EECE 5644 Intro to Machine Learning Homework #3

Question 1 (50%)

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

The class priors are $P(L=0) = 0.6$ and $P(L=1) = 0.4$. The class class-conditional pdfs are $p(\mathbf{x}|L=0) = w_1 g(\mathbf{x}|\mathbf{m}_{01}, \mathbf{C}_{01}) + w_2 g(\mathbf{x}|\mathbf{m}_{02}, \mathbf{C}_{02})$ and $p(\mathbf{x}|L=1) = g(\mathbf{x}|\mathbf{m}_1, \mathbf{C}_1)$, where $g(\mathbf{x}|\mathbf{m}, \mathbf{C})$ is a multivariate Gaussian probability density function with mean vector $\mathbf{m}$ and covariance matrix $\mathbf{C}$. The parameters of the class-conditional Gaussian pdfs are: $w_1 = w_2 = 1/2$, and

$$\mathbf{m}_{01} = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \quad \mathbf{C}_{01} = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix} \quad \mathbf{m}_{02} = \begin{bmatrix} 0 \\ 4 \end{bmatrix} \quad \mathbf{C}_{02} = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix} \quad \mathbf{m}_1 = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad \mathbf{C}_1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

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

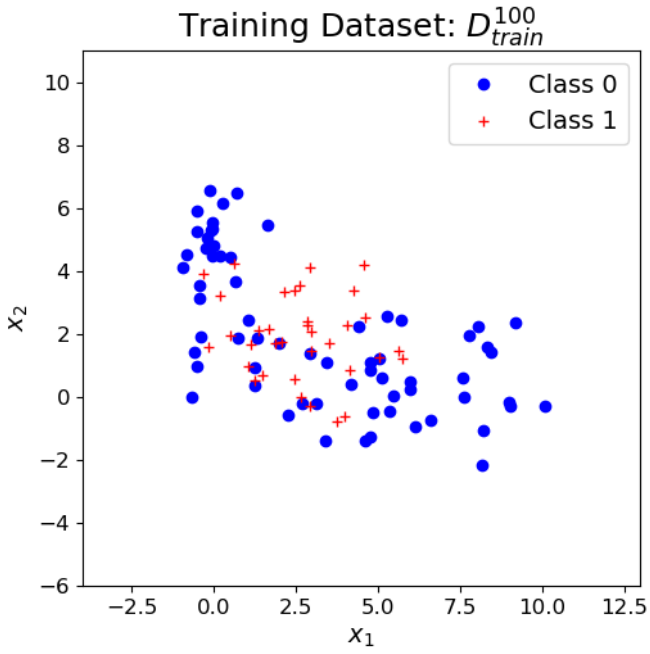
- D_{train}^{100} consists of 100 samples and their labels for training:
- D_{train}^{1000} consists of 1000 samples and their labels for training:
- D_{train}^{10K} consists of 10000 samples and their labels for training:
- $D_{validate}^{20K}$ consists of 20000 samples and their labels for training:

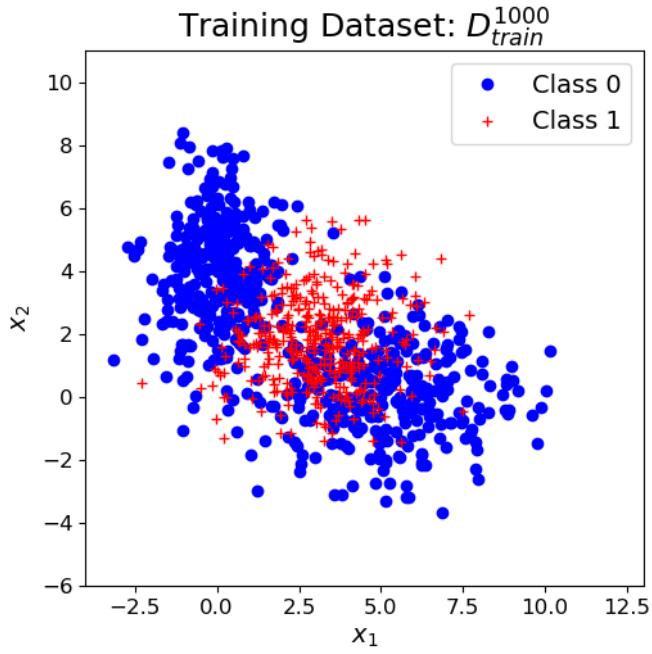
Part 1: (10%)

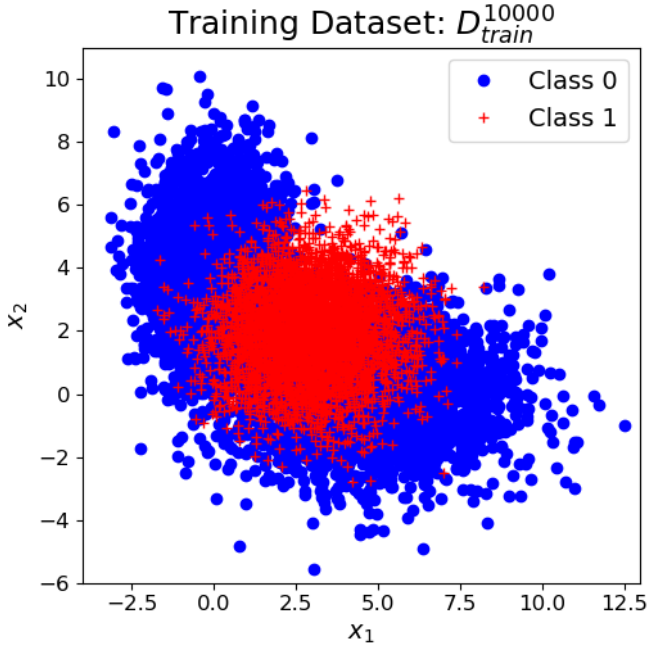
Determine the theoretically optimal classifier that achieves minimum probability of error using the knowledge of the true pdf. Specify the classifier mathematically and implement it; then apply it to all samples in $D_{validate}^{20K}$ validate. From the decision results and true labels for this validation set, estimate and plot the ROC curve of this min-P(error) classifier, and on the ROC curve indicate, with a special marker, the point that corresponds to the min-P(error) classifier's operating point. Also report your estimate of the min-P(error) achievable, based on counts of decision-truth label pairs on $D_{validate}^{20K}$.

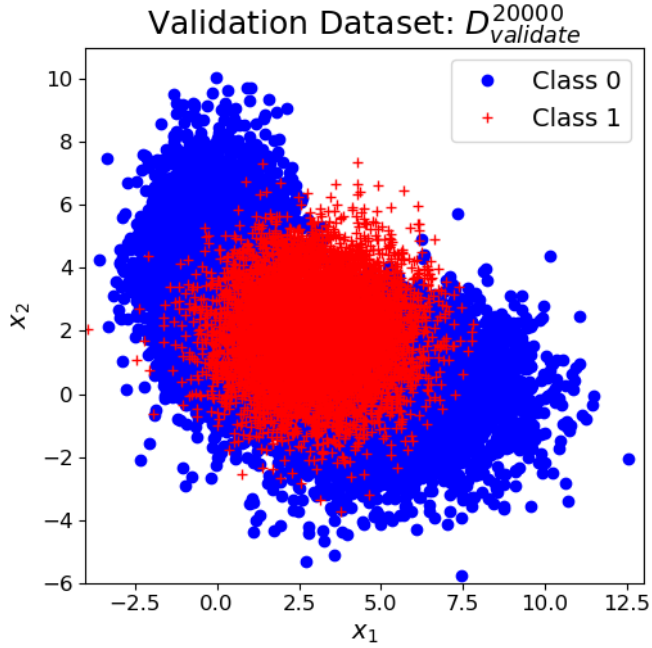
Optional: As supplementary visualization, generate a plot of the decision boundary of this classification rule overlaid on the validation dataset. This establishes an aspirational performance level on this data for the following approximations.

Generated Datasets:









The theoretically optimal classifier that will achieve min-P(error) using the knowledge of true pdf is the ERM classifier with 0-1 loss or Maximum a Posterior (MAP) classifier

$$\frac{p(\mathbf{x}|L=1)}{p(\mathbf{x}|L=0)} > \frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})}\frac{P(L=0)}{P(L=1)}$$

Since we are assuming 0-1 loss, this decision rule reduces to a MAP classification rule involving the likelihood ratio test of the class-conditional PDFs and the class priors of the model:

$$\ln\frac{p(\mathbf{x}|L=1)}{p(\mathbf{x}|L=0)} > \ln\frac{P(L=0)}{P(L=1)}$$

$$\ln p(\mathbf{x}|L=1) - \ln p(\mathbf{x}|L=0) > \ln\left(\frac{0.6}{0.4}\right)$$

$$\ln p(\mathbf{x}|L=1) - \ln p(\mathbf{x}|L=0) > \ln(1.5)$$

Thus, the decision rule for the theoretically optimal classifier, are the MAP decision rules:

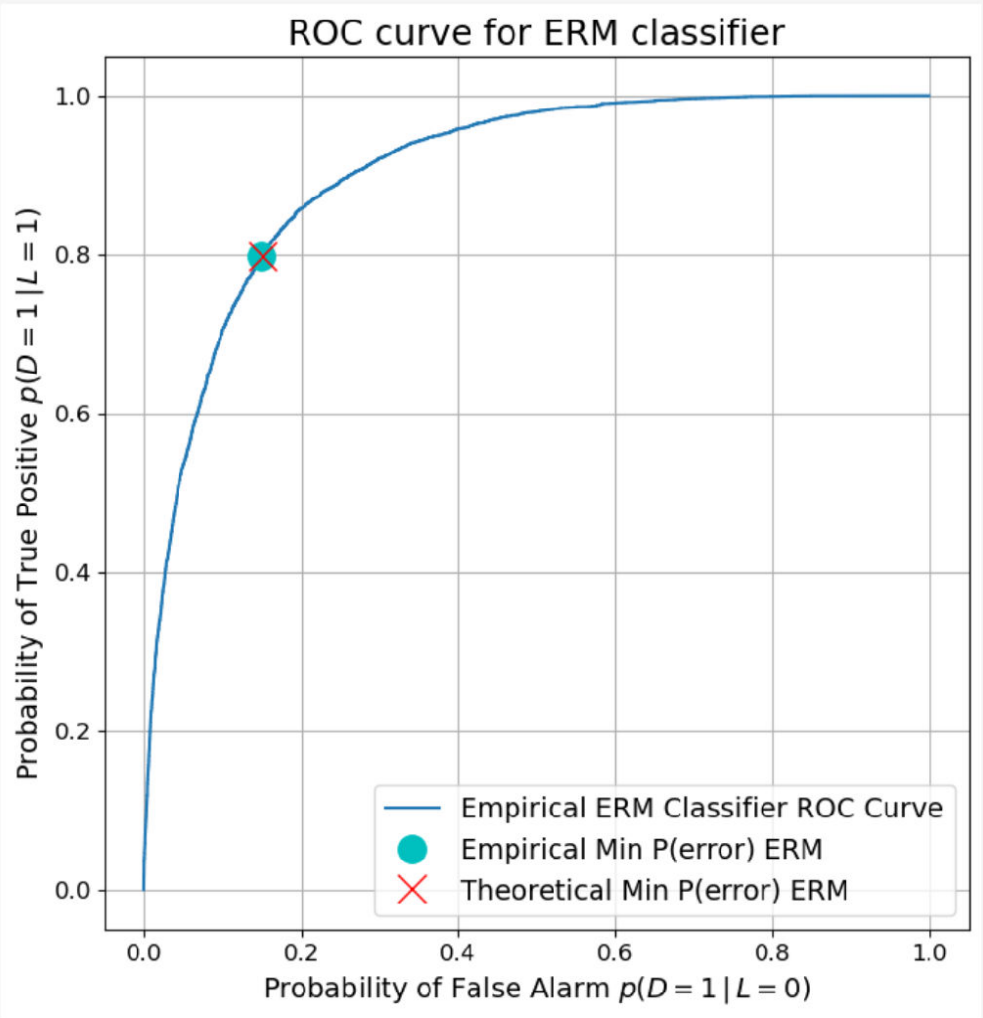
$$\ln p(\mathbf{x}|L=1) - \ln p(\mathbf{x}|L=0) > \ln(1.5) \rightarrow \text{ Decide Class 1 } (D=1)$$

and

$$\ln p(\mathbf{x}|L=1) - \ln p(\mathbf{x}|L=0) < \ln(1.5) \rightarrow \text{ Decide Class 0 } (D=0)$$

From here, we will express the decision-threshold value as $\gamma = 1.5$

After evaluating the above decision rule on the $D^{20K}_{validation}$ dataset in Python, the following ROC curve with estimated Empirical and Theoretical min-P(error) were generated as:




```
Min Empirical Pr(error) for ERM = 0.1708
Min Empirical Gamma = 1.510
Min Theoretical Pr(error) for ERM = 0.1709
Min Theoretical Gamma = 1.500
```

From my estimations:

The minimum empirical $P(error) = 0.1708$ for the ERM Classifier with an empirical value of $\gamma = 1.510$

The minimum theoretical $P(error) = 0.1709$ for the ERM Classifier with a theoretical value of $\gamma = 1.5$, which is what was previously calculated above.

Part 2: (20%)

(a) Using the maximum likelihood parameter estimation technique, estimate the class priors and class conditional pdfs using training data in D_{train}^{10K}. As class conditional pdf models, for $L = 0$ use a Gaussian Mixture model with 2 components, and for $L = 1$ use a single Gaussian pdf model. For each estimated parameter, specify the maximum-likelihood-estimation objective function that is maximized as well as the iterative numerical optimization procedure used, or if applicable, for analytically tractable parameter estimates, specify the estimator formula. Using these estimated class priors and pdfs, design and implement an approximation of the min-P(error) classifier, apply it on the validation dataset $D_{validate}^{20K}$. Report the ROC curve and minimum probability of error achieved on the validation dataset with this classifier that is trained with 10000 samples.

MLE parameters:

For Class 1, since this only a single Gaussian pdf model, the maximum-likelihood-estimations for its parameters can be derived using the Sample Mean, Sample Covariance, and number of training samples to calculate the class prior value.

The formulas for each would be:

Sample Mean:

$$\widehat{\mu_k} = \frac{1}{N}\sum_{i=1}^{n} x_i$$

Sample Covariance:

$$\widehat{\Sigma}_k = \sum_{i=1}^{n} (\mathbf{x} - \widehat{\mu}_k)(\mathbf{x} - \widehat{\mu}_k)^T$$

Prior:

$$\pi_k = \frac{N}{n}$$

where the N is the number of training sampled for class 1, and n is the total number of samples in entire dataset $D \in \{100, 1000, 10000\}$

For Class 0, because we are considering it to be a Gaussian Mixture Model with 2 components, we have to use a more complex method for calculating the MLE parameters.

This can be achieved by using the EM algorithm, where we iteratively compute the most maximum-likelihood for what each parameter for the 2 component GMM should be. This can be down using the following formulas:

In the general case, the MLE for GMM with unknown parameters $\theta = \{\mu_1, \dots, \mu_k, \sigma_1, \dots, \sigma_k, \pi_k, \dots, \pi_k\}$ can be expressed using the likelihood function:

$$L(\theta|X_i, \dots, X_n) = \prod_{i=1}^{n} \sum_{k=1}^{K} \pi_k N(x_i; \mu_k; \Sigma_k)$$

which can also be expressed in the log-likelihood

$$l(\theta) = \sum_{i=1}^{n} \ln \left(\sum_{k=1}^{K} \pi_k N(x_i; \mu_k; \Sigma_k) \right)$$

where the GMM contains K mixture components.

In order to apply the EM algorithm to a GMM, we first introduce the latent variable $Z_i \in \{1, \dots, K\}$ to represent the mixture component for X_i, then $P(X_i|Z_i)$ represents the distribution of the mixture component with π_k being the mixture prior value for the kth component in X_i

In the Expectation-Step, we calculate the posterior of Z_i:

$$P(Z_i = k|X_i) = \frac{P(X_i|Z_i = k)P(Z_i = k)}{P(X_i)} = \frac{\pi_k N(\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(\mu_j, \Sigma_j)} = \gamma(z_{nk})$$

Then, using the expression for $\gamma(z_{nk})$ into the the formula for EM algorithm:

$$Q(\theta^*|\theta) = E[\ln p(\mathbf{X}, \mathbf{Z}|\theta^*)] = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \theta) \ln p(\mathbf{X}, \mathbf{Z}|\theta)$$

where $p(\mathbf{Z}|\mathbf{X}, \theta) = \gamma(z_{nk})$ and $p(\mathbf{X}, \mathbf{Z}|\theta) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi^{z_{nk}} N(x_n|\mu_k, \Sigma_k)^{z_{nk}}$

The formula for the Expectation-step for the EM algorithm simplifies to:

$$Q(\theta^*|\theta) = \prod_{n=1}^{N} \prod_{k=1}^{K} \gamma(z_{nk})[\ln \pi_k + \ln N(x_n|\mu_k, \Sigma_k)]$$

The M-Step is then described by:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} Q(\theta^*|\theta)$$

where the old values of θ are used to calcualte the new values $\rightarrow \theta^*$

The MLE parameters are then estimated for the GMM by:

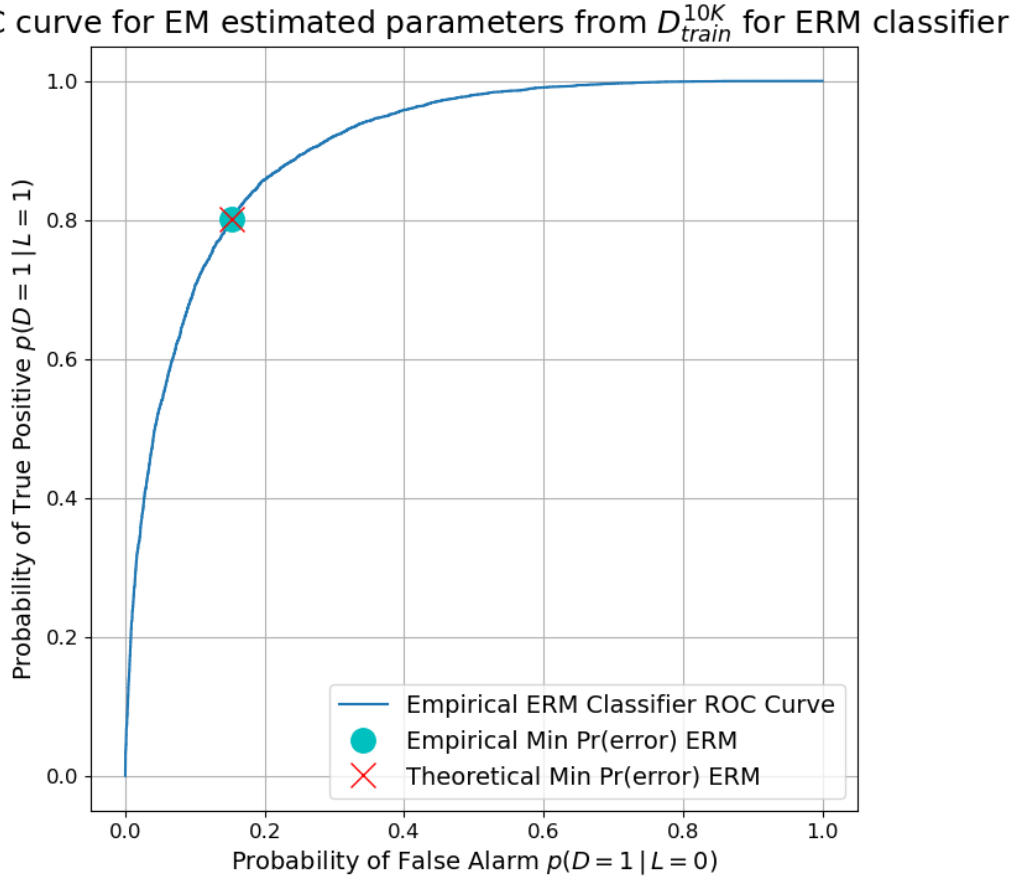
$$\pi_k = \frac{\sum_{n=1}^{N} \gamma(z_{nk})}{N}; \quad \mu_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n}{\sum_{n=1}^{N} \gamma(z_{nk})}; \quad \Sigma_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma(z_{nk})}$$

We then use the newly estimated values for the parameters to then calculate γ in the following E-Step and repeat the process until the parameters converge to a particular set of values.

The estimated values from the D_{train}^{10K} dataset using the **estimateMLE()** function in the Appendix, yielded:

```
MLE parameters for Class 0 are:
Component #1:
w1 = 0.4749241737656275
μ01 = [4.90459804 0.00437509]
Σ01 = [[4.10767903 0.035769  ]
 [0.035769   2.02239383]]
Component #2:
w2 = 0.5250758262343724
μ02 = [-0.00094393  4.01328161]
Σ02 = [[1.01308525 0.05529933]
 [0.05529933 3.06647151]]
Prior Value for Class 0: P(L=0) = 0.5915
MLE parameters for Class 1 are:
μ1 = [2.96621234 1.98195459]
Σ1 = [[2.02802519 0.02463674]
 [0.02463674 2.00481197]]
Prior value for Class 1: P(L=1) = 0.4085
```

Using these estimated parameters from the D_{train}^{10K} dataset, I generated the priors and class-conditional pdfs for Class 0 and Class 1, and approximated the min-$P(error)$ classifier and applied it the the $D_{validate}^{20K}$ dataset to achieve the following ROC curve and min-$P(error)$:



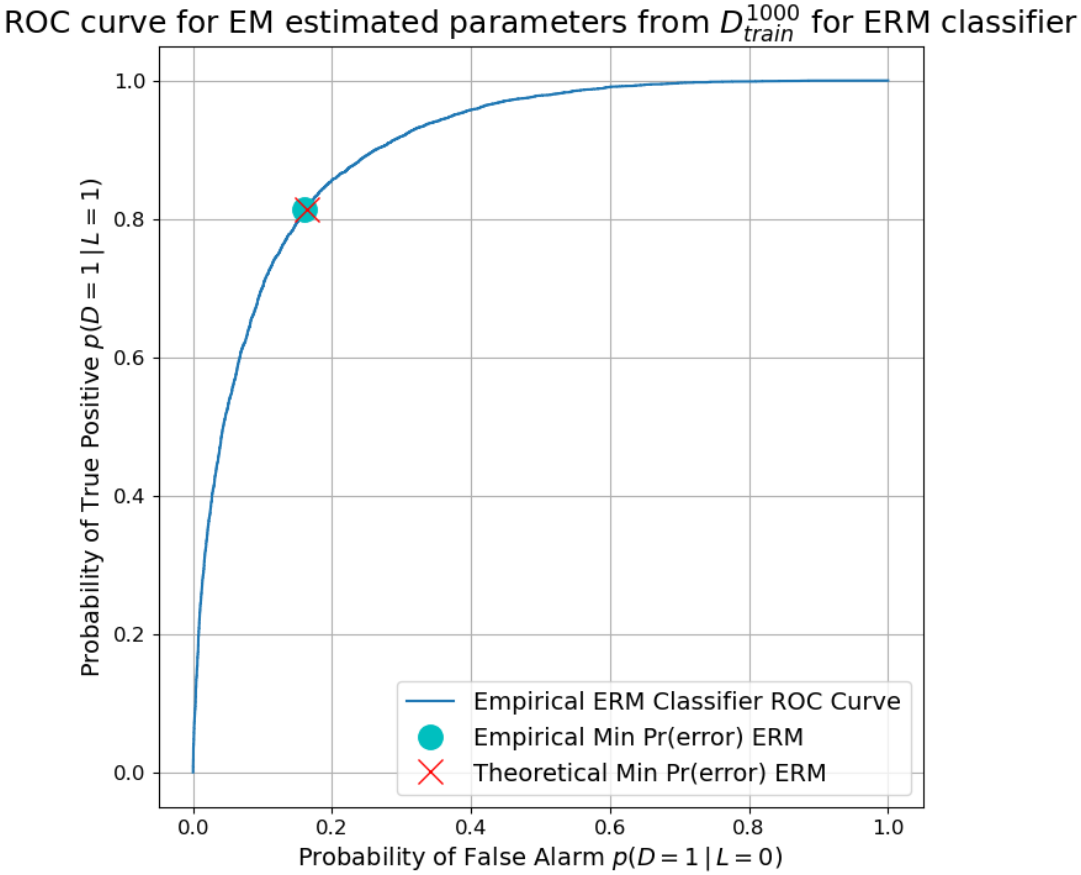
This approximation of the ERM classifer achieved an empirical min-$P(error) = 0.1708$

(b) Repeat Part (2a) using D_{train}^{1000} as the training dataset.

The estimated values from the D_{train}^{1000} dataset using the **estimateMLE()** function in the Appendix, yielded:

```
MLE parameters for Class 0 are:
Component #1:
w1 = 0.4848200279886821
μ01 = [5.05881979 0.11322488]
Σ01 = [[ 3.85366525 -0.02812353]
 [-0.02812353  1.96427345]]
Component #2:
w2 = 0.5151799720113177
μ02 = [0.03864031 4.00524791]
Σ02 = [[0.95376155 0.09399782]
 [0.09399782 2.81218181]]
Prior Value for Class 0: P(L=0) = 0.609
MLE parameters for Class 1 are:
μ1 = [3.05935063 1.84404866]
Σ1 = [[2.22422843 0.02738178]
 [0.02738178 2.02587521]]
Prior value for Class 1: P(L=1) = 0.391
```

Using these estimated parameters from the D_{train}^{1000} dataset, I generated the priors and class-conditional pdfs for Class 0 and Class 1, and approximated the min-$P(error)$ classifier and applied it the the $D_{validate}^{20K}$ dataset to achieve the following ROC curve and min-$P(error)$:



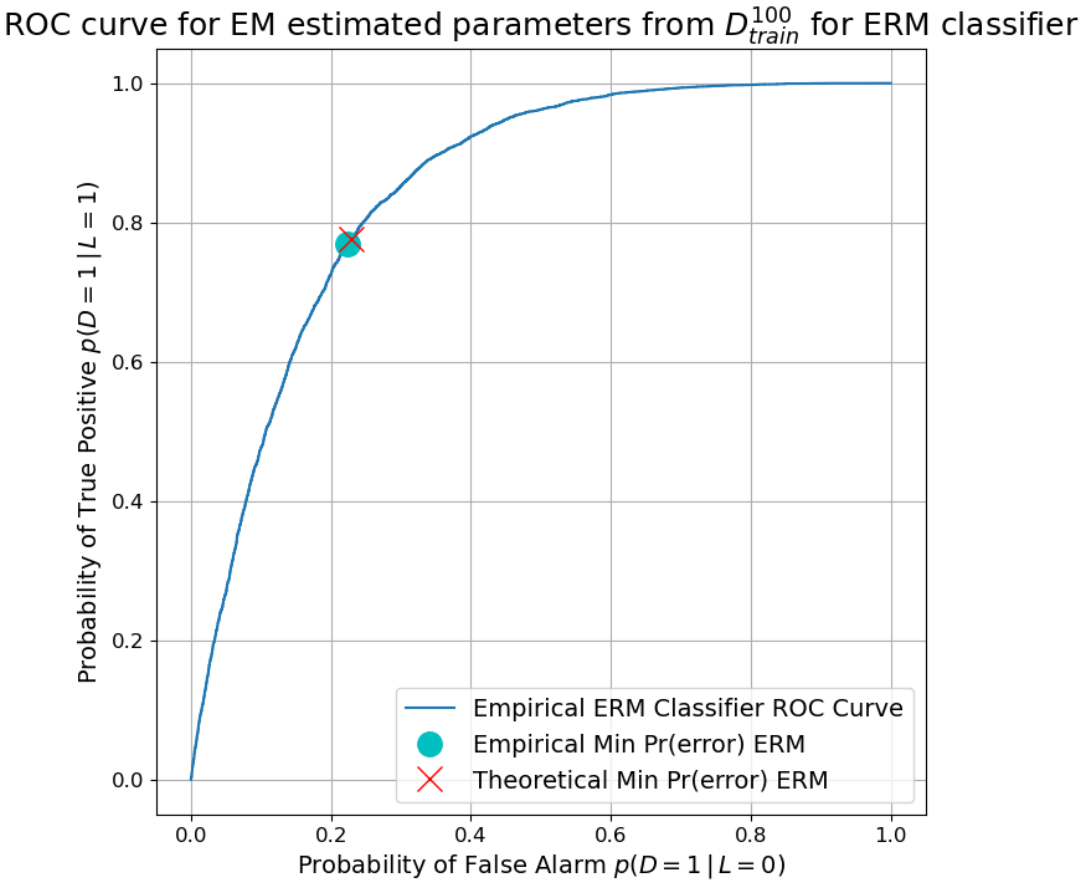
This approximation of the ERM classifer achieved an empirical min-$P(error) = 0.1713$

(c) Repeat Part (2a) using D_{train}^{100} as the training dataset. How does the performance of your approximate min-P(error) classifier change as the model parameters are estimated (trained) using fewer samples?

The estimated values from the D_{train}^{100} dataset using the **estimateMLE()** function in the Appendix, yielded:

```
MLE parameters for Class 0 are:
Component #1:
w1 = 0.6930129860831892
μ01 = [4.74803644 0.60413727]
Σ01 = [[ 8.63146683 -0.73835373]
 [-0.73835373  1.49016587]]
Component #2:
w2 = 0.3069870139168108
μ02 = [-0.02493863  4.94525416]
Σ02 = [[0.32599595 0.13970419]
 [0.13970419 0.89190335]]
Prior Value for Class 0: P(L=0) = 0.65
MLE parameters for Class 1 are:
μ1 = [2.6248776 1.9161448]
Σ1 = [[ 2.54231672 -0.39170124]
 [-0.39170124  1.82283037]]
Prior value for Class 1: P(L=1) = 0.35
```

Using these estimated parameters from the D_{train}^{100} dataset, I generated the priors and class-conditional pdfs for Class 0 and Class 1, and approximated the min-$P(error)$ classifier and applied it the the $D_{validate}^{20K}$ dataset to achieve the following ROC curve and min-$P(error)$:



This approximation of the ERM classifer achieved an empirical min-$P(error) = 0.2260$

From the repeated estimations of the ROC curves using the training sets, we can observe that the performance of the min-$P(error)$ classifier is dependent on the number of training samples we used before attempting to classifiy the $D_{validate}^{20K}$ dataset. We observe that the performance of the min-$P(error)$ classifier and ROC curves becomes worse when using less and less training samples.

When using the D_{train}^{10K} dataset, the ROC curve looks almost identical to the ROC curve generated from the $D_{validate}^{20K}$ dataset and generated an empirical min-$P(error)$ = 0.1708, which was identical in value to the min-$P(error)$ achieved with the D_{train}^{10K} dataset

When using the D_{train}^{1000} dataset, the ROC curve has looks worse than the previous and generated an empirical min-$P(error)$ = 0.1713, which is not that much of a increase in performance error

Lastly, when using the D_{train}^{100} dataset, the ROC curve has a less arch to its shape signifiying a weaker performance for this iteration of the ERM classifier. This is also evidence from the achieved empirical min-$P(error)$ = 0.2260.

This demonstrates that when we are data-starved for training samples for our ERM classifier we should expect a poorer performance than if we have an abundance of training samples.

Part 3: (20%)

(a) Using the maximum likelihood parameter estimation technique train a logistic-linear-function-based approximation of class label posterior function given a sample. As in part 2, repeat the training process for each of the three training sets to see the effect of training set sample count; use the validation set for performance assessment in each case. 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* or Python's *minimize*. Use the trained class-label-posterior approximations to classify validation samples to approximate the minimum-P(error) classification rule; estimate the probability of error that these three classifiers 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.

Our binary logistic regression model of class label posterior function can be specified as

$$p(l|\mathbf{x};\boldsymbol{\theta}) = g(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}))^l(1 - g(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x})))^{(1-l)}$$

where $g(z)$ represents the sigmoid function, l is the true class label (0 or 1), and $\phi(x)$ is a new feature matrix consisting of all polynomial combinations of the features $\mathbf{x}$ (this definition applies to both the linear case $\phi(\mathbf{x}) =$ degree 1 and quadratic case $\phi(\mathbf{x}) =$ degree 2)

We wish to express this class label posterior function in terms of its Log-Likelihood form given N samples in the dataset:

$$LL(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[l^{(i)}\log g\left(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}^{(i)})\right) + (1 - l^{(i)})\log\left(1 - g(\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}^{(i)}))\right)\right]$$

In order to derive the MLE for the model's parameters, θ, by minimizing the Negative-Log-Likelihood in order to derive $\theta_{MLE} = \mathbf{w}^*$

This expression is of the form

$$\mathbf{w}^* = \underset{\theta}{\operatorname{argmin}} \mathrm{NLL}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\left[l^{(i)} \log g\left(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}^{(i)})\right) + (1 - l^{(i)}) \log\left(1 - g(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}^{(i)}))\right)\right]$$

Then after obtaining our MLE parameter of $\mathbf{w}^*$, we derive our min-$P(error)$ decision rule using the sigmoid function's shape for our predictions, and generate the following decision rule:

$$g(\mathbf{w}^{*T}\boldsymbol{\phi}(\mathbf{x}^{(i)})) \geq 0.5 \rightarrow \text{Decide Class } 1 \rightarrow (D = 1)$$

$$g(\mathbf{w}^{*T}\boldsymbol{\phi}(\mathbf{x}^{(i)})) < 0.5 \rightarrow \text{Decide Class } 0 \rightarrow (D = 0)$$

These decisions reflect the situation where if comparing the posteriors, our decision is made from whichever class posterior is greater,

e.g. we choose Class 1 if $p(L = 1|x;\theta) > p(L = 0|x;\theta)$, else we would choose Class 0.

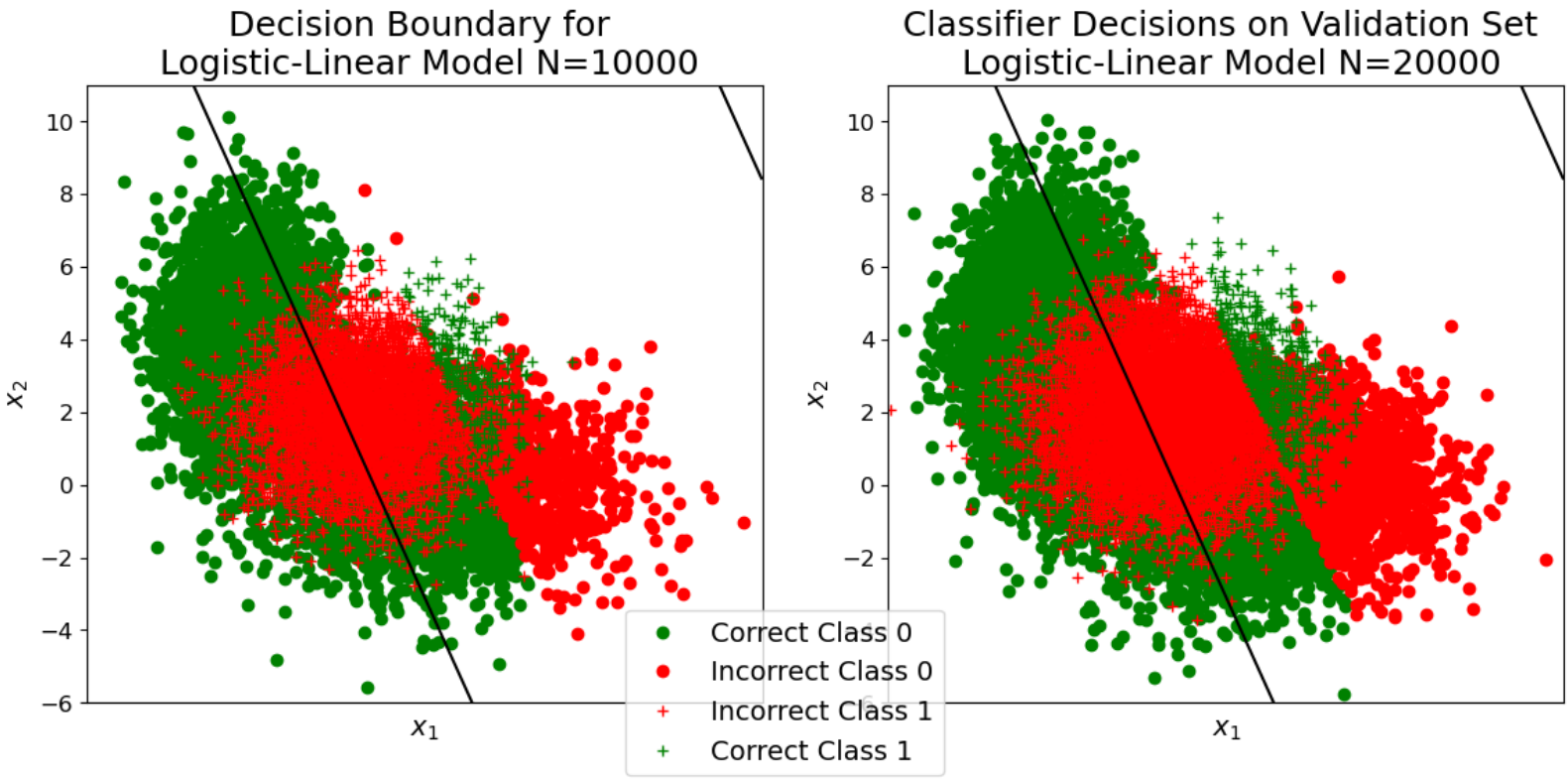
This corresponds to the inner argument of the sigmoid function specifying the decision rule by:

$$\mathbf{w}^{*T}\boldsymbol{\phi}(\mathbf{x}^{(i)}) \geq 0 \rightarrow \text{Decide Class } 1 \rightarrow (D = 1)$$

$$\mathbf{w}^{*T}\boldsymbol{\phi}(\mathbf{x}^{(i)}) < 0 \rightarrow \text{Decide Class } 0 \rightarrow (D = 0)$$

Using the Python code in Appendix: Question 1 Part 3(a), I was able to generate the following plots and their corresponding approximations of the min-$P(error)$ classifiers using a logistic-linear-function-based approximation of class label posteriors

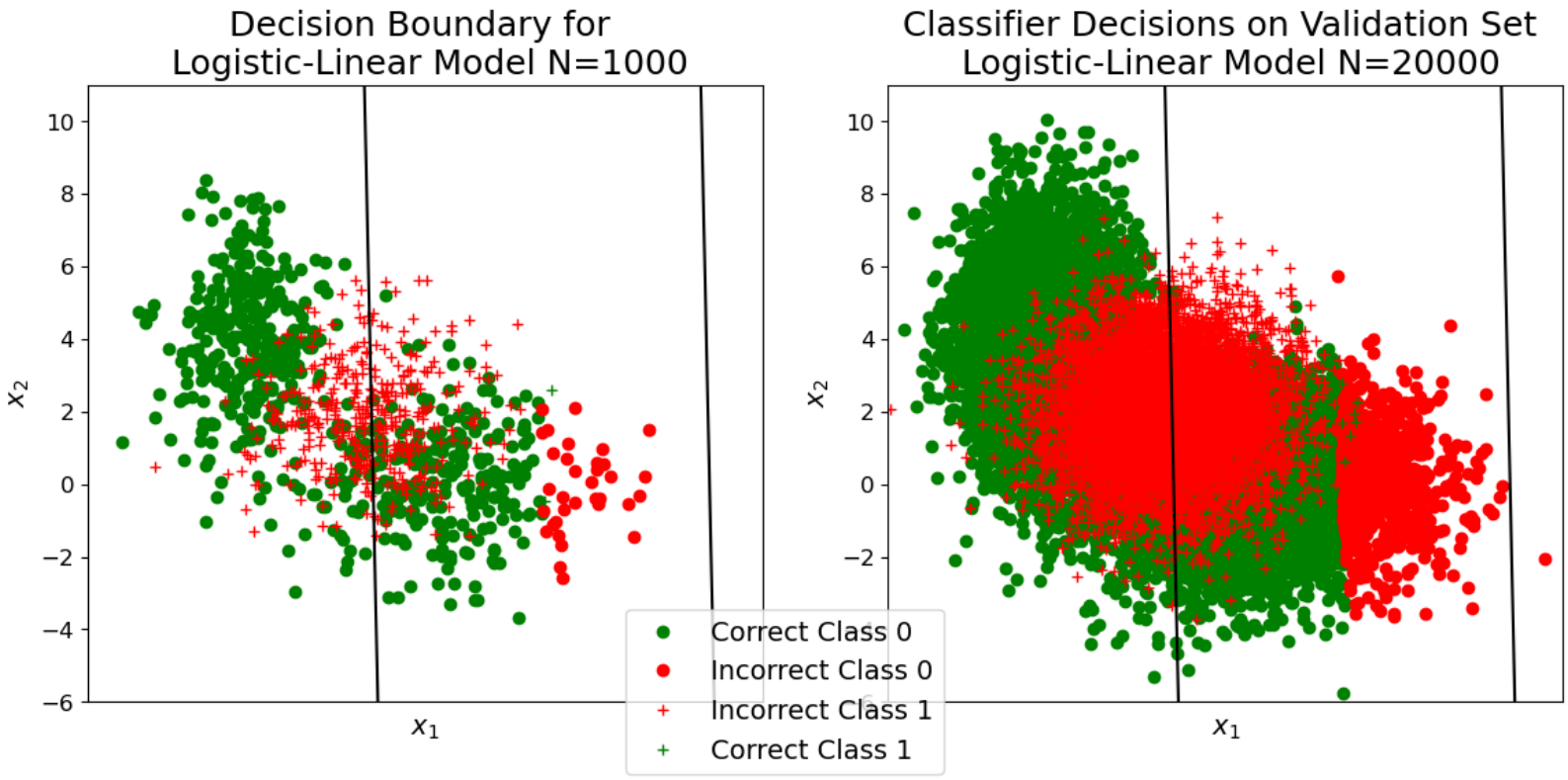
Starting with the Logistic-Linear Model trained using the D_{train}^{10K} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-0.842, 0.1353, 0.0558]^T$ with a min-$P(error) = 0.461$

The linear decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

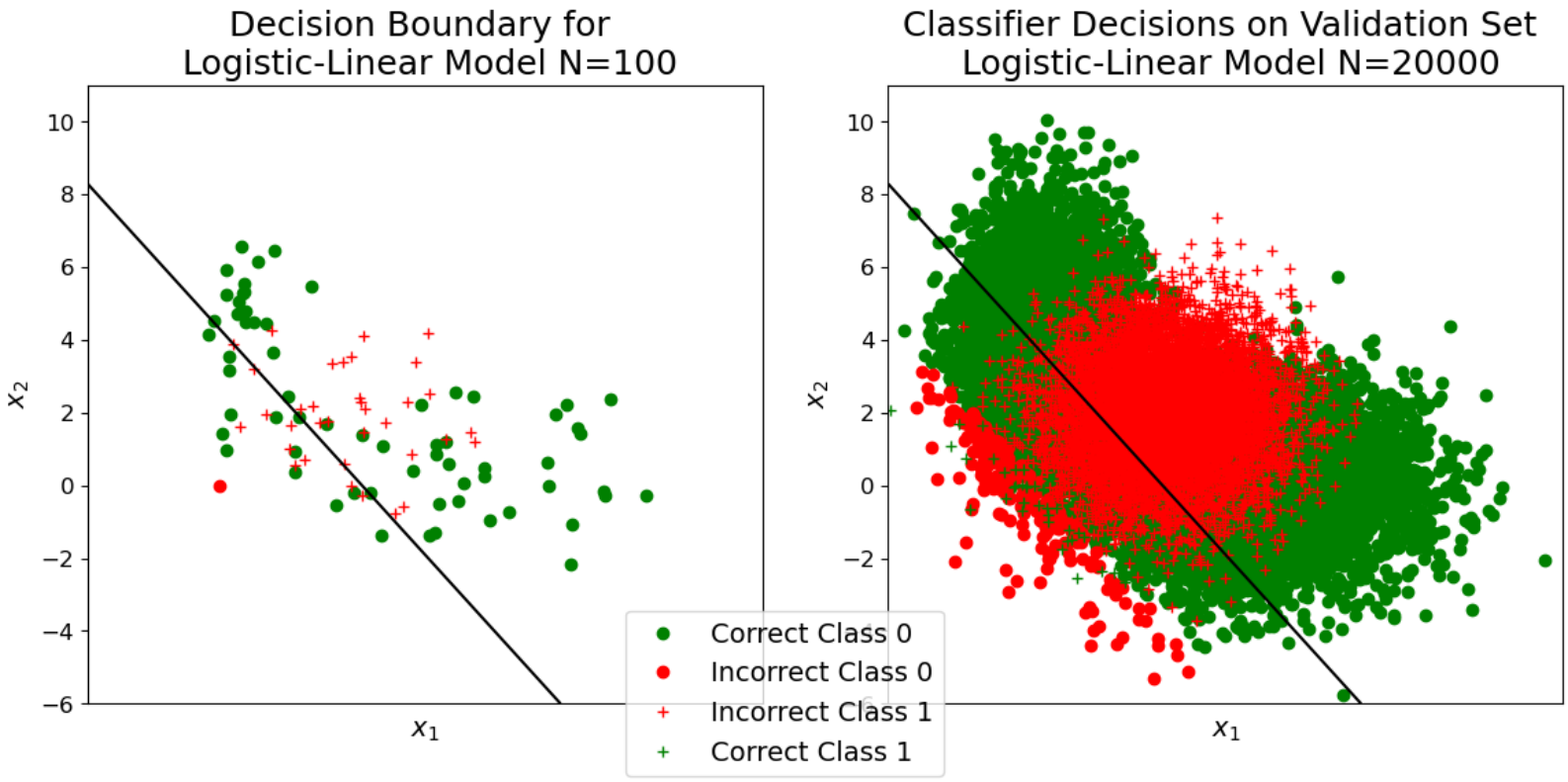
Next, the Logistic-Linear Model trained using the D_{train}^{1000} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-0.7111, 0.0956, 0.0019]^T$ with a min-$P(error) = 0.435$

Again, decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

Lastly, the Logistic-Linear Model trained using the D_{train}^{100} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-0.0247, -0.1306, -0.1089]^T$ with a min-$P(error) = 0.412$

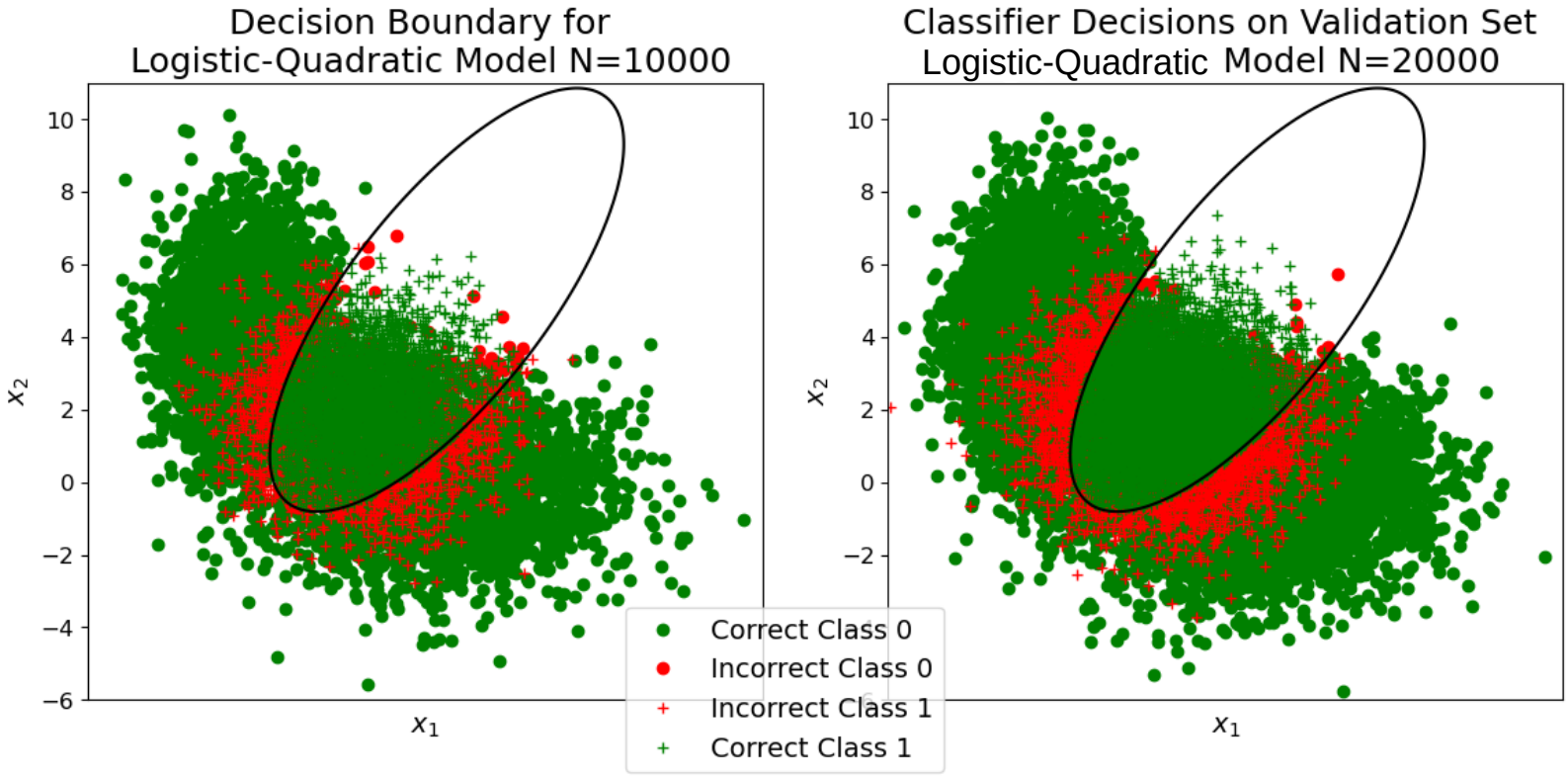
Again, decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

We observe that the performance of the Logistic-Linear models performed worse as the number of training samples increased and that the decision boundaries generated with less samples generated a better min-$P(error)$ classifier. Though, this seems to have just been circumstantial and no amount of data would generate a better linear decision boundary.

(b) Repeat the process described in Part (3a) using a logistic-quadratic-function-based approximation of class label posterior functions given a sample.

Again, using the Python code in Appendix: Question 1 Part 3(b), I was able to generate the following plots and their corresponding approximations of the min-$P(error)$ classifiers using a logistic-quadratic-function-based approximation of class label posteriors

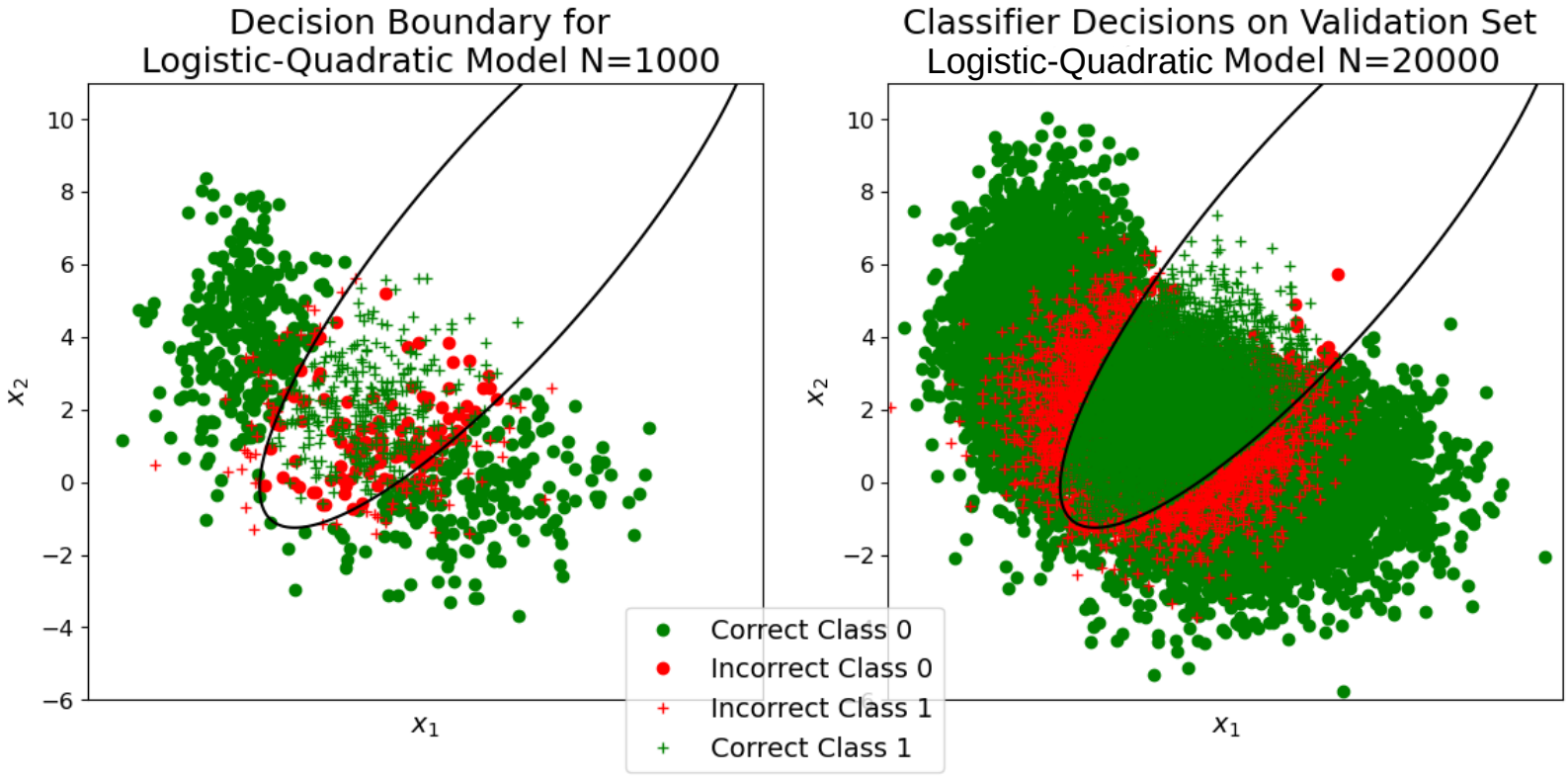
Starting with the Logistic-Quadratic Model trained using the D_{train}^{10K} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-0.7019, 1.2035, 0.0518, -0.2699, 0.3035, -0.1578]^T$ with a min-$P(error) = 0.174$

The quadratic decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

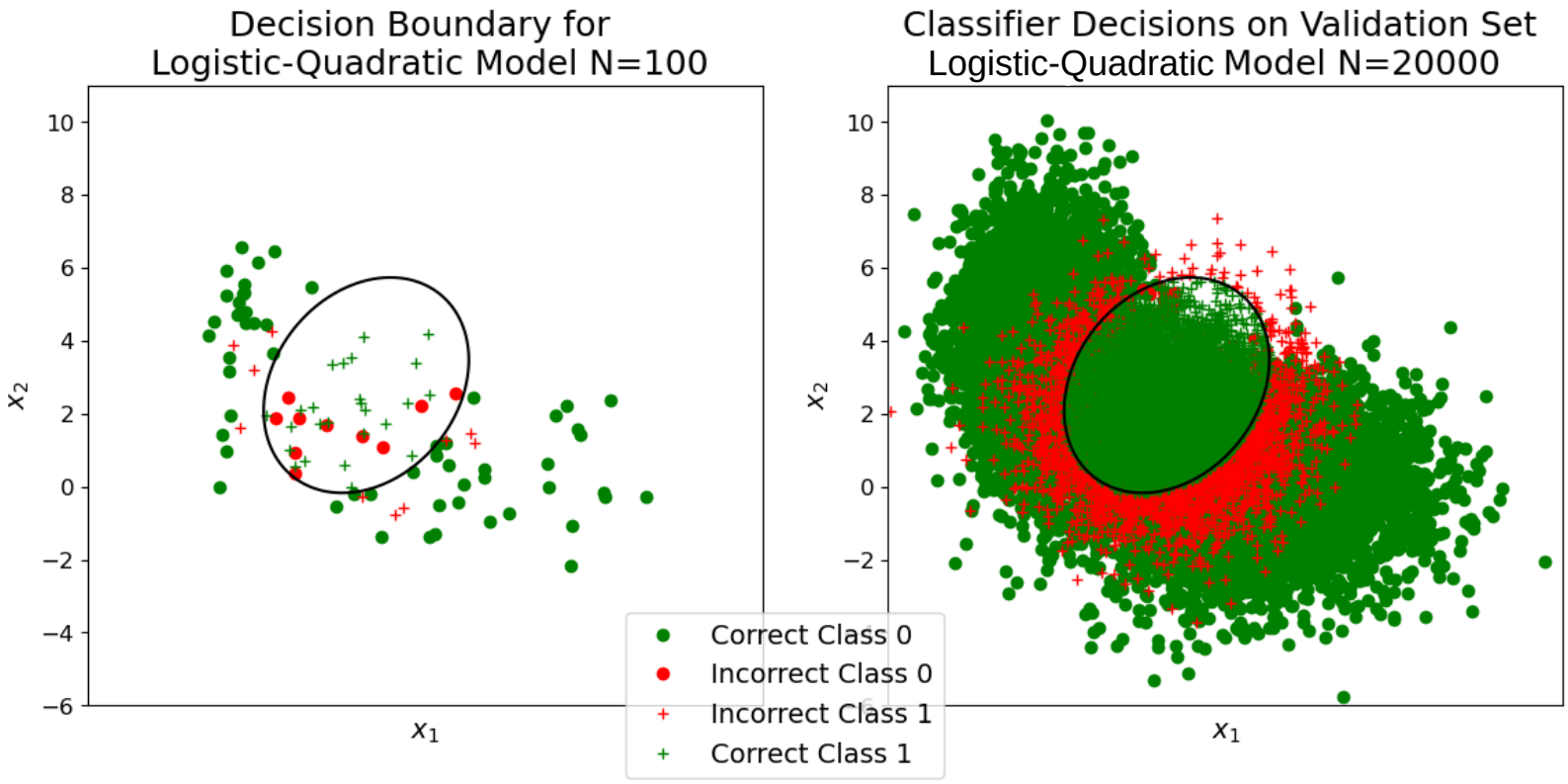
Next, the Logistic-Quadratic Model trained using the D_{train}^{1000} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-0.3098, 0.9914, -0.1569, -0.2379, 0.3282, -0.1527]^T$ with a min-$P(error) = 0.176$

Again, quadratic decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

Lastly, the Logistic-Quadratic Model trained using the D_{train}^{100} dataset:



I estimated the MLE parameter $\mathbf{w}^* = [-1.4173, 1.3144, 0.80943457 - 0.2683, 0.1106, -0.2056]^T$ with a min-$P(error) = 0.190$

Again, quadratic decision boundaries were derived from the ϕ-transformation of input features $\mathbf{x}$ and superimposed on the training and validation datasets

The keytakeaway for the Logistic-Quadratic model is that the min-$P(error)$ classifier performed between when the number of training samples increased and was superior to the logistic-linear model. Again, these decision boundaries were circumstantially-derived based on the random placement of training samples. Though, provided with more training samples, the logistic-quadratic model was able to draw decision boundaries that resulted in a fairly accurate classifer, but was still did not outperform the theoretically optimal model of Part 1.

Question 2 (30%)

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 $r_i = d_{Ti} + n_i$ for $i \in \{1, \ldots, K\}$, where $d_{Ti} = \|[x_T, y_T]^T - [x_i, y_i]^T\|$ is the true distance between the vehicle and the i^{th} reference point, and n_i is a zero mean Gaussian distributed measurement noise with known variance σ_i^2. The noise in each measurement is independent from the others. Assume that we have the following prior knowledge regarding the position of the vehicle:

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

where $[x, y]^T$ indicates a candidate position under consideration

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

We can express the optimization problem by solving for the values that maximize the Posterior objective function:

$$p([x_T, y_T]^T | r_i) = \prod_{i=1}^{K} p(r_i | [x_T, y_T]^T)\, P\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$$

The Posterior of the objective function is expressed as the likelihood that some noisy distance measurement is the true position of the vehicle, multiplied by the prior knowledge of the vehicle. And since each distance measurement is independent, we can express K distance measurements as the product of the per-base-station likelihoods.

Elaborating further, we can take the log-posterior which would simplify the right-hand side into the sum of the log-likelihoods added together with the log of the prior knowledge

$$\ln p([x_T, y_T]^T | r_i) = \sum_{i=1}^{K} \ln p(r_i | [x_T, y_T]^T) + \ln P\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$$

Simplifying the log of the prior knowledge:

$$\ln p([x_T, y_T]^T | r_i) = \sum_{i=1}^{K} \sqrt{(x_i - x_T)^2 + (y_i - y_T)^2} + n_i + \ln\left((2\pi\sigma_x\sigma_y)^{-1}\right) - \frac{1}{2}[x \quad y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1}\begin{bmatrix} x \\ y \end{bmatrix}$$

$$\ln p([x_T, y_T]^T | r_i) = \sum_{i=1}^{K} \sqrt{(x_i - x_T)^2 + (y_i - y_T)^2} + n_i - \ln(2\pi\sigma_x\sigma_y) - \frac{1}{2}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right)$$

$$LL = -\sum_{i=1}^{K} \sqrt{(x_i - x_T)^2 + (y_i - y_T)^2} + n_i + \ln(2\pi\sigma_x\sigma_y) + (1/2)\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right)$$

Simplifying and removing the constant terms, we have the Log-Likelihoods is:

$$LL = -\sum_{i=1}^{K} r_i + \ln(2\pi\sigma_x\sigma_y) + (1/2)\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right)$$

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 is get more certain? Explain how your contours justify your conclusions.

```
% Parameters for PDF of Vehicle Position
mu = [0 0]; sig_x = 0.25; sig_y = 0.25; sig_n = 0.3;
Sigma = [sig_x^2 0; 0 sig_y^2];
% Create Meshgrid for contour plots
xT = -2:.01:2; yT = -2:.01:2;
[XT,YT] = meshgrid(xT,yT);
XY = [XT(:) YT(:)];
% Generate PDF for prior knowledge of vehicle position
P = mvnpdf(XY,mu,Sigma);
P = reshape(P,length(yT),length(xT));
K = 1;
lb = [-0.5475 -0.5475];
ub = [0.5475 0.5475];
x0 = [0 0];
obj = @(x)LogLikelihood(K,x);
P1 = fmincon(obj,x0,[],[],[],[],lb,ub);
```

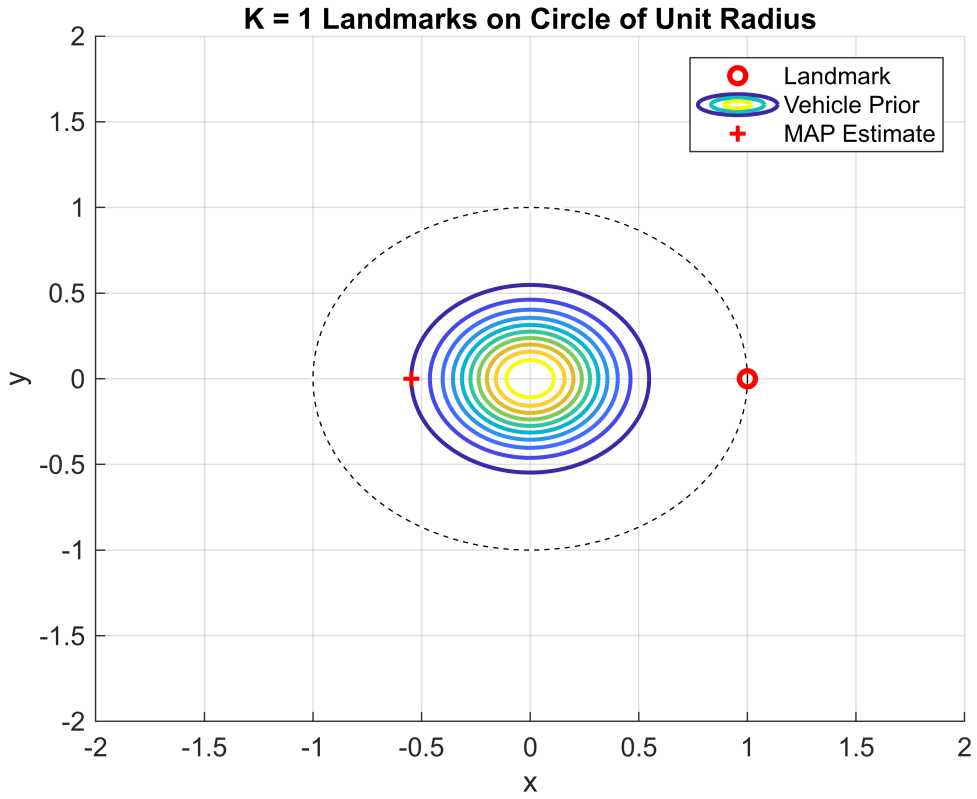
```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

```
figure(1)
[Kx1,Ky1] = Q2plotKContours(K,xT,yT,P);
plot(P1(1),P1(2),'r+','LineWidth',1.5), hold off, pause(1)
```

```
legend('Landmark','Vehicle Prior','MAP Estimate')
```



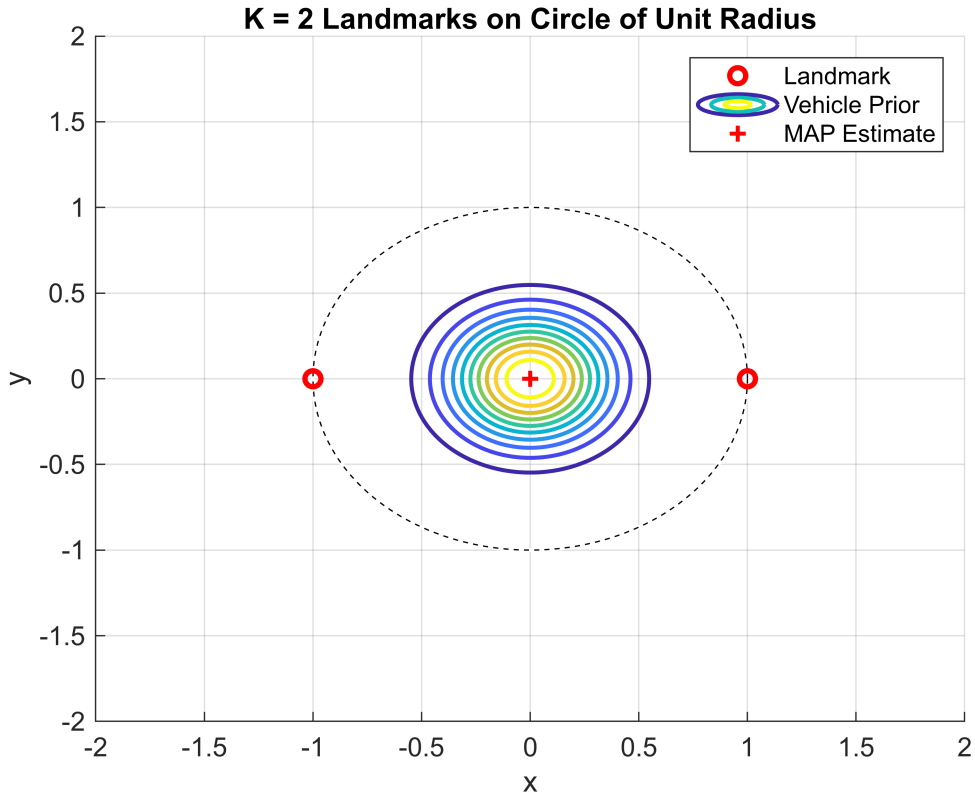

```
K = 2;
figure(2)
[Kx2,Ky2] = Q2plotKContours(K,xT,yT,P);
lb = [-0.5475 -0.5475];
ub = [0.5475 0.5475];
x0 = [0 0];
obj = @(x)LogLikelihood(K,x);
xopt = fmincon(obj,x0,[],[],[],[],lb,ub);
```

```
Initial point is a local minimum that satisfies the constraints.

Optimization completed because at the initial point, the objective function is non-decreasing
in feasible directions to within the value of the optimality tolerance, and
constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

```
plot(xopt(1),xopt(2),'r+','LineWidth',1.5), hold off, pause(1)
legend('Landmark','','Vehicle Prior','MAP Estimate')
```



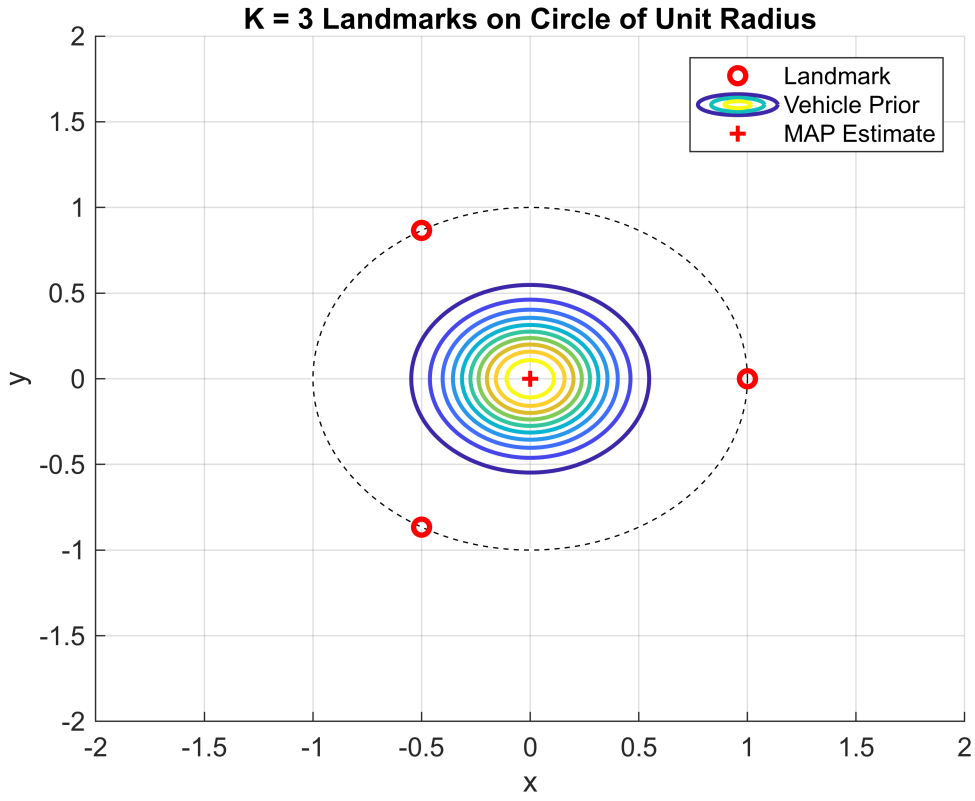

```
K = 3;
figure(3)
[Kx3,Ky3] = Q2plotKContours(K,xT,yT,P);
lb = [-0.5475 -0.5475];
ub = [0.5475 0.5475];
x0 = [0 0];
obj = @(x)LogLikelihood(K,x);
xopt = fmincon(obj,x0,[],[],[],[],lb,ub);
```

```
Initial point is a local minimum that satisfies the constraints.

Optimization completed because at the initial point, the objective function is non-decreasing
in feasible directions to within the value of the optimality tolerance, and
constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

```
plot(xopt(1),xopt(2),'r+','LineWidth',1.5), hold off, pause(1)
legend('Landmark','','','Vehicle Prior','MAP Estimate')
```



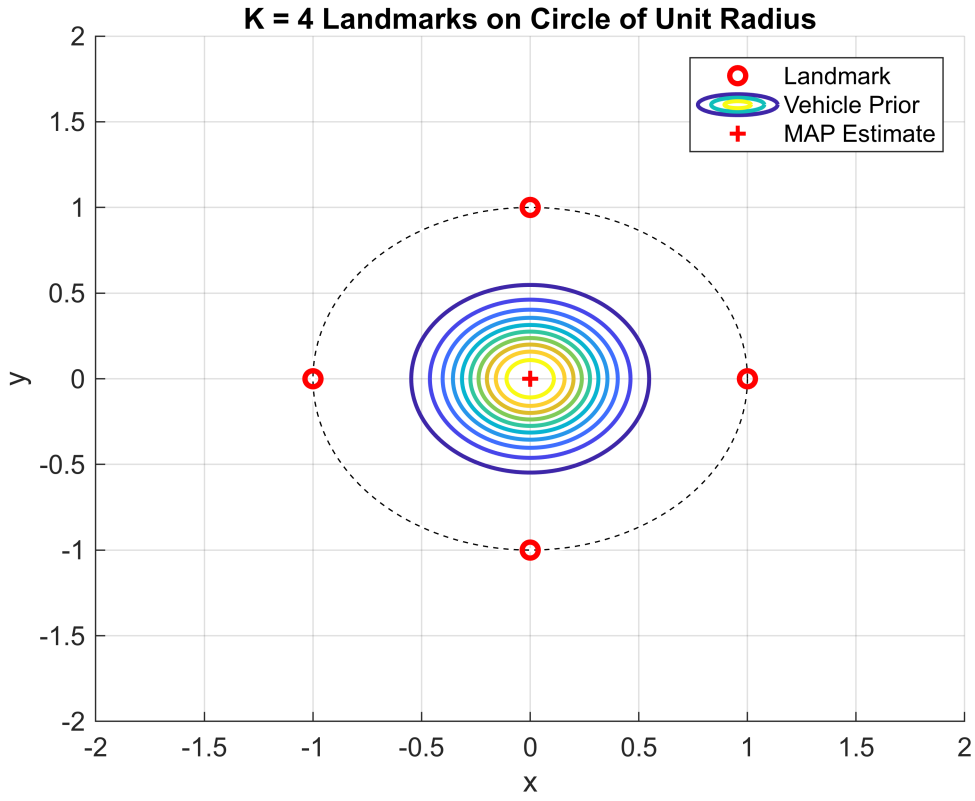

```
K = 4;
figure(4)
[Kx4,Ky4] = Q2plotKContours(K,xT,yT,P);
lb = [-0.5475 -0.5475];
ub = [0.5475 0.5475];
x0 = [0 0];
obj = @(x)LogLikelihood(K,x);
xopt = fmincon(obj,x0,[],[],[],[],lb,ub);
```

```
Initial point is a local minimum that satisfies the constraints.

Optimization completed because at the initial point, the objective function is non-decreasing
in feasible directions to within the value of the optimality tolerance, and
constraints are satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```

```
plot(xopt(1),xopt(2),'r+','LineWidth',1.5), hold off, pause(1)
legend('Landmark','','','','Vehicle Prior','MAP Estimate')
```



From the above plots, we visualize the MAP estimates on the Vehicle Prior contours as the number of landmark references increases from one to four. When there is only one landmark reference, the uncertainty of the vehicle's position is at the highest of each of the plots and guesses the vehicle's location to be along the outer edge of the prior knowledge contour.

As the number of landmark references increases to two, the uncertainty of the MAP estimate drops and finds the local minimum for the vehicle's position to be centered at the origin. This uncertainty steadily decreases as the number of landmark references increases, but remains nonzero due to the additive noise for each of the K distance measurements. Since the prior knowledge assumes the form of a white Gaussian distribution centered at the origin, the local minimum achieved is at the origin of the prior knowledge which confirms the convergence for the true vehicle position to be at the origin for $K > 1$ landmark references.

Question 3 (20%)

Problem 2.13 from Duda-Hart textbook:

Section 2.4

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

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

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

Let's first define the expression for conditional risk for c number of classes:

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

where $i, j = 1, \ldots, c$ and α_i is the action we take that minimizes the conditional risk $R(\alpha_i|\mathbf{x})$

The conditional risk for the action of rejection can be defined as:

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

Then including the expected loss term λ_s into the expression for conditional risk

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

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

$$= \lambda_s(1 - P(\omega_i|\mathbf{x})$$

Our decision based on the expected losses would be to choose the option of rejections if and only if

$$\lambda_r < \lambda_s(1 - P(\omega_i|\mathbf{x})) \qquad \forall \ i = 1, \ldots, c$$

where the loss of rejection is the true minimal risk decision for all classes. Rearranging this equation, we find

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

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

Dividing by -1 changes the sign, and we end up with

$$P(\omega_i|\mathbf{x}) < 1 - \frac{\lambda_r}{\lambda_s} \qquad \forall \ i = 1, \ldots, c$$

So, all the class posteriors will be under this threshold, which then we should choose rejection. Else, we choose

$$\lambda_s(1 - P(\omega_i|\mathbf{x})) < \lambda_s(1 - P(\omega_j|\mathbf{x})) \qquad \forall \ j = 1, \ldots, c, j \neq i$$

which after some simple algebraic opertations is equal to

$P(\omega_i|\mathbf{x}) > P(\omega_j|\mathbf{x})$

which means we choose the larger class posterior probability to achieve the minimum error rate for all j

Again, looking at our decision rule for rejection:

$$P(\omega_i|\mathbf{x}) < 1 - \frac{\lambda_r}{\lambda_s} \qquad \forall \ i = 1, \ldots, c$$

For the case when there is zero-loss associated with rejection $\rightarrow \lambda_r = 0$, the right-hand side of the above equality equates the class posteriors to being less than one, which will always be the case since the posterior probabilities will always be less than one. In this instance, if $\lambda_r = 0$, then we will always choose rejection for minimum risk decision.

For the case when loss from rejection is greater than loss from substitution error $\rightarrow \ \lambda_r > \lambda_s$, the right-hand side of the equality will become a negative value and the rejection expression will never be valid, and thus we will never choose rejection.

Appendix: Homework Code & Citations

Note:

For Problem 1: All functions and expressions were written/performed in Python due to available online sources I cited and expansive libraries for Data Science functions in Python. Problem 2 was done using MATLAB and functions used are listed below.

Question 1 Code:

Part 1:

Data-Generating and Evaluation Functions:

```
def HW3generateDataQ1(N, pdf_params):
    # Use mean vectors to determine dimensionality of data
    d = pdf_params['mu'].shape[1]
    # Decide randomly which samples come from one of the Gaussian Components
    u = np.random.rand(N)
    # Determine thresholds based on the mixture weights/priors for the GMM
    thresh = np.cumsum(np.append(pdf_params['weights'].dot(pdf_params['priors'][0]), pdf_params['priors'][1]))
    thresh = np.insert(thresh, 0, 0)
    # Since only 2 classes, we can use logical indexing to decide between Class 0 and Class 1
```

```
    labels = u >= pdf_params['priors'][0]
    X = np.zeros((N, d))
    numGaussians = len(pdf_params['mu'])
    for i in range(1, numGaussians+1):
        # Get randomly sampled indices for Gaussian component
        idx = np.argwhere((thresh[i-1] <= u) & (u <= thresh[i]))[:, 0]
        X[idx, :] =  mvn.rvs(pdf_params['mu'][i-1], pdf_params['Sigma'][i-1], len(idx))

    return X, labels

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

    # Use gamma values that will account for every possible classification split
    # Use epsilon to avoid values of lower and upper bounds equating to zero
    gammas = ([sortedScore[0] - float_info.epsilon] +
              sortedScore +
              [sortedScore[-1] + float_info.epsilon])
    # Calculate the decision label for each observation for each gamma
    decisions = [scores >= g for g in gammas]

    # Find indices where FPs decisions made
    idxFP = [np.argwhere((d == 1) & (labels == 0)) for d in decisions]
    # Compute FPR
    FPR = [len(inds) / N_labels[0] for inds in idxFP]
    # Find indices where TPs decisions made
    idxTP = [np.argwhere((d == 1) & (labels == 1)) for d in decisions]
    # Compute TPR
    TPR = [len(inds) / N_labels[1] for inds in idxTP]

    # ROC dictionary to store TPR and FPR
    roc = {}
    roc['FPR'] = np.array(FPR)
    roc['TPR'] = np.array(TPR)

    return roc, gammas
def estimateClassMetrics(predictions, labels, N_labels):
    # Get indices and probability estimates of the four decision scenarios:
    # (true negative, false positive, false negative, true positive)
    classMetrics = {}

    # True Negative Probability Rate
    classMetrics['TN'] = np.argwhere((predictions == 0) & (labels == 0))
    classMetrics['TNR'] = len(classMetrics['TN']) / N_labels[0]
    # False Positive Probability Rate
    classMetrics['FP'] = np.argwhere((predictions == 1) & (labels == 0))
    classMetrics['FPR'] = len(classMetrics['FP']) / N_labels[0]
    # False Negative Probability Rate
    classMetrics['FN'] = np.argwhere((predictions == 0) & (labels == 1))
    classMetrics['FNR'] = len(classMetrics['FN']) / N_labels[1]
    # True Positive Probability Rate
    classMetrics['TP'] = np.argwhere((predictions == 1) & (labels == 1))
    classMetrics['TPR'] = len(classMetrics['TP']) / N_labels[1]
    return classMetrics
def estimateERMscores(X, pdf):
```

```
    # Compute class conditional likelihoods to express ratio test, where ratio is discriminant score
    # Class-Conditional pdf for Class 0 containing 2 Gaussian components
    pxL0 = (pdf['weights'][0]*mvn.pdf(X, pdf['mu'][0], pdf['Sigma'][0])
                  + pdf['weights'][1]*mvn.pdf(X, pdf['mu'][1], pdf['Sigma'][1]))
    # Class-Conditional pdf for Class 1 -> single Gaussian pdf
    pxL1 = mvn.pdf(X, pdf['mu'][2], pdf['Sigma'][2])
    # ERM scores taken by log-likelihood ratio or subtraction, to be evaluated with log(gamma)
    ermScores = np.log(pxL1) - np.log(pxL0)

return ermScores
```

Plotting Datasets

```
# Define dictionary for holding parameters of Gaussian Mixture Model
pdf = {}
# Priors for Class 0 = 0.6 and Class 1 = 0.4
pdf['priors'] = np.array([0.6, 0.4])
# Weights for the multicomponent gaussian of class 0
pdf['weights'] = np.array([0.5, 0.5])
# Means for GMM
pdf['mu'] = np.array([[5, 0],
                      [0, 4],
                      [3, 2]])
# Covariance matrices for GMM
pdf['Sigma'] = np.array([[[4, 0],
                          [0, 2]],
                         [[1, 0],
                          [0, 3]],
                         [[2, 0],
                          [0, 2]]])

# Number of training samples for each dataset
D_train = [100, 1000, 10000]
numDatasets = len(D_train)
# Making lists for the samples and labels
X_train = []
labels_train = []
N_labels_train = []
# Generate each training dataset using data generation function
D100, D100_labels  = HW3generateDataQ1(D_train[0], pdf)
D1K,  D1K_labels   = HW3generateDataQ1(D_train[1], pdf)
D10K, D10K_labels  = HW3generateDataQ1(D_train[2], pdf)
# Generate sample validation dataset using data generation function
D_validate = 20000
D20K, D20K_labels = HW3generateDataQ1(D_validate, pdf)
# Count up the number of samples per class in each dataset
# D100 samples
X_train.append(D100)
labels_train.append(D100_labels)
N_labels_train.append(np.array((sum(D100_labels == 0), sum(D100_labels == 1))))
# D1K samples
X_train.append(D1K)
labels_train.append(D1K_labels)
N_labels_train.append(np.array((sum(D1K_labels == 0), sum(D1K_labels == 1))))
# D10K samples
X_train.append(D10K)
labels_train.append(D10K_labels)
N_labels_train.append(np.array((sum(D10K_labels == 0), sum(D10K_labels == 1))))
```

```
# Print out and display the # of true class labels in each training dataset
print('The number of true class labels for each dataset are :')
print(N_labels_train)
# D20K samples
Nl_valid = np.array((sum(D20K_labels == 0), sum(D20K_labels == 1)))
# Print out and display the # of true class labels in validation dataset
print('The number of true class labels for validation dataset are : ')
print(Nl_valid)
# Plot the original data and their true labels
fig, ax = plt.subplots(2, 2, figsize=(11, 11))
# Plot D100 train with correct class labels
ax[0, 0].plot(D100[D100_labels==0, 0], D100[D100_labels==0, 1], 'bo', label="Class 0")
ax[0, 0].plot(D100[D100_labels==1, 0], D100[D100_labels==1, 1], 'r+', label="Class 1")
ax[0, 0].set_title(r"Training Dataset: $D^{%d}_{train}$" % (D_train[0]))
ax[0, 0].set_xlabel(r"$x_1$")
ax[0, 0].set_ylabel(r"$x_2$")
ax[0, 0].legend()
# Plot D10K train with correct class labels
ax[0, 1].plot(D1K[D1K_labels==0, 0], D1K[D1K_labels==0, 1], 'bo', label="Class 0")
ax[0, 1].plot(D1K[D1K_labels==1, 0], D1K[D1K_labels==1, 1], 'r+', label="Class 1")
ax[0, 1].set_title(r"Training Dataset: $D^{%d}_{train}$" % (D_train[1]))
ax[0, 1].set_xlabel(r"$x_1$")
ax[0, 1].set_ylabel(r"$x_2$")
ax[0, 1].legend()
# Plot D10K train with correct class labels
ax[1, 0].plot(D10K[D10K_labels==0, 0], D10K[D10K_labels==0, 1], 'bo', label="Class 0")
ax[1, 0].plot(D10K[D10K_labels==1, 0], D10K[D10K_labels==1, 1], 'r+', label="Class 1")
ax[1, 0].set_title(r"Training Dataset: $D^{%d}_{train}$" % (D_train[2]))
ax[1, 0].set_xlabel(r"$x_1$")
ax[1, 0].set_ylabel(r"$x_2$")
ax[1, 0].legend()
# Plot D20K validate with correct class labels
ax[1, 1].plot(D20K[D20K_labels==0, 0], D20K[D20K_labels==0, 1], 'bo', label="Class 0")
ax[1, 1].plot(D20K[D20K_labels==1, 0], D20K[D20K_labels==1, 1], 'r+', label="Class 1")
ax[1, 1].set_title(r"Validation Dataset: $D^{%d}_{validate}$" % (D_validate))
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(D20K[:,0])), ceil(np.max(D20K[:,0])))
x2_valid_lim = (floor(np.min(D20K[:,1])), ceil(np.max(D20K[:,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()
```

ROC Curve and min P(error)

```
# Generate the class-conditional likelihoods and compute ERM scores
ermScores = estimateERMscores(D20K, pdf)
# Estimate the ROC curve
roc, gammas = estimateROC(ermScores,D20K_labels,Nl_valid)
# Plot the ROC curve with the empirical value for min-P(error)
fig_roc, ax_roc = plt.subplots(figsize=(8, 8));
ax_roc.plot(roc['FPR'], roc['TPR'], label="Empirical ERM Classifier ROC Curve")
ax_roc.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
ax_roc.set_title('ROC curve for ERM classifier')
```

```
# ROC returns FPR vs TPR, but needs False Negative Rates, which is 1 - TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
#Perror_empirical = np.array((roc['FPR'], 1 - roc['TPR'])).T.dot(Nl_valid / D_validate)
N_class0 = Nl_valid[0]/D_validate
N_class1 = Nl_valid[1]/D_validate
Perror_empirical = roc['FPR']*N_class0 + (1-roc['TPR'])*N_class1
# Min prob error for the empirical gamma values
min_Perror_empirical = np.min(Perror_empirical)
min_idx_empirical = np.argmin(Perror_empirical)
# Compute theoretical gamma
gamma_theoretical = pdf['priors'][0] / pdf['priors'][1]
MAP_decisions = ermScores >= np.log(gamma_theoretical)
# Calculate the estimates for TPR, FNR, FPR, and TNR
MAP_metrics = estimateClassMetrics(MAP_decisions, D20K_labels, Nl_valid)
# Compute probability of error using FPR and FNR and class priors
min_prob_error_map = np.array((MAP_metrics['FPR'] * pdf['priors'][0] + MAP_metrics['FNR'] * pdf['priors'][1]))
# Plot theoretical and empirical
ax_roc.plot(roc['FPR'][min_idx_empirical], roc['TPR'][min_idx_empirical], 'co', label="Empirical Min P(error)
            markersize=14)
ax_roc.plot(MAP_metrics['FPR'], MAP_metrics['TPR'], 'rx', label="Theoretical Min P(error) ERM", markersize=14)
ax_roc.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
ax_roc.set_title('ROC curve for ERM classifier')
plt.grid(True)
plt.legend()
plt.show()
print("Min Empirical Pr(error) for ERM = {:.4f}".format(min_Perror_empirical))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas[min_idx_empirical])))

print("Min Theoretical Pr(error) for ERM = {:.4f}".format(min_prob_error_map))
print("Min Theoretical Gamma = {:.3f}".format(gamma_theoretical))
```

Part 2:

Functions:

```
def estimateMLE(X, labels, N_labels):
    # Use logical Indexing to extract Class 0 and Class 1 Samples
    idx0 = labels < 1
    idx1 = labels > 0
    # Use indices above to separate Dataset into Class 0 and Class 1 samples
    C0 = X[idx0]
    C1 = X[idx1]
    # Create dictionaries for each class to store MLE parameters for each
    X0 = {}
    X1 = {}
    # For Class 1, find Sample Mean, Sample Covarience and # samples to calculate Class Prior for MLE paramete
    X1['mu'] = np.mean(C1,axis=0)
    X1['Sigma'] = np.cov(C1.T)
    X1['prior'] = len(C1)/N_labels
    # Use EM Algorithm and GaussianMixture object to estimate the MLE parameters for Class 0
    gmm = GaussianMixture(n_components = 2,n_init = 10, init_params='random_from_data').fit(C0)
    X0['mu'] = gmm.means_
    X0['Sigma'] = gmm.covariances_
    X0['weights'] = gmm.weights_
    X0['prior'] = np.sum(gmm.weights_*(len(C0)/N_labels))
    # Print MLE parameters to console
    print("MLE parameters for Class 0 are: ")
```

```
    print("Component #1: \nw1 =",X0['weights'][0],"\n\u03BC01 =",X0['mu'][0], "\n\u03A301 =",X0['Sigma'][0])
    print("Component #2: \nw2 =",X0['weights'][1],"\n\u03BC02 =",X0['mu'][1], "\n\u03A302 =",X0['Sigma'][1])
    print("Prior Value for Class 0: P(L=0) =", round(X0['prior'],4))
    print("MLE parameters for Class 1 are: ")
    print("\u03BC1 =",X1["mu"], "\n\u03A31 =",X1['Sigma'],"\nPrior value for Class 1: P(L=1) =",round(X1['pric

    return X0, X1
def estimateERMscoresV2(X, A, B):
    # Compute class conditional likelihoods to express ratio test, where ratio is discriminant score
    # Recall that there is a mixture weighting for class 0!
    pxL0 = (A['weights'][0]*mvn.pdf(X, A['mu'][0], A['Sigma'][0])
                  + A['weights'][1]*mvn.pdf(X, A['mu'][1], A['Sigma'][1]))
    pxL1 = mvn.pdf(X, B['mu'], B['Sigma'])
    # Class conditional log likelihoods equate to decision boundary log gamma in the 0-1 loss case
    ermScores = np.log(pxL1) - np.log(pxL0)

    return ermScores
```

Part 2(a):

```
# Use estimateMLE function to calculate the MLE parameters for D10K dataset using just the samples
Class0_10K, Class1_10K = estimateMLE(D10K, D10K_labels, D_train[2])

# Generate the class-conditional likelihoods and compute ERM scores
ermScores_10K = estimateERMscoresV2(D20K, Class0_10K, Class1_10K)
# Estimate the ROC curve
roc_10K, gammas_10K = estimateROC(ermScores_10K,D20K_labels,Nl_valid)
# Plot the ROC curve with the empirical value for min-P(error)
fig_roc_10K, ax_roc_10K = plt.subplots(figsize=(8, 8));
ax_roc_10K.plot(roc_10K['FPR'], roc_10K['TPR'], label="Empirical ERM Classifier ROC Curve")
ax_roc_10K.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_10K.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
# ROC returns FPR vs TPR, but needs False Negative Rates, which is 1 - TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
Perror_empirical_10K = np.array((roc_10K['FPR'], 1 - roc_10K['TPR'])).T.dot(Nl_valid / D_validate)
# Min prob error for the empirical gamma values
min_Perror_empirical_10K = np.min(Perror_empirical_10K)
min_idx_empirical_10K = np.argmin(Perror_empirical_10K)
# Compute theoretical gamma
gamma_theoretical_10K = Class0_10K['prior'] / Class1_10K['prior']
MAP_decisions_10K = ermScores_10K > np.log(gamma_theoretical)
# Calculate the estimates for TPR, FNR, FPR, and TNR
MAP_metrics_10K = estimateClassMetrics(MAP_decisions_10K, D20K_labels, Nl_valid)
# Compute probability of error using FPR and FNR and class priors
min_prob_error_map_10K = np.array((MAP_metrics_10K['FPR'] * Class0_10K['prior'] + MAP_metrics_10K['FNR'] * Cla
# Plot theoretical and empirical
ax_roc_10K.plot(roc_10K['FPR'][min_idx_empirical_10K], roc_10K['TPR'][min_idx_empirical_10K], 'co', label="Emp
            markersize=14)
ax_roc_10K.plot(MAP_metrics_10K['FPR'], MAP_metrics_10K['TPR'], 'rx', label="Theoretical Min Pr(error) ERM", m
ax_roc_10K.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_10K.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
ax_roc_10K.set_title('ROC curve for EM estimated parameters from $D^{10K}_{train}$ for ERM classifier')
plt.grid(True)
plt.legend()
plt.show()
print("Min Empirical P(error) for ERM = {:.4f}".format(min_Perror_empirical_10K))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas_10K[min_idx_empirical_10K])))
```

```
print("Min Theoretical P(error) for ERM = {:.4f}".format(min_prob_error_map_10K))
print("Min Theoretical Gamma = {:.3f}".format(gamma_theoretical))
```

Part2(b):

```
# Use estimateMLE function to calculate the MLE parameters for D1000 dataset using just the samples
Class0_1K, Class1_1K = estimateMLE(D1K, D1K_labels, D_train[1])

# Generate the class-conditional likelihoods and compute ERM scores
ermScores_1K = estimateERMscoresV2(D20K, Class0_1K, Class1_1K)
# Estimate the ROC curve
roc_1K, gammas_1K = estimateROC(ermScores_1K,D20K_labels,Nl_valid)
# Plot the ROC curve with the empirical value for min-P(error)
fig_roc_1K, ax_roc_1K = plt.subplots(figsize=(8, 8));
ax_roc_1K.plot(roc_1K['FPR'], roc_1K['TPR'], label="Empirical ERM Classifier ROC Curve")
ax_roc_1K.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_1K.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
# ROC returns FPR vs TPR, but needs False Negative Rates, which is 1 - TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
Perror_empirical_1K = np.array((roc_1K['FPR'], 1 - roc_1K['TPR'])).T.dot(Nl_valid / D_validate)
# Min prob error for the empirical gamma values
min_Perror_empirical_1K = np.min(Perror_empirical_1K)
min_idx_empirical_1K = np.argmin(Perror_empirical_1K)
# Compute theoretical gamma
gamma_theoretical_1K = Class0_1K['prior'] / Class1_1K['prior']
MAP_decisions_1K = ermScores_1K > np.log(gamma_theoretical)
# Calculate the estimates for TPR, FNR, FPR, and TNR
MAP_metrics_1K = estimateClassMetrics(MAP_decisions_1K, D20K_labels, Nl_valid)
# Compute probability of error using FPR and FNR and class priors
min_prob_error_map_1K = np.array((MAP_metrics_1K['FPR'] * Class0_1K['prior'] + MAP_metrics_1K['FNR'] * Class1_
# Plot theoretical and empirical
ax_roc_1K.plot(roc_1K['FPR'][min_idx_empirical_1K], roc_1K['TPR'][min_idx_empirical_1K], 'co', label="Empirica
            markersize=14)
ax_roc_1K.plot(MAP_metrics_1K['FPR'], MAP_metrics_1K['TPR'], 'rx', label="Theoretical Min Pr(error) ERM", mark
ax_roc_1K.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_1K.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
ax_roc_1K.set_title('ROC curve for EM estimated parameters from $D^{1000}_{train}$ for ERM classifier')
plt.grid(True)
plt.legend()
plt.show()
print("Min Empirical P(error) for ERM = {:.4f}".format(min_Perror_empirical_1K))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas_1K[min_idx_empirical_1K])))

print("Min Theoretical P(error) for ERM = {:.4f}".format(min_prob_error_map_1K))
print("Min Theoretical Gamma = {:.3f}".format(gamma_theoretical))
```

Part 2(c):

```
# Use estimateMLE function to calculate the MLE parameters for D10K dataset using just the samples
Class0_100, Class1_100 = estimateMLE(D100, D100_labels, D_train[0])

# Generate the class-conditional likelihoods and compute ERM scores
ermScores_100 = estimateERMscoresV2(D20K, Class0_100, Class1_100)
# Estimate the ROC curve
roc_100, gammas_100 = estimateROC(ermScores_100,D20K_labels,Nl_valid)
# Plot the ROC curve with the empirical value for min-P(error)
fig_roc_100, ax_roc_100 = plt.subplots(figsize=(8, 8));
ax_roc_100.plot(roc_100['FPR'], roc_100['TPR'], label="Empirical ERM Classifier ROC Curve")
```

```
ax_roc_100.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_100.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
# ROC returns FPR vs TPR, but needs False Negative Rates, which is 1 - TPR
# Pr(error; γ) = p(D = 1|L = 0; γ)p(L = 0) + p(D = 0|L = 1; γ)p(L = 1)
Perror_empirical_100 = np.array((roc_100['FPR'], 1 - roc_100['TPR'])).T.dot(Nl_valid / D_validate)
# Min prob error for the empirical gamma values
min_Perror_empirical_100 = np.min(Perror_empirical_100)
min_idx_empirical_100 = np.argmin(Perror_empirical_100)
# Compute theoretical gamma
gamma_theoretical_100 = Class0_100['prior'] / Class1_100['prior']
MAP_decisions_100 = ermScores_100 > np.log(gamma_theoretical)
# Calculate the estimates for TPR, FNR, FPR, and TNR
MAP_metrics_100 = estimateClassMetrics(MAP_decisions_100, D20K_labels, Nl_valid)
# Compute probability of error using FPR and FNR and class priors
min_prob_error_map_100 = np.array((MAP_metrics_100['FPR'] * Class0_100['prior'] + MAP_metrics_100['FNR'] * Cla
# Plot theoretical and empirical
ax_roc_100.plot(roc_100['FPR'][min_idx_empirical_100], roc_100['TPR'][min_idx_empirical_100], 'co', label="Emp
            markersize=14)
ax_roc_100.plot(MAP_metrics_100['FPR'], MAP_metrics_100['TPR'], 'rx', label="Theoretical Min Pr(error) ERM", m
ax_roc_100.set_xlabel(r"Probability of False Alarm $p(D=1\,|\,L=0)$")
ax_roc_100.set_ylabel(r"Probability of True Positive $p(D=1\,|\,L=1)$")
ax_roc_100.set_title('ROC curve for EM estimated parameters from $D^{100}_{train}$ for ERM classifier')
plt.grid(True)
plt.legend()
plt.show()
print("Min Empirical P(error) for ERM = {:.4f}".format(min_Perror_empirical_100))
print("Min Empirical Gamma = {:.3f}".format(np.exp(gammas_100[min_idx_empirical_100])))

print("Min Theoretical P(error) for ERM = {:.4f}".format(min_prob_error_map_100))
print("Min Theoretical Gamma = {:.3f}".format(gamma_theoretical))
```

Part 3:

Functions:

```
# Define Epsilon; smallest positive floating value
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) inputs to the sigmoid referred to as logits
def logisticPrediction(X, w):
    z = X.dot(w)
    return sigmoid(z)
# Negative log Likelihood is equivalent to Binary Cross-Entropy Loss Function
def NLL(labels, predictions):
    # Limit values within arrays with addition of epsilon for avoidance of underflow
    predictions = np.clip(predictions, epsilon, 1 - epsilon)
    # log of class-conditional pdf for Class 0 - log p(L=0 | x; theta)
    log_p0 = (1 - labels) * np.log(1 - predictions + epsilon)
    # log of class-conditional pdf for Class 1 - log p(L=1 | x; theta)
    log_p1 = labels * np.log(predictions + epsilon)
    # Take negative result of mean-summation of log likelihoods
    return -np.mean(log_p0 + log_p1, axis=0)
def estimateLogisticMLE(X, labels):
    # Dimensionality including the addition of bias term
    theta0 = np.random.randn(X.shape[1])
```

```
    # Calculate the MLE parameter w* for the model using minimize() function
    cost = lambda w: NLL(labels, logisticPrediction(X, w))
    resultMLE = minimize(cost, theta0, tol=1e-6)
    # Return the solution array 'x' from minimize() function which is our MLE parameter - w*
    return resultMLE.x
def createPredictionScoreGrid(bounds_X, bounds_Y, params, prediction_function, phi=None, num_coords=200):
    # Note that I am creating a 200x200 rectangular grid
    xx, yy = np.meshgrid(np.linspace(bounds_X[0], bounds_X[1], num_coords),
                         np.linspace(bounds_Y[0], bounds_Y[1], num_coords))

    # Flattening grid and feed into a fitted transformation function if provided
    grid = np.c_[xx.ravel(), yy.ravel()]
    if phi:
        grid = phi.transform(grid)

    # Z matrix are the predictions given the provided model parameters
    Z = prediction_function(grid, params).reshape(xx.shape)

    return xx, yy, Z



# Function to compute classification results and plot correct and incorrect predictions
def estimateLogisticResults(ax, X, w, labels, N_labels, phi=None):
    #Report the probability of error and plot the classified data, plus predicted
    #decision contours of the logistic classifier applied to the phi-transformed data.
    predictions = logisticPrediction(phi.fit_transform(X), w)
    # Predicted decisions based on the default 0.5 threshold (higher probability mass on one side or the other
    decisions = np.array(predictions >= 0.5)
    logistic_metrics = estimateClassMetrics(decisions, labels, N_labels)
    # To compute probability of error, we need FPR and FNR
    prob_error = np.array((logistic_metrics['FPR'], logistic_metrics['FNR'])).T.dot(N_labels / labels.shape[0]
    # Plot correct and incorrect decisions (green = correct, red = incorrect)
    ax.plot(X[logistic_metrics['TN'], 0], X[logistic_metrics['TN'], 1], 'og', label="Correct Class 0");
    ax.plot(X[logistic_metrics['FP'], 0], X[logistic_metrics['FP'], 1], 'or', label="Incorrect Class 0");
    ax.plot(X[logistic_metrics['FN'], 0], X[logistic_metrics['FN'], 1], '+r', label="Incorrect Class 1");
    ax.plot(X[logistic_metrics['TP'], 0], X[logistic_metrics['TP'], 1], '+g', label="Correct Class 1");
    # Get grid coordinates and corresponding probability scores for the prediction function
    # Draw the decision boundary based on the phi transformation of input features x
    # Re-use the bounds from the validation set's x1 and x2 axes (global variables)
    xx, yy, Z = createPredictionScoreGrid(x1_valid_lim, x2_valid_lim, w, logisticPrediction, phi)
    # Once reshaped as a grid, plot contour of probabilities per input feature (ignoring bias)
    cs = ax.contour(xx, yy, Z, levels=1, colors='k')

    ax.set_xlabel(r"$x_1$")
    ax.set_ylabel(r"$x_2$")

    return prob_error
```

Part3(a):

```
# Define phi transformation (linear => degree 1)
phi = PolynomialFeatures(degree=1)
# Figure for D10K training decision boundary and D20K Classifier Decisions
fig_linear, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D1000 training dataset
w_mle = estimateLogisticMLE(phi.fit_transform(D10K), D10K_labels)
print("Logistic-Linear N={}; MLE for w: {}".format(10000, w_mle))
```

```
prob_error = estimateLogisticResults(ax1, D10K, w_mle, D10K_labels, N_labels_train[2], phi)
print("Training set error for the N={} classifier is {:.3f}".format(10000, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Linear Model N={}".format(10000))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_mle, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N={} classifier is {:.3f}\n\n".format(10000, prob_error))
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_linear.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()

# Define phi transformation (linear => degree 1)
phi = PolynomialFeatures(degree=1)
# Figure for D1000 training decision boundary and D20K Classifier Decisions
fig_linear, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D1000 training dataset
w_mle = estimateLogisticMLE(phi.fit_transform(D1K), D1K_labels)
print("Logistic-Linear N={}; MLE for w: {}".format(1000, w_mle))
prob_error = estimateLogisticResults(ax1, D1K, w_mle, D1K_labels, N_labels_train[1], phi)
print("Training set error for the N={} classifier is {:.3f}".format(1000, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Linear Model N={}".format(1000))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_mle, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N={} classifier is {:.3f}\n\n".format(1000, prob_error))
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_linear.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()

# Define phi transformation (linear => degree 1)
phi = PolynomialFeatures(degree=1)
# Figure for D100 training decision boundary and D20K Classifier Decisions
fig_linear, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D100 training dataset
w_MLE = estimateLogisticMLE(phi.fit_transform(D100), D100_labels)
print("Logistic-Linear N={}; MLE for w: {}".format(100, w_MLE))
prob_error = estimateLogisticResults(ax1, D100, w_MLE, D100_labels, N_labels_train[0], phi)
print("Training set error for the N={} classifier is {:.3f}".format(100, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Linear Model N={}".format(100))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_MLE, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N={} classifier is {:.3f}\n\n".format(100, prob_error))
```

```
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_linear.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()
```

Part3(b):

```
# Define phi transformation (quadratic => degree 2)
phi = PolynomialFeatures(degree=2)
# Figure for D100 training decision boundary and D20K Classifier Decisions
fig_quad, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D100 training dataset
w_mle = estimateLogisticMLE(phi.fit_transform(D10K), D10K_labels)
print("Logistic-Quadratic N={}; MLE for w: {}".format(10000, w_mle))
prob_error = estimateLogisticResults(ax1, D10K, w_mle, D10K_labels, N_labels_train[2], phi)
print("Training set error for the N={} classifier is {:.3f}".format(10000, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Quadratic Model N={}".format(10000))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_mle, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N={} classifier is {:.3f}\n\n".format(10000, prob_error))
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_quad.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()

# Define phi transformation (quadratic => degree 2)
phi = PolynomialFeatures(degree=2)
# Figure for D1K training decision boundary and D20K Classifier Decisions
fig_quad, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D1K training dataset
w_mle = estimateLogisticMLE(phi.fit_transform(D1K), D1K_labels)
print("Logistic-Quadratic N={}; MLE for w: {}".format(1000, w_mle))
prob_error = estimateLogisticResults(ax1, D1K, w_mle, D1K_labels, N_labels_train[1], phi)
print("Training set error for the N={} classifier is {:.3f}".format(1000, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Quadratic Model N={}".format(1000))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_mle, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N={} classifier is {:.3f}\n\n".format(1000, prob_error))
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
```

```
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_quad.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()

# Define phi transformation (quadratic => degree 2)
phi = PolynomialFeatures(degree=2)
# Figure for D100 training decision boundary and D20K Classifier Decisions
fig_quad, (ax1, ax2) = plt.subplots(1,2, figsize = (12,6));
# D100 training dataset
w_mle = estimateLogisticMLE(phi.fit_transform(D100), D100_labels)
print("Logistic-Quadratic N = {}; MLE for w: {}".format(100, w_mle))
prob_error = estimateLogisticResults(ax1, D100, w_mle, D100_labels, N_labels_train[0], phi)
print("Training set error for the N = {} classifier is {:.3f}".format(100, prob_error))
ax1.set_title("Decision Boundary for \n Logistic-Quadratic Model N={}".format(100))
ax1.set_xticks([])
prob_error = estimateLogisticResults(ax2, D20K, w_mle, D20K_labels, Nl_valid, phi)
ax2.set_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(D_validate))
print("Validation set error for the N = {} classifier is {:.3f}\n\n".format(100, prob_error))
ax2.set_xticks([])
# Again use the most sampled subset (validation) to define x-y limits
plt.setp((ax1, ax2), xlim=x1_valid_lim, ylim=x2_valid_lim)
# Adjust subplot positions
plt.subplots_adjust(hspace=0.3)
# Super plot the legends
handles, labels = ax2.get_legend_handles_labels()
fig_quad.legend(handles, labels, loc='lower center')
plt.tight_layout()
plt.show()
```

Question 1 Citations:

Gaussian Mixture Models Explained - https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

EM Algorithm and Gaussian Mixture Models - https://xavierbourretsicotte.github.io/gaussian_mixture.html

sklearn.mixture.GaussianMixture - https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html

sklearn.preprocessing.PolynomialFeatures - https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

Logistic and Quadratic Regression Classifier in Python - https://towardsdatascience.com/logistic-regression-from-scratch-in-python-ec66603592e2

Question 2 Code:

```
function [Kx,Ky] = Q2plotKContours(K,xT,yT,P)
% Function to compute base x, base y of K landmarks and compute distance
% between 2D base station coordinates and potential true position of
% vehicle using Prior knowledge and parameters of x_T and y_T
```

```
% Plot a Circle of Unit Radius that K landmarks reside on
viscircles([0 0],1,'Color','k','LineStyle','--','LineWidth',0.5); hold on
w = 2*pi/K;
Kx = zeros(1,length(K));
Ky = zeros(1,length(K));
for i = 0:K-1
Kx(1,i+1) = round(cos(w*i),6); Ky(1,i+1) = round(sin(w*i),6);
plot(Kx(1,i+1),Ky(1,i+1),'ro','LineWidth',2)
end
% Contour Plot for PDF of Vehicle Position
contour(xT,yT,P,10,'LineWidth',1.5)
xlabel('x'),ylabel('y'), grid on
title("\it{K} = "+K+" Landmarks on Circle of Unit Radius")
end

function LL = LogLikelihood(K,x)
%Function that computes the Negative-Log-Likelihood from the given Range
%Values, Prior parameters, and x/y-vectors that serve as inputs
w = 2*pi/K;
kx = zeros(1,length(K));
ky = zeros(1,length(K));
sig_x = 0.25;
sig_y = 0.25;
for i = 0:K-1
kx(1,i+1) = round(cos(w*i),6); ky(1,i+1) = round(sin(w*i),6);
end
base = [kx' ky'];
for i = 1:K
    r(i,:) = calculateRange(base,x,0.3);
end
ri = sum(r(:));
x_P = 0.2;
y_P = -0.1;
LL = -(ri+(1/2)*(x_P^2/sig_x^2 + y_P^2/sig_y^2));
end
```