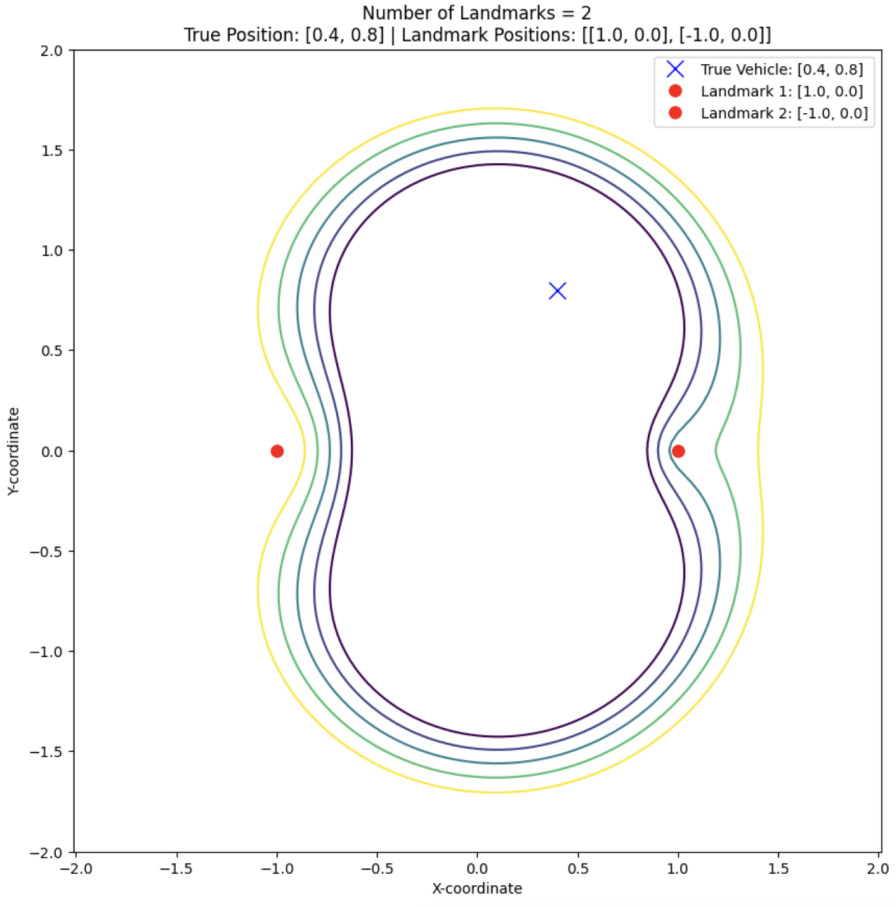
- The code creates and displays the contours of an objective function used in a Maximum A Posteriori (MAP) estimation task for determining a vehicle's position in a 2D space. The true location of the vehicle is set at $[0.4, 0.8]$. A range measurement model is implemented using the function `generate_range_measurement(landmark_position)`, which produces a noisy range measurement from a specified landmark position to the vehicle's actual position.
- The objective function for MAP estimation, defined as `objective_function_estimate_position`, includes both a prior term based on the vehicle's location and an error term that accounts for the discrepancy between the measured ranges and the distances from landmarks to the estimated position. The objective value is calculated as the sum of these terms.
- The code then plots the contours of the objective function under various scenarios with different numbers of landmarks. In each scenario, landmarks are positioned on a circle, and valid range measurements are generated accordingly. The contours of the objective function are visualized on a logarithmic scale, with the actual vehicle position indicated by a blue 'x' and the landmark positions by red circles.
- This visualization aids in understanding how the objective function varies with different landmark placements, offering insights into the optimization landscape for the MAP estimator in vehicle localization contexts.

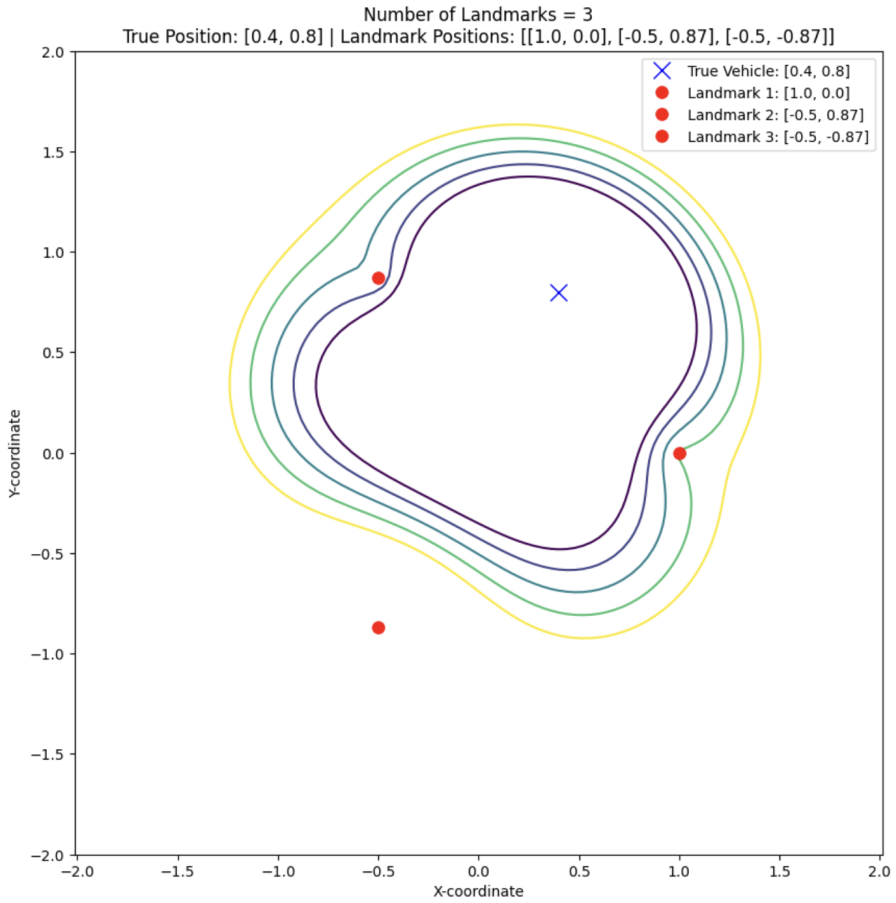
```python
import matplotlib.pyplot as plt
import numpy as np

# True position of the vehicle inside the unit circle
true_position = np.array([0.4, 0.8])

# Measurement noise standard deviation
measurement_noise_range = 0.3

# Generate a noisy range measurement to a landmark
def generate_range_measurement(landmark_position):
    measured_range = (
        np.sqrt(np.sum((landmark_position - true_position) ** 2)) +
        np.random.normal(0, measurement_noise_range)
    )
    return measured_range

# Compute the objective function for MAP estimation
def objective_function_estimate_position(position, landmark_positions, measured_ranges):
    prior_term = np.sum((position - np.array([0, 0])) ** 2) / 2
    error_term = np.sum(
        (measured_ranges - np.sqrt(np.sum((landmark_positions - position) ** 2, axis=1))) ** 2
    )
    objective_value = prior_term + error_term
    return objective_value

# Sample a valid range measurement, ensuring non-negative values
def sample_valid_range(landmark_position):
    while True:
        sample = generate_range_measurement(landmark_position)
        if sample >= 0:
            return sample
```

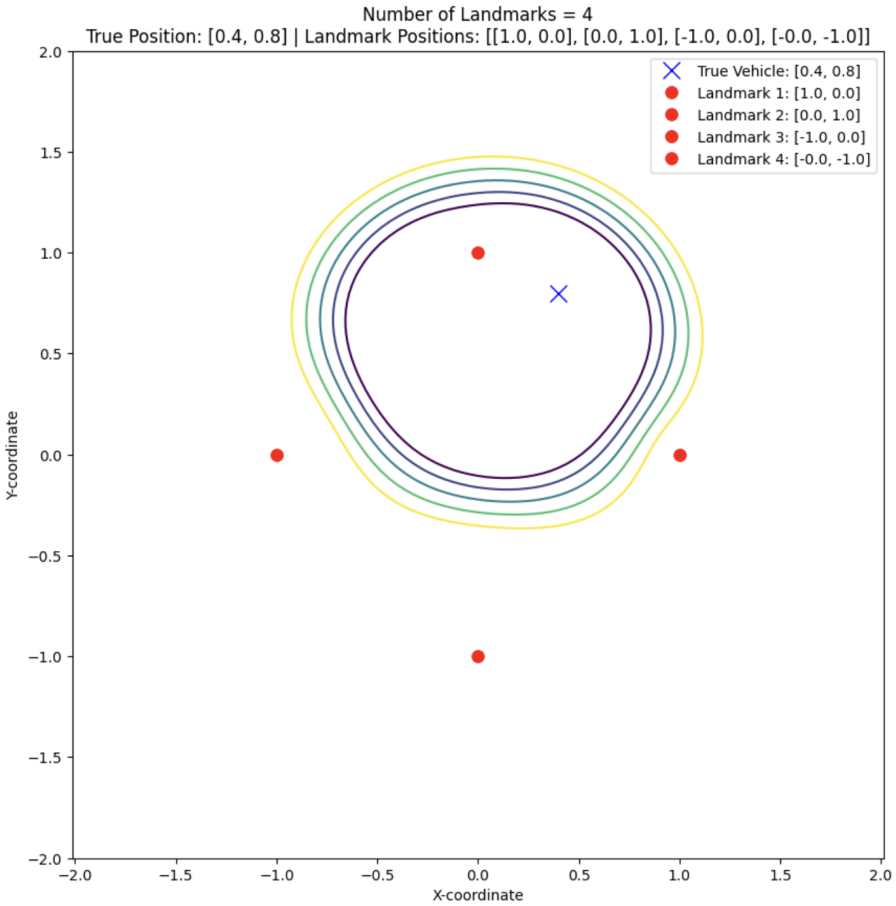
```python
# Plot contours of the MAP objective function with rounded coordinates
def plot_objective_function_contour_modified(num_landmarks):
    """Plots contours of the MAP objective function with annotations for rounded coordinates."""

    # Place landmarks on a circle
    theta_values = np.linspace(0, 2 * np.pi, num_landmarks + 1)[:-1]
    landmark_positions = np.array([np.cos(theta_values), np.sin(theta_values)]).T
    measured_ranges = np.array([sample_valid_range(landmark) for landmark in landmark_positions])

    # Round coordinates for cleaner display
    rounded_landmarks = np.round(landmark_positions, decimals=2).tolist()
    rounded_true_position = np.round(true_position, decimals=2).tolist()

    # Define ranges for coordinates
    x_range = y_range = np.linspace(-2, 2, 101)
    X, Y = np.meshgrid(x_range, y_range)
    Z = np.zeros_like(X)

    # Compute the objective function for each grid point
    for i in range(len(x_range)):
        for j in range(len(y_range)):
            position = np.array([x_range[i], y_range[j]])
            Z[j, i] = objective_function_estimate_position(position, landmark_positions, measured_ranges)

    # Plot contours
    plt.figure(figsize=(10, 10))
    levels = np.logspace(np.log10(3 / 2), np.log10(5 / 2), 5)
    plt.contour(X, Y, Z, levels=levels)

    # Plot true vehicle position and landmark positions with rounded annotations
    plt.plot(true_position[0], true_position[1], 'bx', markersize=12, label=f'True Vehicle: {rounded_true_position}')
    for idx, landmark in enumerate(rounded_landmarks):
        plt.plot(landmark[0], landmark[1], 'ro', markersize=8, label=f'Landmark {idx + 1}: {landmark}')

    # Set axis properties and labels
    plt.axis('equal')
    plt.legend()
    plt.title(f'Number of Landmarks = {num_landmarks}\n'
              f'True Position: {rounded_true_position} | Landmark Positions: {rounded_landmarks}')
    plt.xlabel('X-coordinate')
    plt.ylabel('Y-coordinate')
    plt.show()

# Generate the modified contour plots for K = 1 to 4
for num_landmarks in range(1, 5):
    plot_objective_function_contour_modified(num_landmarks)
```

OUTPUT


Number of Landmarks = 1
True Position: [0.4, 0.8] | Landmark Positions: [[1.0, 0.0]]
True Vehicle: [0.4, 0.8]
Landmark 1: [1.0, 0.0]
Y-coordinate
X-coordinate


Number of Landmarks = 2
True Position: [0.4, 0.8] | Landmark Positions: [[1.0, 0.0], [-1.0, 0.0]]
True Vehicle: [0.4, 0.8]
Landmark 1: [1.0, 0.0]
Landmark 2: [-1.0, 0.0]
Y-coordinate
X-coordinate


Number of Landmarks = 3
True Position: [0.4, 0.8] | Landmark Positions: [[1.0, 0.0], [-0.5, 0.87], [-0.5, -0.87]]
True Vehicle: [0.4, 0.8]
Landmark 1: [1.0, 0.0]
Landmark 2: [-0.5, 0.87]
Landmark 3: [-0.5, -0.87]
Y-coordinate
X-coordinate


Number of Landmarks = 4
True Position: [0.4, 0.8] | Landmark Positions: [[1.0, 0.0], [0.0, 1.0], [-1.0, 0.0], [-0.0, -1.0]]
True Vehicle: [0.4, 0.8]
Landmark 1: [1.0, 0.0]
Landmark 2: [0.0, 1.0]
Landmark 3: [-1.0, 0.0]
Landmark 4: [-0.0, -1.0]
Y-coordinate
X-coordinate

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 rejection is not too high, rejection may be a desirable action. Let

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

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 $P(\omega_i \mid x) \geq P(\omega_j \mid x)$ for all j and if $P(\omega_i \mid x) \geq 1 - \frac{\lambda_r}{\lambda_s}$, and reject otherwise. What happens if $\lambda_r = 0$? What happens if $\lambda_r > \lambda_s$?

Answer

- $$\lambda(a_i|\omega_j) = \begin{cases} 0 & \text{if } i = j \\ \lambda_r & \text{if } i = c+1 \\ \lambda_s & \text{otherwise} \end{cases}$$

Decision\Truth	1	2	3	$\cdots$	n
1	0	λ_s	λ_s	$\cdots$	λ_s
2	λ_s	0	λ_s	$\cdots$	λ_s
3	λ_s	λ_s	0	$\cdots$	λ_s
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	λ_s	λ_s	λ_s	$\cdots$	0
$n+1$	λ_r	λ_r	λ_r	$\cdots$	λ_r

- The decision rule for minimum risk in pattern classification problems with the option of rejection is based on minimizing the expected risk. The risk of making a decision a_i when the true state is ω_j is given by $\lambda(a_i|\omega_j)$.
  Let $R(a_i|x)$ denote the risk of deciding a_i given the observation x.

- The decision rule that minimizes the expected risk is to choose the action that minimizes the conditional risk $R(\alpha_i \mid x)$ for each i.
  Moreover, it is optimal to reject the pattern if the risk of rejection is less than the risk of choosing any of the c classes.

- The risk of choosing a_i when the true state is ω_j is given by:

$$R(a_i|\omega_j) = \lambda(a_i|\omega_j)P(\omega_j|x)$$

- In general, to minimize risk:

$$R(a_i|x) = \sum_{j=1}^{c} \lambda_{ij} P(\omega_j|x)$$

  where $\lambda_{ij} = \text{cost}(i, j)$ and $P(\omega_j|x)$ is the posterior of label j.

- The risk associated with deciding class i is:

$$R(a_i|x) = \lambda_s \left[\sum_{\substack{j=1 \\ j \neq i}}^{c} P(\omega_j|x) \right] \quad (1)$$

But:

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

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

Placing the value in equation (1):

$$R(a_i|x) = \lambda_s \left[1 - P(\omega_i|x)\right] \quad (2)$$

- Risk associated with deciding class i:

$$R(a_i|x) = \lambda_s \left[1 - P(\omega_i|x)\right] \quad (3)$$

- To decide between class i and class k, we will decide class i if:

$$R(a_i|x) \leq R(a_k|x)$$

Thus:

$$\lambda_s \left[1 - P(\omega_i|x)\right] \leq \lambda_s \left[1 - P(\omega_k|x)\right]$$

Thus:

$$P(\omega_i|x) \geq P(\omega_k|x) \quad \text{for all } k \neq i$$

- The risk associated with the reject class is:

$$R(a_{c+1}|x) = \sum_{j=1}^{c} \lambda_{rj} P(\omega_j|x) = \lambda_r \sum_{j=1}^{c} P(\omega_j|x) = \lambda_r$$

- The decision rule will be minimizing risk associated with decisions.
  We will decide class i if:

$$R(a_i|x) \leq R(a_{c+1}|x)$$

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

Thus:

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

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

- If:

$$P(\omega_i|x) < 1 - \frac{\lambda_r}{\lambda_s}, \quad \text{choose reject class } a_{c+1}$$

- If $\lambda_r = 0$:

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

$$P(\omega_i|x) = 1 \quad \text{(Decision rule will always select reject class)}$$

The risk of rejection becomes zero, and therefore, choose the class with the maximum posterior probability.

- If:

$$\lambda_s \geq \lambda_r$$

$$1 - \frac{\lambda_r}{\lambda_s} < 0$$

Thus:

$$P(\omega_i|x) \geq 1 - \frac{\lambda_r}{\lambda_s} < 0$$

Therefore, the decision rule will always choose some class i and never select the reject class.

Question 5 (20%)

Let Z be drawn from a categorical distribution (takes discrete values) with K possible outcomes/states and parameter $\boldsymbol{\Theta}$, represented by $Cat(\boldsymbol{\Theta})$. Describe the value/state using a 1-of-K scheme for $\mathbf{z} = [z_1, \ldots, z_K]^T$ where $z_k = 1$ if the variable is in state k and $z_k = 0$ otherwise. Let the parameter vector for the pdf be $\boldsymbol{\Theta} = [\theta_1, \ldots, \theta_K]^T$, where $P(z_k = 1) = \theta_k$, for $k \in \{1, \ldots, K\}$.
Given $\mathcal{D} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ with i.i.d. samples $\mathbf{z}_n \sim Cat(\boldsymbol{\Theta})$ for $n \in \{1, \ldots, N\}$:

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

Hint: The Dirichlet distribution with parameter $\boldsymbol{\alpha}$ is given by

$$p(\boldsymbol{\Theta} \mid \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1},$$

where the normalization constant is

$$B(\boldsymbol{\alpha}) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}.$$

Answer

- Let's consider the problem of estimating the parameter vector θ for the categorical distribution Cat(θ) based on the given data:

$$D = \{z_1, z_2, \ldots, z_N\}$$

where each z_n is an independent and identically distributed sample from the categorical distribution.

Maximum Likelihood (ML) Estimation

- The probability mass function (PMF) of the categorical distribution is given by:

$$P(z_n = k|\theta) = \theta_k$$

  where z_n is the observed category for the n-th sample, and $\theta = [\theta_1, \theta_2, \ldots, \theta_K]$ is the parameter vector representing the probabilities of each category.

- The likelihood function for the entire dataset is the product of the probabilities for each sample:

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

- To find the ML estimator $\hat{\theta}_{ML}$, we maximize the likelihood function with respect to θ. Taking the logarithm simplifies the product to a sum:

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

- Maximizing this log-likelihood is subject to the constraint:

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

- This can be done using Lagrange multipliers or other optimization techniques. The ML estimator is given by:

$$\hat{\theta}_{ML} = \frac{1}{N} \sum_{n=1}^{N} z_n$$

Maximum A Posteriori (MAP) Estimation

- Let's introduce a prior distribution on θ. The given prior is a Dirichlet distribution with hyperparameters $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_K]$:

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

- The posterior distribution is proportional to the product of the likelihood and the prior:

$$p(\theta|D, \alpha) \propto L(\theta|D) \times p(\theta|\alpha)$$

- Taking the logarithm and dropping the constant term, it becomes:

$$\log p(\theta|D, \alpha) \propto \log L(\theta|D) + \sum_{k=1}^{K} (\alpha_k - 1) \log(\theta_k)$$

- Maximizing this log-posterior subject to the constraint $\sum_{k=1}^{K} \theta_k = 1$ yields the MAP estimator:

$$\hat{\theta}_{MAP} = \frac{\sum_{n=1}^{N} z_n + \alpha_k - 1}{N + \sum_{k=1}^{K} (\alpha_k - 1)}$$