EECE5644 Assignment 2

Mariona Jaramillo Civill

October 29, 2024

NUID: 002413201 - jaramillocivill.m@northeastern.edu

Problem 1

First, the independent datasets D^{20}_{train}, D^{200}_{train}, D^{2000}_{train}, $D^{10K}_{\text{validate}}$ have been generated according to the provided data distribution. Its data distribution is shown in Figure 1

Part 1

The theoretically optimal classifier that achieves minimum probability of error using the knowledge of the true pdf can be expressed in the form of a likelihood-ratio test where:

$$\text{Decide } L_1 \text{ if } \frac{g(x|m_1, C_1)}{g(x|m_0, C_0)} \geq \frac{P(L_0)}{P(L_1)} = \gamma, \quad \text{otherwise decide } L_0$$

where $P(L_0) = 0.6$ and $P(L_1) = 0.4$.

The decision results resulting from a discriminant score for this classifier, together with its labels, have been used to be able to plot the ROC curve. On top of the ROC curve, the points linked to the classifiers that achieve the minimum probability of error both theoretically and empirically have been plotted in Figure 2.

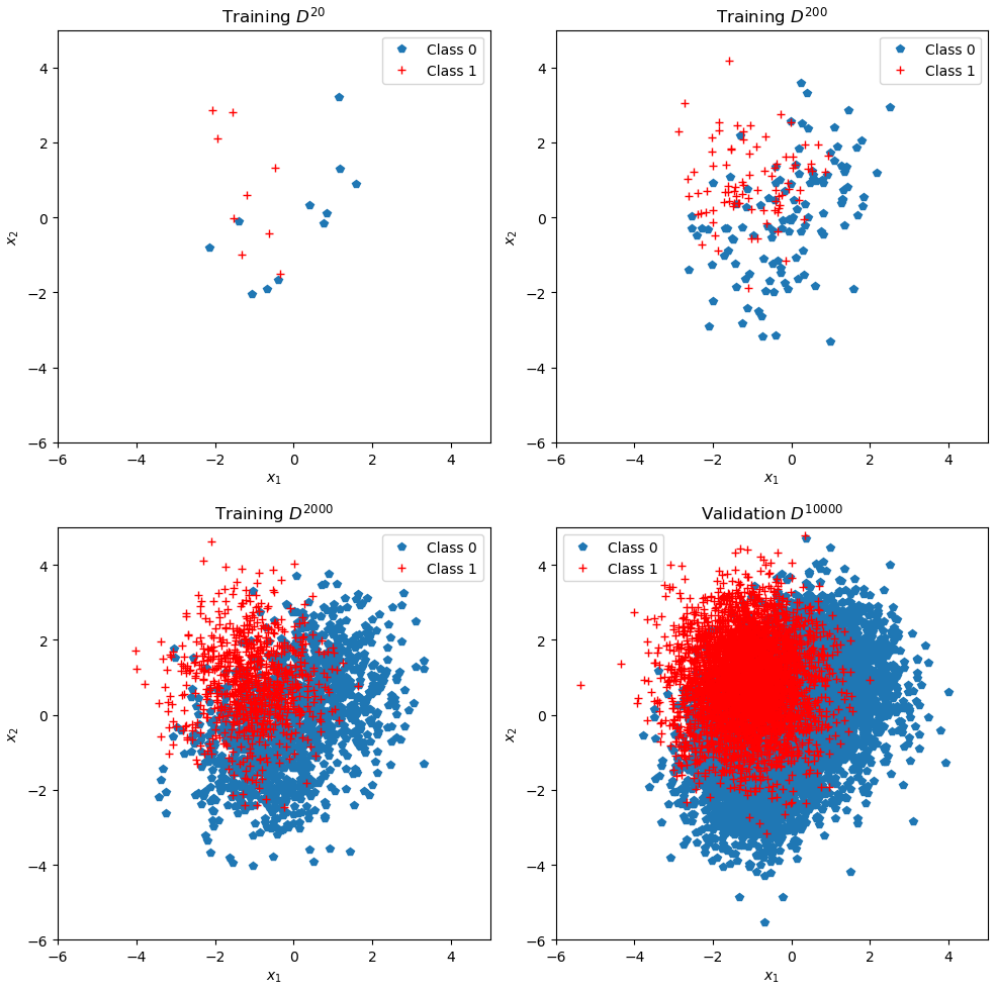

Figure 1: Distribution of the data generated

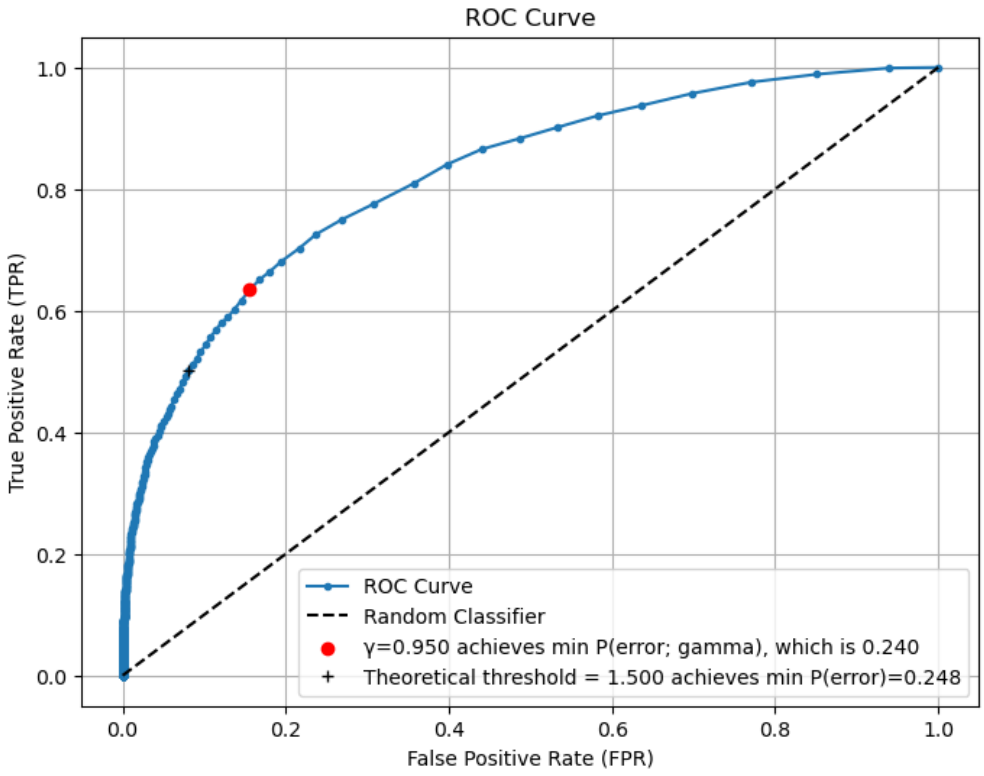

Figure 2: Approximation of the ROC curve of the minimum probability of error classifier

Part 2

The objective is to approximate class label posterior functions, given a sample, by training three separate logistic regression models on three different training datasets of varying sizes: D_{20}, D_{200}, and D_{2000}. The goal is to estimate the parameters of the logistic models using the maximum likelihood estimation (MLE) technique and subsequently apply these models to binary classify samples from a validation dataset D_{10K}, estimating the classification error.

In logistic regression, the posterior probability of class $y = 1$ given a sample $\mathbf{x}$ is modeled by the logistic function:

$$P(y = 1 \mid \mathbf{x}; \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x})}$$

where $\mathbf{w}$ represents the parameters (weights) to be estimated. These parameters are learned by maximizing the likelihood of the observed data, which is equivalent to minimizing the negative log-likelihood (NLL).

Assuming we have N independent samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $y_i \in \{0, 1\}$, the likelihood of the observed data is:

$$L(\mathbf{w}) = \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i; \mathbf{w}) = \prod_{i=1}^{N} \sigma(\mathbf{w}^\top \mathbf{x}_i)^{y_i} \cdot (1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))^{1-y_i}$$

Then, the NLL is given by:

$$\mathcal{L}_{\text{NLL}}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log(\sigma(\mathbf{w}^\top \mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)) \right]$$

where $\sigma(\mathbf{w}^\top \mathbf{x}_i)$ is the posterior probability estimated by the model for sample i, and y_i is the true class label.

Once the parameters $\mathbf{w}$ have been estimated through MLE, the logistic regression model outputs the posterior probabilities for each sample. To classify a sample $\mathbf{x}$, the decision rule is based on the posterior probability:

$$\text{Classify as } y = 1 \text{ if } P(y = 1 \mid \mathbf{x}; \mathbf{w}) > 0.5$$

otherwise, the sample is classified as $y = 0$. This decision rule approximates the minimum probability of error classification rule, which aims to minimize the number of incorrect classifications on a given dataset.

This minimization has been performed for the three training datasets using the quasi Newton method BFGS and the probability of error has been estimated for every case on the generated validation set. The probability of error is

$$P(\text{error}; \gamma) = P(D = 1 | L = 0; \gamma) P(L = 0) + P(D = 0 | L = 1; \gamma) P(L = 1)$$

The probability of error for the validation set under the different models trained is:

1. Probability of error on training dataset using the model trained with the training dataset D^{20}_{train}: 0.2626
2. Probability of error on validation dataset using the model trained with the training dataset D^{20}_{train}: 0.3214
3. Probability of error on training dataset using the model trained with the training dataset D^{200}_{train}: 0.3324
4. Probability of error on validation dataset using the model trained with the training dataset D^{200}_{train}: 0.3145
5. Probability of error on training dataset using the model trained with the training dataset D^{2000}_{train}: 0.3096
6. Probability of error on validation dataset using the model trained with the training dataset D^{2000}_{train}: 0.3143

Also, a plot to visualize the decision boundaries of these trained classifiers on the respective training and validation sets is shown on Figure 3 and Figure 4

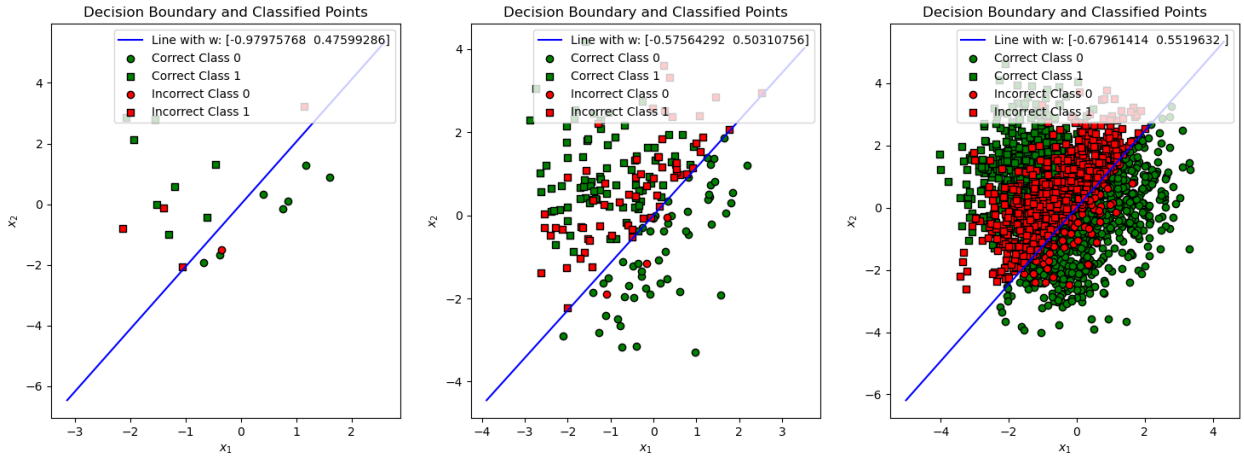

Figure 3: Plot to visualize the decision boundaries on the training datasets for each linear model

Additionally, the process has been repeated using a logistic-quadratic-function-based approximation of class label posterior functions, incorporating quadratic terms into the decision rule.

In quadratic logistic regression, terms for the squared values of the predictors and possibly interaction terms between predictors are added:

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_k X_k + \gamma_1 X_1^2 + \gamma_2 X_2^2 + ... + \gamma_k X_k^2 + \delta_{12} X_1 X_2 + ...)}}$$

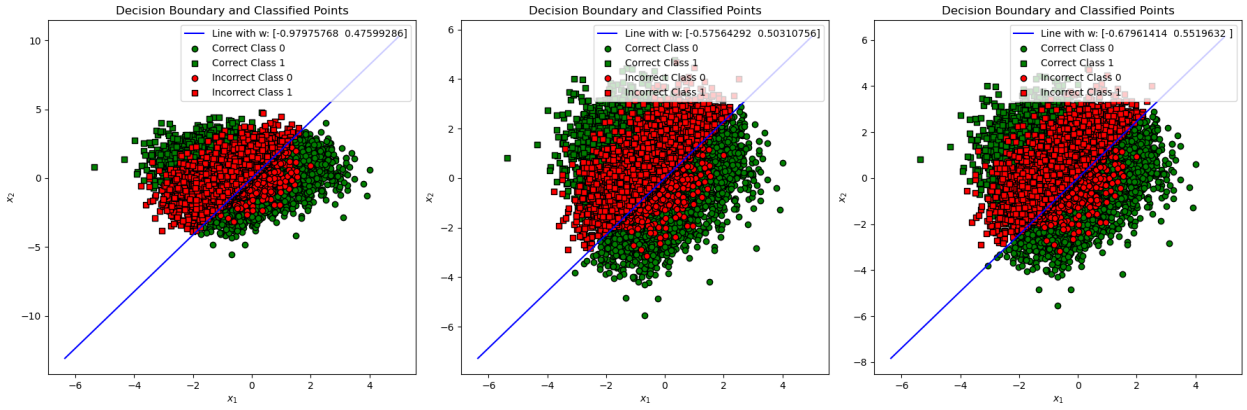

Figure 4: Plot to visualize the decision boundaries on the validation datasets for each linear model

where:

- γ_i are the coefficients for the squared terms X_i^2,
- δ_{ij} are the coefficients for the interaction terms $X_i \times X_j$.

The result has been:

1. Probability of error on training dataset using the model trained with the training dataset D_{train}^{20}: 0.1434
2. Probability of error on validation dataset using the model trained with the training dataset D_{train}^{20}: 0.2665
3. Probability of error on training dataset using the model trained with the training dataset D_{train}^{200}: 0.2275
4. Probability of error on validation dataset using the model trained with the training dataset D_{train}^{200}: 0.2081
5. Probability of error on training dataset using the model trained with the training dataset D_{train}^{2000}: 0.2043
6. Probability of error on validation dataset using the model trained with the training dataset D_{train}^{2000}: 0.2053

Moreover, the plots to show the decision quadratic boundaries are shown on Figure 5 and Figure 6

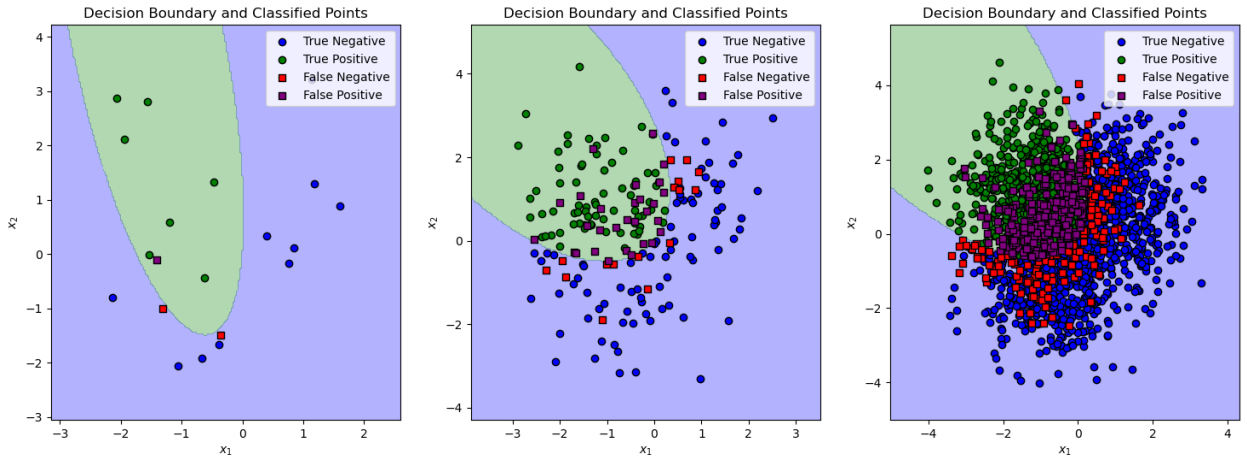

Figure 5: Plot to visualize the decision boundaries on the training datasets for each quadratic model

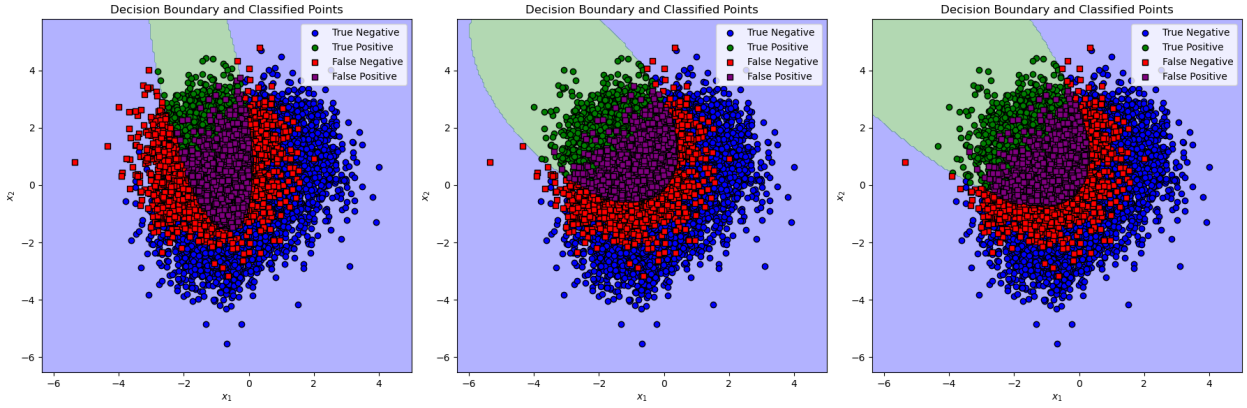

Figure 6: Plot to visualize the decision boundaries on the validation datasets for each quadratic model

Discussion

The performance of the classifiers on the validation set, trained with both linear and quadratic logistic regression, improves as the number of training samples increases. Moreover, the quadratic model outperforms the linear model as it better captures the structure of the data.

The theoretically optimal classifier achieves a minimum probability of error of 0.248 on the validation set, which is lower than the minimum probability of error achieved by the linear logistic classifier (0.3143) but higher than that of the quadratic logistic classifier (0.2053). This indicates that the quadratic model even surpasses the performance of the theoretically optimal classifier, despite the latter having access to the true parameters from the normal probability density function. The quadratic model effectively fits the data better when using a training set of 2000 samples and a validation set of 10000 samples.

Problem 2

The model is $y = c(x, w)+v$ where $c(x, w)$ is a cubic polynomial in x with coefficients $\mathbf{w}$, and v is a Gaussian noise with mean 0 and variance σ^2 and the dataset $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ consists of N i.i.d. samples of (x, y) pairs.

The goal of ML estimation is to find the parameter vector $\mathbf{w}$ that maximizes the likelihood of observing the dataset given the model.

Given that the noise v is Gaussian, the likelihood function for a single observation (x_i, y_i) is:

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

Assuming independence of the samples, the likelihood of the entire dataset is:

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

The log-likelihood (which is easier to maximize) is:

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

Maximizing the log-likelihood is equivalent to minimizing the sum of squared errors:

$$\min_w \sum_{i=1}^{N}(y_i - c(x_i, w))^2$$

This is a least-squares optimization problem and to solve it, we express the cubic polynomial $c(x, w)$ as a linear combination of terms:

$$c(x, w) = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2 + w_6x_1^3 + w_7x_2^3 + w_8x_1^2x_2 + w_9x_1x_2^2$$

Given this, we can construct a design matrix X where each row corresponds to one observation x_i, and each column corresponds to a term in the cubic polynomial:

$$X = \begin{bmatrix} 1 & x_{11} & x_{21} & x_{11}^2 & x_{21}^2 & x_{11}x_{21} & x_{11}^3 & x_{21}^3 & x_{11}^2x_{21} & x_{11}x_{21}^2 \\ 1 & x_{12} & x_{22} & x_{12}^2 & x_{22}^2 & x_{12}x_{22} & x_{12}^3 & x_{22}^3 & x_{12}^2x_{22} & x_{12}x_{22}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{1N} & x_{2N} & x_{1N}^2 & x_{2N}^2 & x_{1N}x_{2N} & x_{1N}^3 & x_{2N}^3 & x_{1N}^2x_{2N} & x_{1N}x_{2N}^2 \end{bmatrix}$$

This design matrix X has N rows (one for each sample) and 10 columns (one for each term in the polynomial).

The ML estimator w_{ML} is the solution to the normal equations:

$$w_{\text{ML}} = (X^TX)^{-1}X^Ty$$

This solution is obtained by setting the derivative of the least-squares cost function with respect to **w** to zero and solving for **w**.

The MAP estimator is also computed. To do so, we combine the likelihood of the data (given the model) with a prior distribution on **w**. The MAP approach integrates prior beliefs into the estimation, providing a regularized version of the maximum likelihood estimator.

We assume a zero-mean Gaussian prior on **w** with covariance matrix γI:

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

The MAP estimate maximizes the posterior distribution of **w** given the data:

$$w_{\text{MAP}} = \arg\max_w \; p(w \mid D)$$

By Bayes' Theorem:

$$p(w \mid D) \propto p(D \mid w) p(w)$$

Taking the log of the posterior:

$$\log p(w \mid D) = -\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - c(x_i, w))^2 - \frac{1}{2\gamma} w^T w + \text{constant}$$

To maximize the log-posterior, we minimize the negative log-posterior:

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

This expression shows that the MAP estimator is essentially a regularized least-squares problem, where the regularization term is $\frac{1}{2\gamma} w^T w$. The parameter γ controls the strength of the regularization.

We express the minimization problem in terms of the design matrix X:

$$w_{\text{MAP}} = \arg\min_w \; \frac{1}{2\sigma^2} \|y - Xw\|^2 + \frac{1}{2\gamma} \|w\|^2$$

The solution is obtained by setting the derivative with respect to **w** equal to zero:

$$\frac{1}{\sigma^2} X^T (Xw - y) + \frac{1}{\gamma} w = 0$$

$$w_{\text{MAP}} = (X^T X + \gamma I)^{-1} X^T y$$

After having derived the estimator expressions for both the MLE and the MAP, we have implemented them in code and applied them to the dataset generated with the Python script provided. The estimators have been obtained from the training dataset and using a range of values for γ from 10^{-m} to 10^{n}. Then, each model has been evaluated using the validation dataset.

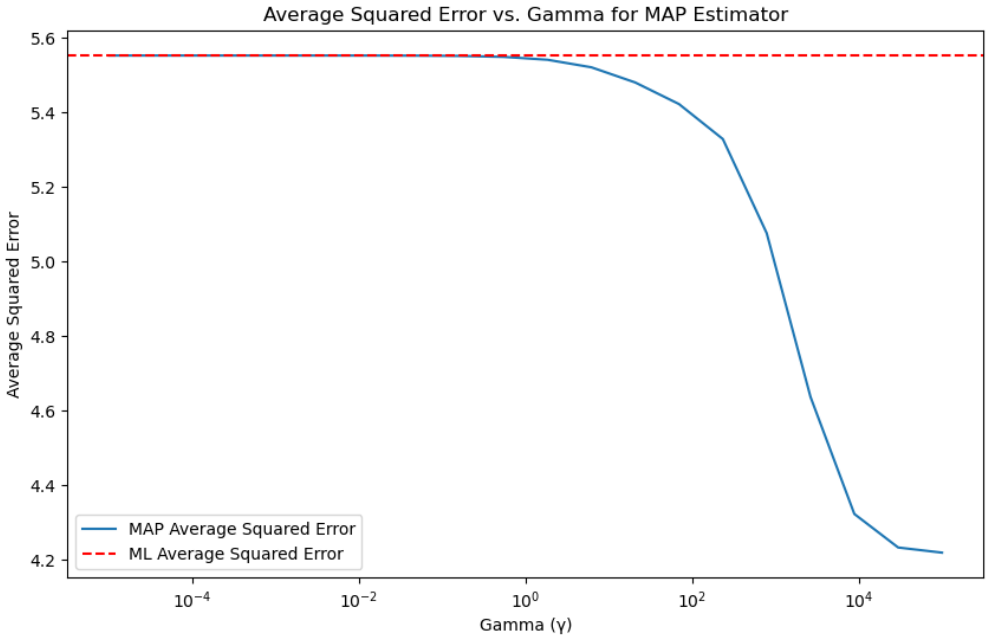

Figure 7: Average Squared Error vs. Gamma for MAP Estimator

Figure 7 plots the average squared error on the validation set as a function of the hyper-parameter gamma for the MAP estimator and also shows the value of the average squared error for the ML one, which is 7.28

As it can be seen, for small values of gamma (close to 0), the prior distribution for $\mathbf{w}$ has a zero-mean Gaussian with covariance matrix close to 0. This means that all coefficients $\mathbf{w}$ have the same prior distribution, which means that the MAP estimator gets closer to the ML estimator when gamma is close to 0. In this case, $\mathbf{w}$ is almost always constrained to be close to zero, meaning that the prior is very strong and the regularization term does not significantly alter the solution.

As γ increases, the covariance is bigger too, meaning that the estimator gains flexibility, leading to a reduction in squared error as it optimizes the fit more effectively.

If γ is very high, the regularization effect is minimal too and the estimator has enough flexibility to minimize the error, that's why the average squared error of the MAP estimator continues to decrease until it reaches a point where it stabilizes.

Why can MAP have a lower squared error than ML?

While the ML estimator minimizes the squared residuals without any constraint, this does not necessarily mean it achieves the best performance, especially when considering over fitting and generalization to unseen data.

While the ML minimizes:

$$w_{\mathrm{ML}} = \arg\min_{w} \sum_{i=1}^{N} (y_i - c(x_i, w))^2$$

MAP minimizes a modified version of the objective function that includes a regularization term:

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

Even though the ML estimator may achieve the smallest training error (squared residuals on the training set), it does not guarantee the smallest error on new, unseen data from the validation set. On the other hand, the MAP estimator, with appropriate regularization, aims to minimize both training and validation errors by preventing over fitting, which often results in better performance.

Problem 3

Optimization Problem

Given the vehicle's true position $[x_T, y_T]^T$ in 2-dimensional space and distance (range) measurements to K reference (landmark) coordinates $\{[x_1, y_1]^T, \ldots, [x_i, y_i]^T, \ldots, [x_K, y_K]^T\}$, the range measurements are defined as:

$$r_i = d_{T_i} + n_i \quad \text{for} \quad i \in \{1, \ldots, K\},$$

where: $d_{T_i} = \|[x_T, y_T]^T - [x_i, y_i]^T\|$ is the true distance between the vehicle and the i-th reference point, n_i is a zero mean Gaussian distributed measurement noise with known variance σ_i^2 and the noise in each measurement is independent from the others.

Assume we have prior knowledge regarding the vehicle's position given by:

$$p([x, y]^T) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{1}{2}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right)\right)$$

To determine the MAP estimate of the vehicle's position, $[x_{\text{MAP}}, y_{\text{MAP}}]^T$, we need to maximize the posterior distribution:

$$[x_{\text{MAP}}, y_{\text{MAP}}]^T = \arg\max_{x,y} \; p([x, y]^T \mid r_1, \ldots, r_K)$$

Applying Bayes' Theorem:

$$p([x, y]^T \mid r_1, \ldots, r_K) \propto p(r_1, \ldots, r_K \mid [x, y]^T) \cdot p([x, y]^T)$$

Assuming the measurements are independent, the likelihood function is:

$$p(r_1, \ldots, r_K \mid [x, y]^T) = \prod_{i=1}^{K} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(r_i - d_i)^2}{2\sigma_i^2}\right)$$

Taking the log of the posterior to form the objective function:

$$\log p([x, y]^T \mid r_1, \ldots, r_K) = \log p(r_1, \ldots, r_K \mid [x, y]^T) + \log p([x, y]^T) + C$$

where C is a constant that does not depend on x and y.

$$\log p([x, y]^T \mid r_1, \ldots, r_K) = -\sum_{i=1}^{K} \frac{(r_i - d_i)^2}{2\sigma_i^2} - \frac{1}{2}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right) + C$$

The MAP estimate $[x_{\text{MAP}}, y_{\text{MAP}}]^T$ is obtained by minimizing the negative log-posterior:

$$[x_{\text{MAP}}, y_{\text{MAP}}]^T = \arg\min_{x,y} \left\{ \sum_{i=1}^{K} \frac{(r_i - d_i)^2}{2\sigma_i^2} + \frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} \right\}$$

Computer code

The implementation of the code is based on this steps:

1. Generate the True Vehicle Position: The true vehicle position is generated inside a unit circle centered at the origin using polar coordinates where the radius is sampled from a uniform distribution between 0 and 1, and the angle is sampled from a uniform distribution between 0 and 2π.

2. Generate Landmarks: For each $K \in \{1, 2, 3, 4\}$, K landmarks are placed evenly spaced on the unit circle centered at the origin. This ensures that the landmarks are symmetrically distributed.

3. Generate Range Measurements: The range measurements from the vehicle to each landmark are generated using the model:

$$r_i = d_{T_i} + n_i$$

where $d_{T_i} = \|[x_T, y_T]^T - [x_i, y_i]^T\|$ is the true distance between the vehicle and the i-th landmark, and n_i is Gaussian noise with mean zero and standard deviation 0.3. If any generated range measurement is negative, it is resampled until it is non-negative.

4. MAP Objective Function: The MAP objective function is defined as:

$$J(x, y) = \frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2} + \sum_{i=1}^{K} \frac{(r_i - d_i)^2}{2\sigma^2}$$

where $\sigma_x = \sigma_y = 0.25$. The function takes into account both the likelihood (distance measurements) and the prior (Gaussian centered at the origin).

5. Plotting Contours: The MAP objective function is evaluated over a grid of points in the range $[-2, 2]$ for both x and y. Contour plots are generated to visualize the MAP objective function landscape. The true vehicle position is marked with a red plus (+), and the landmarks are marked with blue circles (o).

6. Evaluation: For each value of K, a contour plot of the MAP objective function is produced. These plots help visualize how the MAP estimate behaves as K increases. As K increases, the MAP estimate (center of the innermost contour) is expected to get closer to the true vehicle position and become more certain (tighter contours), indicating improved localization accuracy.

The contour plots for different values of K in Figure 8 demonstrate that as the number of landmarks increases, the MAP estimate becomes more accurate and certain. With more landmarks, the contours of the objective function become more concentrated around the true vehicle position, reflecting reduced uncertainty. This behavior is consistent with the expectation that increasing the number of independent measurements (landmarks) improves the localization accuracy of the MAP estimate.

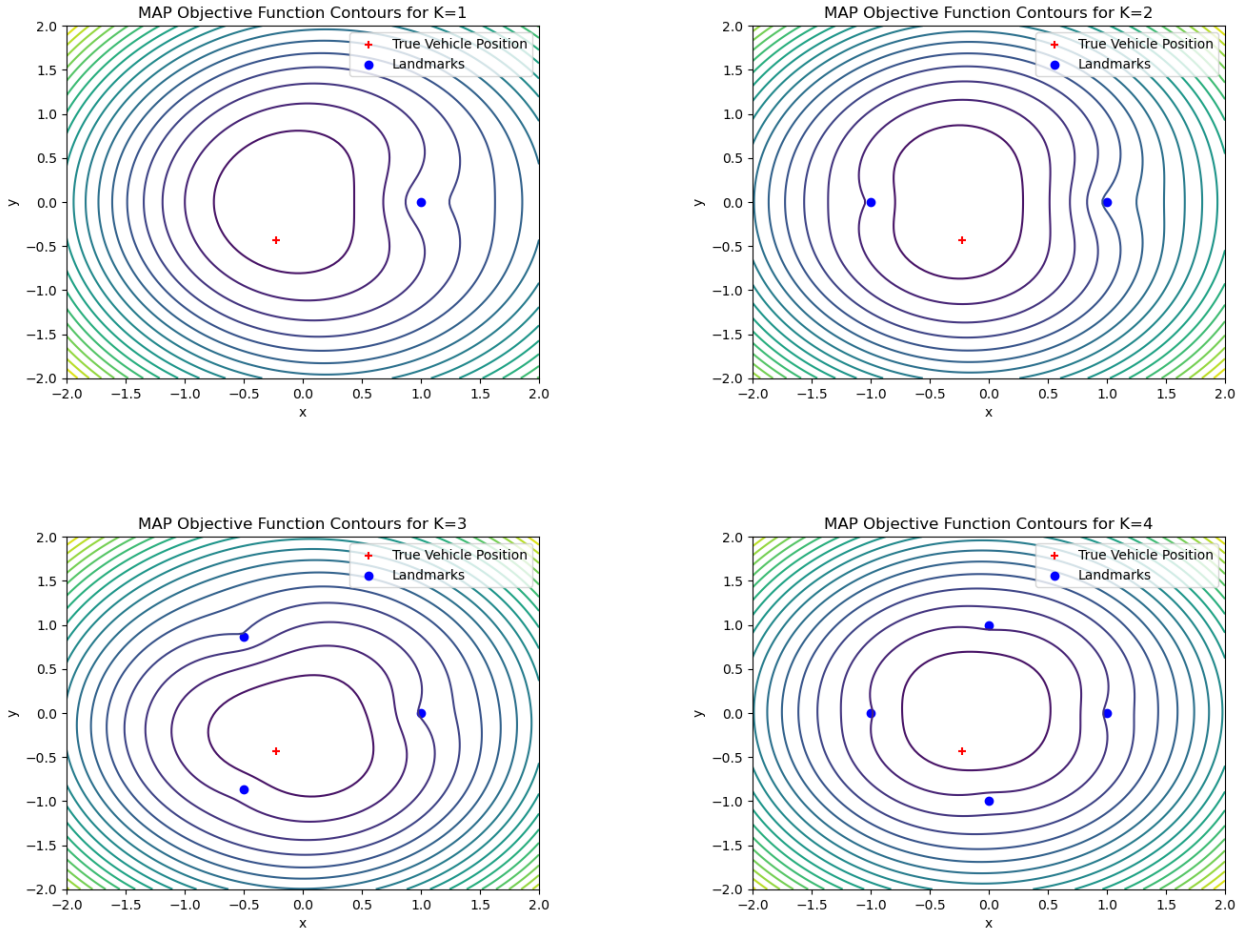

Figure 8: MAP objective function contours for K=1,2,3,4

Problem 4

We are given a classification problem where we have c classes, and we have the option to reject a pattern. The loss function is defined as follows:

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

where:

- λ_r is the loss incurred for choosing the rejection action (the $(c+1)$th action).
- λ_s is the loss incurred for making any substitution error (assigning to the wrong class).

The goal is to minimize the risk $R(\alpha_i|\mathbf{x})$, which is the expected loss given the feature vector $\mathbf{x}$. The risk can be expressed as:

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

To minimize the risk, we need to compare the risks associated with each possible action:

- If we assign the pattern to class ω_i, the risk is:

  $$R(\alpha_i|\mathbf{x}) = \lambda_s \left(1 - P(\omega_i|\mathbf{x})\right)$$

  because the only time the loss is zero is when $i = j$. For any other $j \neq i$, the loss is λ_s.

- If we choose the rejection action (denoted as α_{c+1}), the risk is:

  $$R(\alpha_{c+1}|\mathbf{x}) = \lambda_r$$

To minimize the risk, we choose the class ω_i if:

$$\lambda_s \left(1 - P(\omega_i|\mathbf{x})\right) \leq \lambda_r$$

Rearranging, we get:

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

Thus, the decision rule is:

- Assign the pattern to class ω_i if $P(\omega_i|\mathbf{x}) \geq 1 - \frac{\lambda_r}{\lambda_s}$ and if $P(\omega_i|\mathbf{x})$ is the maximum among all $P(\omega_j|\mathbf{x})$ for $j = 1, \ldots, c$.
- Otherwise, reject.

Case $\lambda_r = 0$

$$1 - \frac{\lambda_r}{\lambda_s} = 1$$

In this case, the decision rule becomes:

- Only assign the pattern to class ω_i if $P(\omega_i|\mathbf{x}) = 1$.
- Otherwise, reject.

This means that we only assign the pattern to a class if we are completely certain. Any uncertainty leads to rejection.

Case $\lambda_r > \lambda_s$

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

In this scenario, the condition $P(\omega_i|\mathbf{x}) \geq 1 - \frac{\lambda_r}{\lambda_s}$ is always satisfied because probabilities are always non-negative ($P(\omega_i|\mathbf{x}) \geq 0$). Therefore, the decision rule becomes:

- Assign the pattern to the class with the highest $P(\omega_i|\mathbf{x})$, as rejection will never be the optimal choice.

Problem 5

Let Z be drawn from a categorical distribution with K possible outcomes/states and parameter Θ, represented by $\text{Cat}(\Theta)$. The variable Z takes discrete values using a 1-of-K scheme:

$$z = [z_1, \ldots, z_K]^T \quad \text{where} \quad z_k = \begin{cases} 1 & \text{if the variable is in state } k, \\ 0 & \text{otherwise.} \end{cases}$$

The parameter vector for the probability density function is $\Theta = [\theta_1, \ldots, \theta_K]^T$, where $P(z_k = 1) = \theta_k$ for $k \in \{1, \ldots, K\}$.

Given a dataset $D = \{z_1, \ldots, z_N\}$ with i.i.d. samples $Z_n \sim \text{Cat}(\Theta)$ for $n \in \{1, \ldots, N\}$, we find:

ML Estimator for Θ

$$P(D \mid \Theta) = \prod_{n=1}^{N} P(Z_n \mid \Theta) = \prod_{n=1}^{N} \prod_{k=1}^{K} \theta_k^{z_{nk}}$$

where z_{nk} is the indicator variable for the n-th sample and k-th state.

The log-likelihood function is:

$$\log P(D \mid \Theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \log \theta_k$$

To find the ML estimator, we maximize $\log P(D \mid \Theta)$ with respect to Θ, subject to the constraint that $\sum_{k=1}^{K} \theta_k = 1$. Using a Lagrange multiplier λ, we define:

$$\mathcal{L}(\Theta, \lambda) = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} z_{nk} \right) \log \theta_k + \lambda \left(1 - \sum_{k=1}^{K} \theta_k \right)$$

Taking the partial derivative of $\mathcal{L}$ with respect to θ_k and setting it to zero:

$$\frac{\partial \mathcal{L}}{\partial \theta_k} = \frac{\sum_{n=1}^{N} z_{nk}}{\theta_k} - \lambda = 0$$

$$\theta_k = \frac{\sum_{n=1}^{N} z_{nk}}{\lambda}$$

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

$$\sum_{k=1}^{K} \frac{\sum_{n=1}^{N} z_{nk}}{\lambda} = 1$$

$$\lambda = \sum_{k=1}^{K} \sum_{n=1}^{N} z_{nk} = N$$

Therefore, the ML estimator for θ_k is:

$$\theta_k^{\mathrm{ML}} = \frac{\sum_{n=1}^{N} z_{nk}}{N}$$

MAP Estimator for Θ

Assuming the prior $p(\Theta)$ for the parameters is a Dirichlet distribution with hyperparameter $\alpha = [\alpha_1, \ldots, \alpha_K]^T$, the prior is:

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

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

The posterior distribution is:

$$p(\Theta \mid D) \propto P(D \mid \Theta) \cdot p(\Theta)$$

$$p(\Theta \mid D) \propto \prod_{k=1}^{K} \theta_k^{\left(\sum_{n=1}^{N} z_{nk}\right)} \cdot \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

$$p(\Theta \mid D) \propto \prod_{k=1}^{K} \theta_k^{\left(\sum_{n=1}^{N} z_{nk} + \alpha_k - 1\right)}$$

To find the MAP estimator, we maximize $\log p(\Theta \mid D)$ with respect to Θ. The constraint is $\sum_{k=1}^{K} \theta_k = 1$. We introduce a Lagrange multiplier λ to enforce this constraint:

$$\mathcal{L}(\Theta, \lambda) = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} z_{nk} + \alpha_k - 1 \right) \log \theta_k + \lambda \left(1 - \sum_{k=1}^{K} \theta_k \right)$$

Taking the partial derivative of $\mathcal{L}$ with respect to θ_k and setting it to zero:

$$\frac{\partial \mathcal{L}}{\partial \theta_k} = \frac{\sum_{n=1}^{N} z_{nk} + \alpha_k - 1}{\theta_k} - \lambda = 0$$

$$\theta_k = \frac{\sum_{n=1}^{N} z_{nk} + \alpha_k - 1}{\lambda}$$

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

$$\sum_{k=1}^{K} \frac{\sum_{n=1}^{N} z_{nk} + \alpha_k - 1}{\lambda} = 1$$

$$\lambda = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} z_{nk} + \alpha_k - 1 \right)$$

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

Substituting λ Back into θ_k

$$\theta_k^{\text{MAP}} = \frac{\sum_{n=1}^{N} z_{nk} + \alpha_k - 1}{N + \sum_{k=1}^{K} \alpha_k - K}$$

Appendix: Code

Listing 1: Question 1

```
import numpy as np
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import pandas as pd
from math import ceil, floor
from scipy.optimize import minimize
from sklearn.metrics import log_loss
from sklearn.preprocessing import PolynomialFeatures

np.random.seed(99)
rng = np.random.default_rng()

def generate_samples(dataset_size, priors, num_dim, means, cov, thresholds):
        # Determine dimensionality from mixture PDF parameters
        # Decide randomly which samples will come from each component u_i ~ Uniform(0, 1)
    for i = 1, ..., N (or 0, ..., N-1 in code)
  u = np.random.rand(dataset_size)

  labels = u >= priors[0]
  X = np.zeros((dataset_size, num_dim))
  for i in range(4):
    indice = np.argwhere((thresholds[i-1] <= u) & (u <= thresholds[i]))[:, 0]
    X[indice, :] = multivariate_normal.rvs(means[i-1], cov, len(indice))
  return labels, X

def get_posteriors(dataset_size, priors, num_classes, means, cov, x):
    posteriors = np.zeros(shape=[dataset_size, num_classes])
    for class_label in range(num_classes):
        if class_label == 0:
            posteriors[:, class_label] = 0.5 * (multivariate_normal.pdf(x, means[0], cov) +
                                                multivariate_normal.pdf(x, means[1], cov)) *
    priors[class_label]
        else:
            posteriors[:, class_label] = 0.5 * (multivariate_normal.pdf(x, means[2], cov) +
                                                multivariate_normal.pdf(x, means[3], cov)) *
    priors[class_label]

    return posteriors

## Data generation
# constants
datasets_train_size = [20,200,2000]
dataset_val_size = 10000
num_dim = 2
num_classes = 2
priors = np.array([0.6, 0.4])
# it uses the thresholds to jointly decide both the class and which Gaussian component
    within that
# class to use for generating the sample; if first we choose one and then the other, it is
    conditional
thresholds = np.array([0, 0.3, 0.6, 1.0])

# parameters of class-conditional Gaussians pdfs
mean0 = np.array([-0.9,-1.1])
mean1 = np.array([0.8,0.75])
mean2 = np.array([-1.1,0.9])
mean3 = np.array([0.9,-0.75])
means = np.array([mean0, mean1, mean2, mean3])
cov = np.array([[0.75, 0], [0, 1.25]])

# Plot the original data and their true labels
fig, ax = plt.subplots(2, 2, figsize=(10, 10))

# generate samples for training datasets
X_train = []
labels_train = []
dist_labels_train = []
t = 0 #axis index
for dataset_train_size in datasets_train_size:
    class_labels, x = generate_samples(dataset_train_size, priors, num_dim, means, cov,
    thresholds)
```

```
    X_train.append(x)
    labels_train.append(class_labels)
    dist_labels_train.append(np.array((sum(class_labels == 0), sum(class_labels == 1))))
    # Axis fancy indexing for the sake of plotting correctly in subplots
    ax[floor(t/2), t%2].set_title(r"Training $D^{%d}$" % (dataset_train_size))
    ax[floor(t/2), t%2].plot(x[class_labels==0, 0], x[class_labels==0, 1], 'p',
    label="Class 0")
    ax[floor(t/2), t%2].plot(x[class_labels==1, 0], x[class_labels==1, 1], 'r+',
    label="Class 1")
    ax[floor(t/2), t%2].set_xlabel(r"$x_1$")
    ax[floor(t/2), t%2].set_ylabel(r"$x_2$")
    ax[floor(t/2), t%2].legend()

    t += 1

# generate samples for validation dataset
labels_val, X_val = generate_samples(dataset_val_size, priors, num_dim, means, cov,
    thresholds)
dist_labels_val = np.array((sum(labels_val == 0), sum(labels_val == 1)))

ax[1, 1].set_title(r"Validation $D^{%d}$" % (dataset_val_size))
ax[1, 1].plot(X_val[labels_val==0, 0], X_val[labels_val==0, 1], 'p', label="Class 0")
ax[1, 1].plot(X_val[labels_val==1, 0], X_val[labels_val==1, 1], 'r+', label="Class 1")
ax[1, 1].set_xlabel(r"$x_1$")
ax[1, 1].set_ylabel(r"$x_2$")
ax[1, 1].legend()

# Using validation set samples to limit axes (most samples drawn, highest odds of spanning
    sample space)
x1_valid_lim = (floor(np.min(X_val[:,0])), ceil(np.max(X_val[:,0])))
x2_valid_lim = (floor(np.min(X_val[:,1])), ceil(np.max(X_val[:,1])))
# Keep axis-equal so there is new skewed perspective due to a greater range along one axis
plt.setp(ax, xlim=x1_valid_lim, ylim=x2_valid_lim)
plt.tight_layout()
plt.show()

# posterior pdf
posteriors_train = []
for x in X_train:
    posteriors_train.append(get_posteriors(x.shape[0], priors, num_classes, means, cov, x))

posterior_val = get_posteriors(len(X_val), priors, num_classes, means, cov, X_val)


## Part 1
gamma_values = np.linspace(start=0, stop=1000, num=20000)
gamma_values = np.append(gamma_values, np.inf)

# For every gamma, we compute the TPR, FPR and probabilities of error
TPRs = []
FPRs = []
prob_errors = []

for gamma in gamma_values:
    ratio = posterior_val[:,1]/posterior_val[:,0]
    decisions = ratio > gamma # np array of 1,0 decisions

    TP = np.sum((decisions == 1) & (labels_val == 1))
    FP = np.sum((decisions == 1) & (labels_val == 0))
    FN = np.sum((decisions == 0) & (labels_val == 1))
    TN = np.sum((decisions == 0) & (labels_val == 0))

    TPR = TP/(TP+FN) # true positive detection probability
    FPR = FP/(FP+TN) # false positive probability
    FNR = FN/(FN+TP)
    prob_error = FPR * priors[0] + FNR * priors[1]

    TPRs.append(TPR)
    FPRs.append(FPR)
    prob_errors.append(prob_error)

min_prob_error = min(prob_errors)
optimal_gamma_index = prob_errors.index(min_prob_error)
optimal_gamma = gamma_values[optimal_gamma_index]

# theoretical optimal gamma
```

```
decisions_theo = (posterior_val[:,1]/posterior_val[:,0]) > priors[0]/priors[1] # np array
    of 1,0 decisions

TP_theo = np.sum((decisions_theo == 1) & (labels_val == 1))
FP_theo = np.sum((decisions_theo == 1) & (labels_val == 0))
FN_theo = np.sum((decisions_theo == 0) & (labels_val == 1))
TN_theo = np.sum((decisions_theo == 0) & (labels_val == 0))

TPR_theo = TP_theo/(TP_theo+FN_theo) # true positive detection probability
FPR_theo = FP_theo/(FP_theo+TN_theo) # false positive probability
FNR_theo = FN_theo/(FN_theo+TP_theo)
prob_error_theo = FPR_theo * priors[0] + FNR_theo * priors[1]

# Plotting the ROC Curve and highlighting the optimal gamma
plt.figure(figsize=(8, 6))
plt.plot(FPRs, TPRs, label='ROC Curve', marker = '.', zorder=2)
plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')  # Dashed diagonal line (random
    classifier)
plt.scatter(FPRs[optimal_gamma_index], TPRs[optimal_gamma_index], color='red', zorder=3,
    label=f'  ={optimal_gamma:.3f} achieves min P(error; gamma), which is
    {min_prob_error:.3f}')
plt.plot(FPR_theo, TPR_theo, '+', label=f'Theoretical threshold = {priors[0]/priors[1]:.3f}
    achieves min P(error)={prob_error_theo:.3f}', color = 'black', zorder=4)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

## Part 2: Linear logistic regression
# Sigmoid function for logistic regression
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Negative Log-Likelihood Loss function for logistic regression
def nll_loss(w, X, y):
    z = np.dot(X, w)
    predictions = sigmoid(z)
    return log_loss(y, predictions)

# Training logistic regression model using MLE
def train_logistic_regression(X, y):
    w0 = np.zeros(X.shape[1])  # Initialize weights as zeros
    result = minimize(nll_loss, w0, args=(X, y), method='BFGS')  # Optimize the NLL
    return result.x  # Return the optimized weights

# Logistic regression classifier
def classify(X, w):
    probabilities = sigmoid(np.dot(X, w))
    return (probabilities >= 0.5).astype(int)

def plot_decision_boundary_with_distinct_markers(X, y, w, ax, labels):
    """
    Plots the decision boundary defined by the logistic regression model (w),
    along with the correctly and incorrectly classified points from the validation set,
    using different markers for the two true classes.
    """
    # Meshgrid for plotting decision boundary
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

    # Define x1 range (horizontal axis)
    x1_vals = np.linspace(x_min, x_max, 100)  # Choose an appropriate range for x1

    # Calculate corresponding x2 values based on the equation: x2 = -(w1 / w2) * x1
    x2_vals = -(w[0] / w[1]) * x1_vals

    # Plot the line
    ax.plot(x1_vals, x2_vals, color='blue', label=f'Line with w: {w}')

    # Correctly and incorrectly classified points
    TN = (y == 0) & (y == labels)
    TP = (y == 1) & (y == labels)
    FN = (y == 0) & (y != labels)
    FP = (y == 1) & (y != labels)
```

```
    FPR = np.sum(FP)/(np.sum(FP)+np.sum(TN)) # false positive probability
    FNR = np.sum(FN)/(np.sum(FN)+np.sum(TP))
    prob_error = FPR * priors[0] + FNR * priors[1]

    # True class 0 marker: 'o', True class 1 marker: 's'
    # Correctly classified points
    ax.scatter(X[TN, 0], X[TN, 1], marker='o', color='green', label='Correct Class 0',
    edgecolor='k')
    ax.scatter(X[TP, 0], X[TP, 1], marker='s', color='green', label='Correct Class 1',
    edgecolor='k')

    # Incorrectly classified points
    ax.scatter(X[FN, 0], X[FN, 1], marker='o', color='red', label='Incorrect Class 0',
    edgecolor='k')
    ax.scatter(X[FP, 0], X[FP, 1], marker='s', color='red', label='Incorrect Class 1',
    edgecolor='k')

    ax.set_xlabel(r'$x_1$')
    ax.set_ylabel(r'$x_2$')
    ax.legend(loc='upper right')
    ax.set_title("Decision Boundary and Classified Points")

    return prob_error

# Apply logistic regression for each dataset size and validate on D_10K with plots
fig_train, axes_train = plt.subplots(1, 3, figsize=(18, 6))  # 3 subplots for the 3 datasets
fig_val, axes_val = plt.subplots(1, 3, figsize=(18, 6))  # 3 subplots for the 3 datasets

for i, (X, y) in enumerate(zip(X_train, labels_train)):
    # Train logistic regression model on dataset
    w = train_logistic_regression(X, y)

    # Predict labels on training set
    y_pred_train = classify(X, w)

    # Predict labels on validation set
    y_pred_val = classify(X_val, w)

    prob_error_train = plot_decision_boundary_with_distinct_markers(X, y_pred_train, w,
    axes_train[i], y)
    prob_error_val = plot_decision_boundary_with_distinct_markers(X_val, y_pred_val, w,
    axes_val[i], labels_val)

    print(f'Probability of error on training dataset using the model trained with the
    training dataset D_{datasets_train_size[i]}: {prob_error_train:.4f}')
    print(f'Probability of error on validation dataset using the model trained with the
    training dataset D_{datasets_train_size[i]}: {prob_error_val:.4f}')
    # Plot decision boundary and classified points

# Show the plot
plt.tight_layout()
plt.show()

## Part 2: Quadratic logistic regression
# Training quadratic logistic regression model using MLE
def train_quadratic_logistic_regression(X, y):
    # Add quadratic terms to X
    poly = PolynomialFeatures(degree=2, include_bias=False)
    X_quad = poly.fit_transform(X)

    w0 = np.zeros(X_quad.shape[1])  # Initialize weights as zeros
    result = minimize(nll_loss, w0, args=(X_quad, y), method='BFGS')  # Optimize the NLL
    return result.x, poly  # Return the optimized weights and the polynomial transformer

# Logistic regression classifier
def classify(X, w, poly):
    X_quad = poly.transform(X)  # Add quadratic terms
    probabilities = sigmoid(np.dot(X_quad, w))
    return (probabilities >= 0.5).astype(int)

def plot_decision_boundary_with_distinct_markers(X, y, w, ax, labels, poly):
    """
    Plots the decision boundary defined by the logistic regression model (w),
    along with the correctly and incorrectly classified points from the validation set,
    using different markers for the two true classes.
    """
```

```
    # Meshgrid for plotting decision boundary
    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.linspace(x_min, x_max, 200), np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]

    Z = classify(grid, w, poly).reshape(xx.shape)

    # Plot the decision boundary
    ax.contourf(xx, yy, Z, alpha=0.3, levels=[-0.1, 0.5, 1.1], colors=["blue", "green"])

    # Correctly and incorrectly classified points
    TN = (y == 0) & (y == labels)
    TP = (y == 1) & (y == labels)
    FN = (y == 0) & (y != labels)
    FP = (y == 1) & (y != labels)

    FPR = np.sum(FP) / (np.sum(FP) + np.sum(TN))  # false positive probability
    FNR = np.sum(FN) / (np.sum(FN) + np.sum(TP))
    prob_error = FPR * priors[0] + FNR * priors[1]

    # True class 0 marker: 'o', True class 1 marker: 's'
    # Correctly classified points
    ax.scatter(X[TN, 0], X[TN, 1], marker='o', color='blue', label='True Negative',
    edgecolor='k')
    ax.scatter(X[TP, 0], X[TP, 1], marker='o', color='green', label='True Positive',
    edgecolor='k')

    # Incorrectly classified points
    ax.scatter(X[FN, 0], X[FN, 1], marker='s', color='red', label='False Negative',
    edgecolor='k')
    ax.scatter(X[FP, 0], X[FP, 1], marker='s', color='purple', label='False Positive',
    edgecolor='k')

    ax.set_xlabel(r'$x_1$')
    ax.set_ylabel(r'$x_2$')
    ax.legend(loc='upper right')
    ax.set_title("Decision Boundary and Classified Points")

    return prob_error

# Apply quadratic logistic regression for each dataset size and validate on D_10K with plots
fig_train, axes_train = plt.subplots(1, 3, figsize=(18, 6))  # 3 subplots for the 3 datasets
fig_val, axes_val = plt.subplots(1, 3, figsize=(18, 6))  # 3 subplots for the 3 datasets

for i, (X, y) in enumerate(zip(X_train, labels_train)):
    # Train quadratic logistic regression model on dataset
    w, poly = train_quadratic_logistic_regression(X, y)

    # Predict labels on training set
    y_pred_train = classify(X, w, poly)

    # Predict labels on validation set
    y_pred_val = classify(X_val, w, poly)

    prob_error_train = plot_decision_boundary_with_distinct_markers(X, y_pred_train, w,
    axes_train[i], y, poly)
    prob_error_val = plot_decision_boundary_with_distinct_markers(X_val, y_pred_val, w,
    axes_val[i], labels_val, poly)

    print(f'Probability of error on training dataset using the model trained with the
    training dataset D_{datasets_train_size[i]}: {prob_error_train:.4f}')
    print(f'Probability of error on validation dataset using the model trained with the
    training dataset D_{datasets_train_size[i]}: {prob_error_val:.4f}')

# Show the plot
plt.tight_layout()
plt.show()
```

Listing 2: Question 2

```
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
from hw2q2 import hw2q2
```

```
np.random.seed(99)

rng = np.random.default_rng()

def cubic_polynomial(x, w):
    """ Computes the cubic polynomial c(x, w) """
    return w[0] + w[1]*x[0] + w[2]*x[1] + w[3]*x[0]**2 + w[4]*x[1]**2 + w[5]*x[0]*x[1] + \
           w[6]*x[0]**3 + w[7]*x[1]**3 + w[8]*x[0]**2*x[1] + w[9]*x[0]*x[1]**2

def ml_estimator(x_train, y_train):
    """ Computes the ML estimator for w """
    N = x_train.shape[1]
    X_design = np.vstack([
        np.ones(N), x_train[0], x_train[1], x_train[0]**2, x_train[1]**2,
    x_train[0]*x_train[1],
        x_train[0]**3, x_train[1]**3, x_train[0]**2*x_train[1], x_train[0]*x_train[1]**2
    ]).T
    w_ml = inv(X_design.T @ X_design) @ X_design.T @ y_train
    return w_ml

def map_estimator(x_train, y_train, gamma):
    """ Computes the MAP estimator for w with a Gaussian prior """
    N = x_train.shape[1]
    X_design = np.vstack([
        np.ones(N), x_train[0], x_train[1], x_train[0]**2, x_train[1]**2,
    x_train[0]*x_train[1],
        x_train[0]**3, x_train[1]**3, x_train[0]**2*x_train[1], x_train[0]*x_train[1]**2
    ]).T
    I = np.eye(X_design.shape[1])
    w_map = inv(X_design.T @ X_design + gamma * I) @ X_design.T @ y_train
    return w_map

def evaluate_model(x, y, w):
    """ Evaluates the model using average squared error """
    y_pred = np.array([cubic_polynomial(x[:, i], w) for i in range(x.shape[1])])
    return np.mean((y - y_pred) ** 2)

x_train, y_train, x_validate, y_validate = hw2q2()

# Obtain ML estimator
w_ml = ml_estimator(x_train, y_train)

# Test a range of gamma values
gammas = np.logspace(-5, 5, 20)
errors_ml = evaluate_model(x_validate, y_validate, w_ml)
errors_map = []

for gamma in gammas:
    w_map = map_estimator(x_train, y_train, gamma)
    error_map = evaluate_model(x_validate, y_validate, w_map)
    errors_map.append(error_map)

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(gammas, errors_map, label='MAP Average Squared Error')
plt.axhline(y=errors_ml, color='r', linestyle='--', label='ML Average Squared Error')
plt.xscale('log')
plt.xlabel('Gamma ( )')
plt.ylabel('Average Squared Error')
plt.title('Average Squared Error vs. Gamma for MAP Estimator')
plt.legend()
plt.show()

print(errors_ml)
print(errors_map)
```

Listing 3: Question 3

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

np.random.seed(99)

rng = np.random.default_rng()
```

```python
def generate_vehicle_position():
    """Generate a true vehicle position inside a unit circle centered at the origin."""
    angle = np.random.uniform(0, 2 * np.pi)
    radius = np.random.uniform(0, 1)
    return np.array([radius * np.cos(angle), radius * np.sin(angle)])

def generate_landmarks(K):
    """Generate K evenly spaced landmarks on the unit circle centered at the origin."""
    angles = np.linspace(0, 2 * np.pi, K, endpoint=False)
    landmarks = np.array([[np.cos(angle), np.sin(angle)] for angle in angles])
    return landmarks

def generate_range_measurements(vehicle_pos, landmarks, sigma):
    """Generate range measurements with Gaussian noise."""
    measurements = []
    for landmark in landmarks:
        while True:
            distance = np.linalg.norm(vehicle_pos - landmark)
            noise = np.random.normal(0, sigma)
            range_measurement = distance + noise
            if range_measurement >= 0:
                measurements.append(range_measurement)
                break
    return np.array(measurements)

def map_objective(x, y, landmarks, measurements, sigma, sigma_x, sigma_y):
    """Calculate the MAP objective function value for a given position [x, y]."""
    prior_term = (x**2 / (2 * sigma_x**2)) + (y**2 / (2 * sigma_y**2))
    likelihood_term = 0
    for i, landmark in enumerate(landmarks):
        distance = np.linalg.norm(np.array([x, y]) - landmark)
        likelihood_term += (measurements[i] - distance)**2 / (2 * sigma**2)
    return prior_term + likelihood_term

def plot_map_contours(vehicle_pos, landmarks, measurements, sigma, sigma_x, sigma_y, K):
    """Plot the MAP objective function contours."""
    x_vals = np.linspace(-2, 2, 100)
    y_vals = np.linspace(-2, 2, 100)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z = np.array([[map_objective(x, y, landmarks, measurements, sigma, sigma_x, sigma_y)
                   for x in x_vals] for y in y_vals])

    plt.figure()
    contour_levels = np.linspace(np.min(Z), np.max(Z), 20)
    plt.contour(X, Y, Z, levels=contour_levels)
    plt.scatter(vehicle_pos[0], vehicle_pos[1], color='red', marker='+', label='True
    Vehicle Position')
    plt.scatter(landmarks[:, 0], landmarks[:, 1], color='blue', marker='o',
    label='Landmarks')
    plt.title(f'MAP Objective Function Contours for K={K}')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()

sigma = 0.3
sigma_x = sigma_y = 0.25
vehicle_pos = generate_vehicle_position()

for K in [1, 2, 3, 4]:
    landmarks = generate_landmarks(K)
    measurements = generate_range_measurements(vehicle_pos, landmarks, sigma)
    plot_map_contours(vehicle_pos, landmarks, measurements, sigma, sigma_x, sigma_y, K)
```