Contents

Problem 1

A 2-dimensional random vector **X** has the probability density function:

$$p(x) = p(\boldsymbol{x}|L=0)P(L=0) + p(\boldsymbol{x}|L=1)p(L=1)$$

With class priors: $P(L=0) = 0.65 \quad and \quad P(L=1) = 0.35$

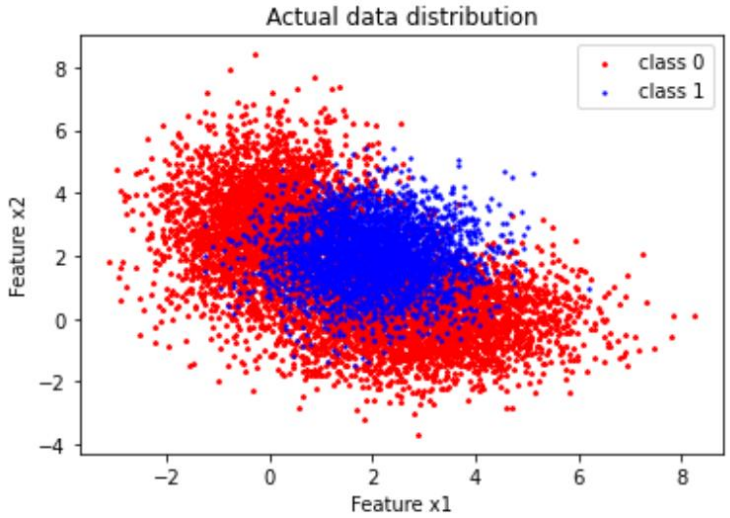
The class-conditional pdfs are:

$$p(\boldsymbol{x}|L=0) = w_1 g(\boldsymbol{x}|\mu_{01}, \Sigma_{01}) + w_2 g(\boldsymbol{x}|\mu_{02}, \Sigma_{02}) \text{ and } p(\boldsymbol{x}|L=1) = g(\boldsymbol{x}|\mu_1, \Sigma_1)$$

With equal weights ($w_1 = w_2 = 1/2$) and the following mean vectors and covariance matrices:

$$\mu_{01} = \begin{bmatrix}3\\0\end{bmatrix}, \Sigma_{01} = \begin{bmatrix}2 & 0\\0 & 1\end{bmatrix} \quad \mu_{02} = \begin{bmatrix}0\\3\end{bmatrix}, \Sigma_{02} = \begin{bmatrix}1 & 0\\0 & 2\end{bmatrix} \quad \mu_1 = \begin{bmatrix}2\\2\end{bmatrix}, \Sigma_1 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

Training datasets consisting of 20, 200, and 2000 samples were generated in addition to a validation set of 10,000 samples shown below:



Part 1 – Theoretically Optimal Classification

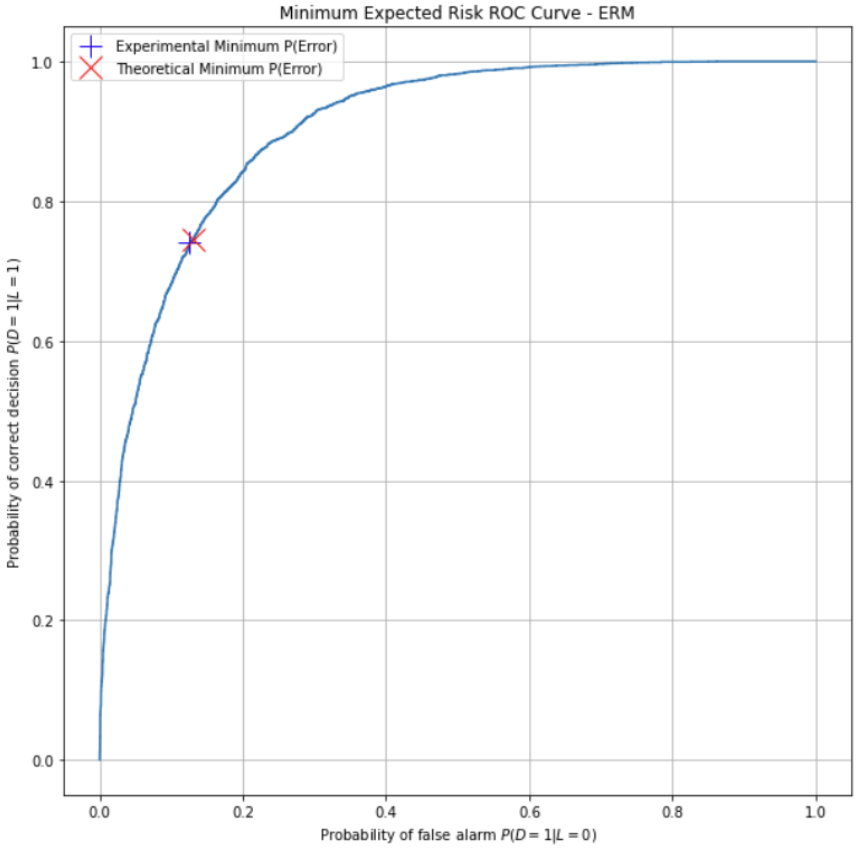
To determine the theoretically optimal classifier, MAP was used as the classification rule:

$$\frac{p(x|L=1)}{p(x|L=0)} = \frac{g(x|m_1, C_1)}{p(x|m_0, C_0)} > \gamma = \frac{p(x|L=0)}{p(x|L=1)} * \frac{\lambda_{01} - \lambda_{00}}{\lambda_{10} - \lambda_{11}}$$

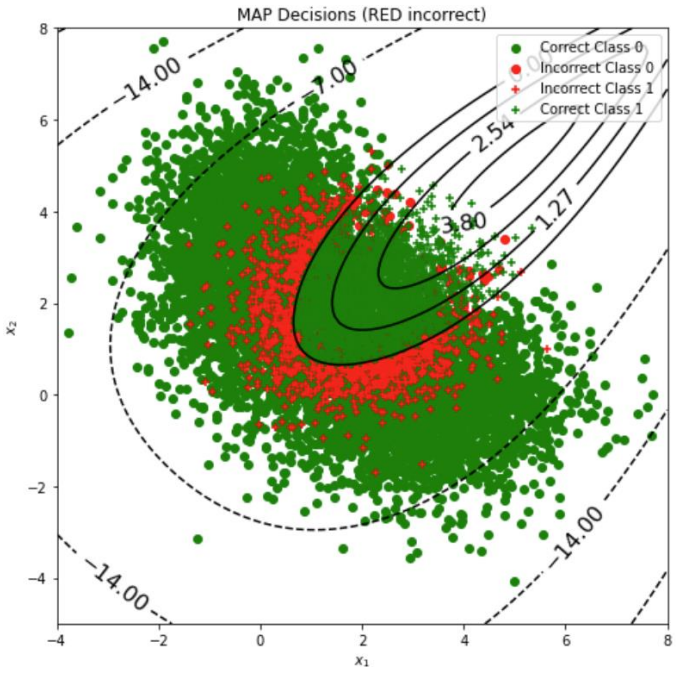
Where the γ threshold is a function of class priors and loss values.

$$\gamma = \frac{0.65}{0.35} * \frac{1-0}{1-0} = 1.857$$

This classifier was applied to all samples in the validation dataset to generate the following ROC Curve from the decision rules and true labels.



The decision boundary using this theoretically ideal classification rule has been overlaid on the validation dataset with green points representing correct classification and red representing incorrect classification. Because we have knowledge of the true dataset characteristics, this establishes an aspirational level of performance for comparison of future approximations.



For this classifier, the theoretical optimum and approximated γ and $P(error)$ were calculated:

	γ	$\min P(error;\gamma)$
Theoretical	1.8571	0.1743
Full Knowledge Estimate	1.9176	0.1728

Part 2 – MLE logistic-linear-function-based approximations

The Maximum Likelihood (ML) parameter estimation technique was used to train three approximations of class label posterior functions given a sample using each of the three generated training datasets.

With ***x*** representing the input sample vector and θ denoting the model parameter vector, logistic-linear-function refers to $h(x,\theta) = 1/1 + e^{-\theta^T z(x)}$, where $z(x) = [1, x^T]^T$ is the augmented input vector (equivalent to $\tilde{x}$ in class notes)

To solve this optimization problem, the minimization of the negative-log-likelihood (NLL) was used on the training datasets followed by gradient descent. These class-label-posterior approximations were used to classify a sample in order to approximate the minimum-P(error) classification rule. These three approximations of class label posterior function on samples in the validation dataset and a probability of error was estimated for each. The results are shown in the table below:

	$\min P(error;\gamma)$ On Training set	$\min P(error;\gamma)$ On Validation set
Theoretical		0.1743
Full Knowledge Estimate		0.1728
Trained on 20 samples (linear)	0.25	0.6208
Trained on 200 samples (linear)	0.4125	0.4346
Trained on 2000 samples (linear)	0.3534	0.3435

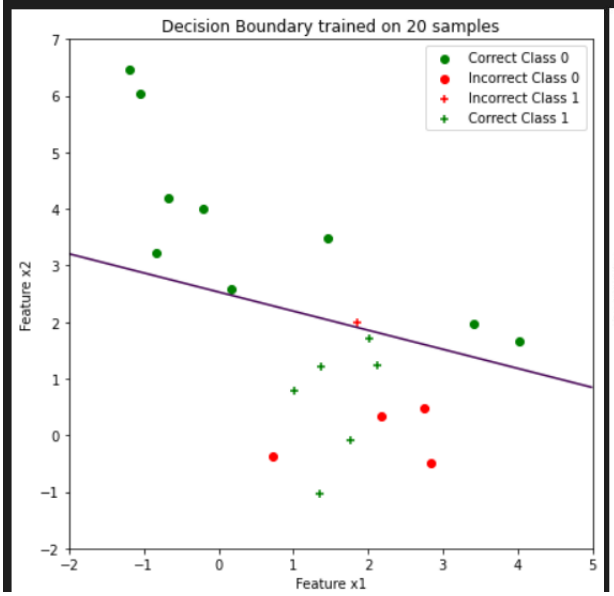
The minimum probability of error decreases as the number of training samples used to generate the decision boundary is increased. This makes sense because as more samples are used, it better represents the actual dataset.

We can also see how the error prediction on the training set does not always correlate to a similarly accurate prediction on the validation set. As the number of training samples increases, the minimum error difference between the training and validation sets increase. The 20-sample estimate on the training set is low (0.25), but when it is applied to the validation set, it gets more than half of the decisions wrong (0.6208). This shows that more samples are needed to represent the dataset.

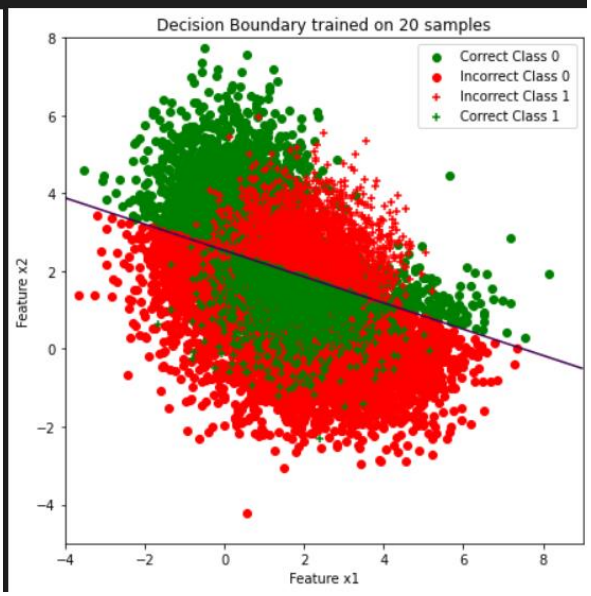
The best approximate using a linear classifier results in a minimum probability of error of 0.3435, which is almost exactly doubled to the theoretically optimal classifier with a minimum probability of error of 0.1728. This concludes that a linear classifier is not a good choice for this dataset.

The resulting plots of correct (green) and incorrect (red) samples are plotted for each of the three classifiers for both the training set and the validation set. These plots are overlayed with the decision boundaries.

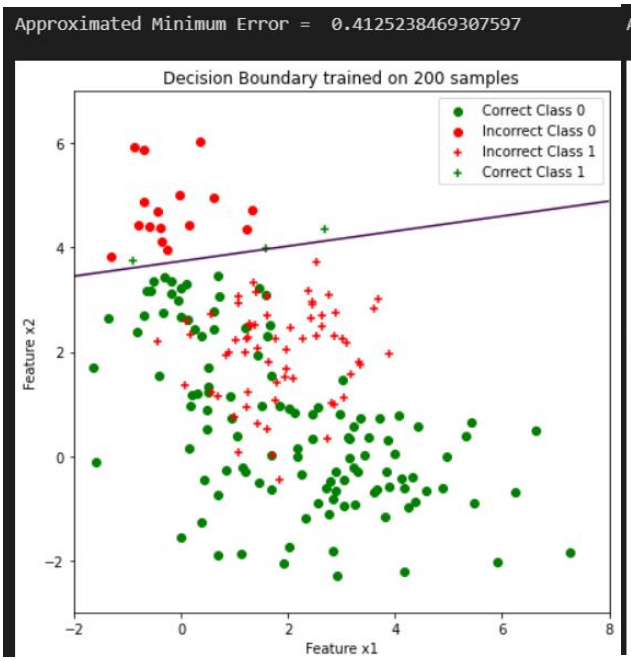
```
Approximated Minimum Error =  0.25
```



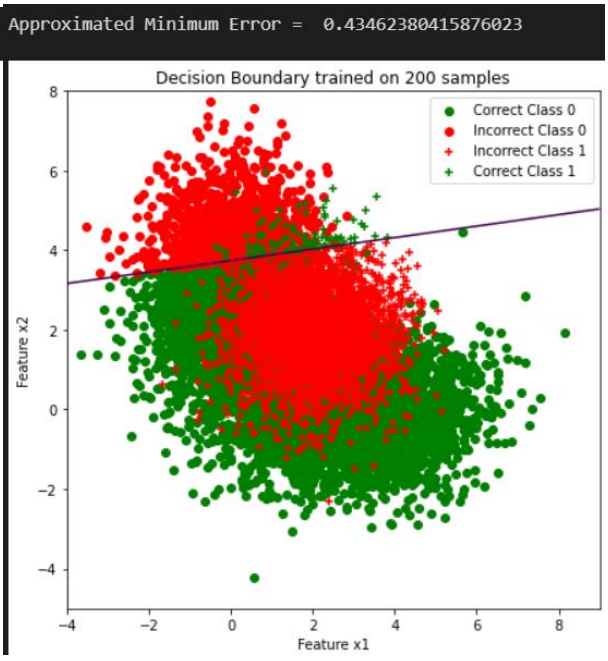

```
Approximated Minimum Error =  0.6208227091239722
```

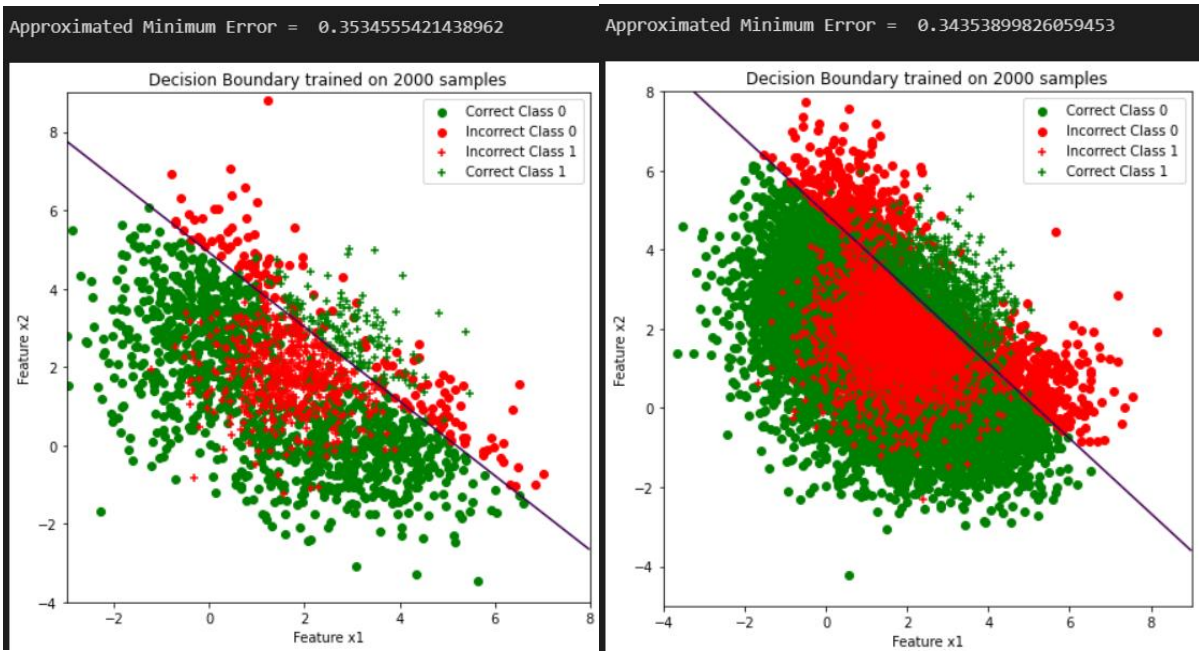



```
Approximated Minimum Error =  0.4125238469307597
```




```
Approximated Minimum Error =  0.43462380415876023
```





Part 2 – MLE logistic-quadratic-function-based approximation

The steps from the previous section are repeated here for a quadratic function.

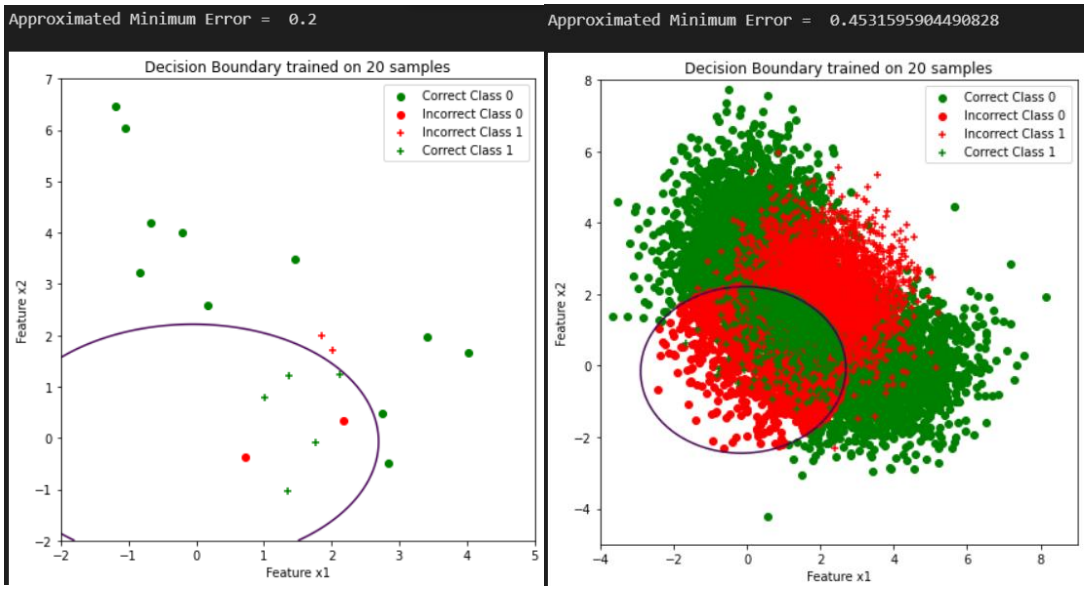
The logistic-quadratic-function refers to $h(x,\theta) = 1/1 + e^{-\theta^T z(x)}$, where $z(x) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T$

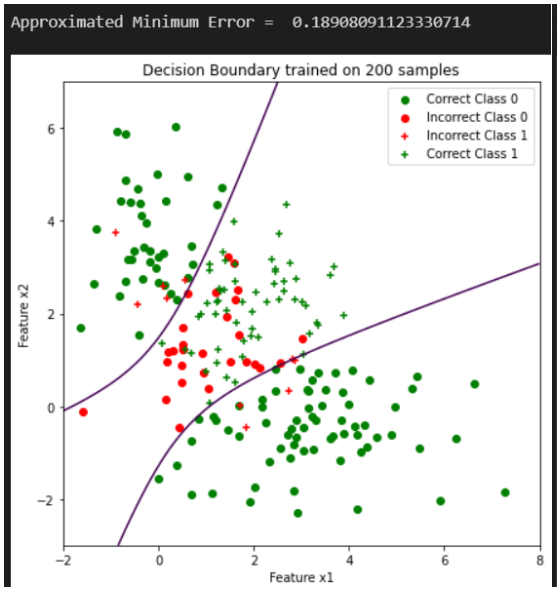
Once again, the three approximations of class label posterior function based this time on the quadratic classifier results in an estimated probability of error on samples in the validation dataset and the training dataset. The results are shown in the table below:

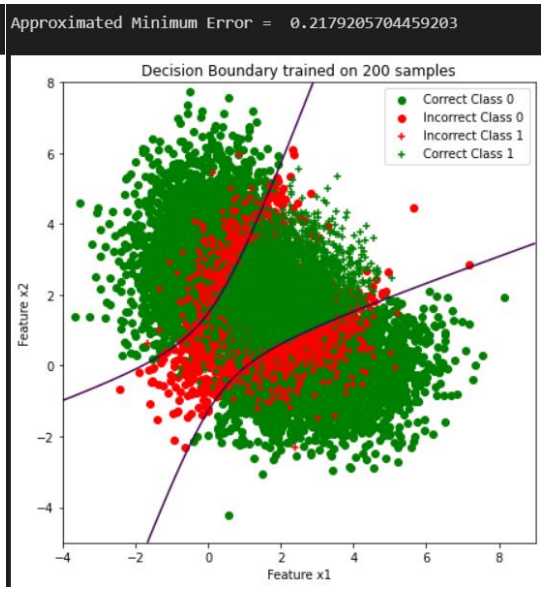
	$\min P(error;\gamma)$ On Training set	$\min P(error;\gamma)$ On Validation set
Theoretical		0.1743
Full Knowledge Estimate		0.1728
Trained on 20 samples (quadratic)	0.2	0.4531
Trained on 200 samples (quadratic)	0.1891	0.2179
Trained on 2000 samples (quadratic)	0.1629	0.1862

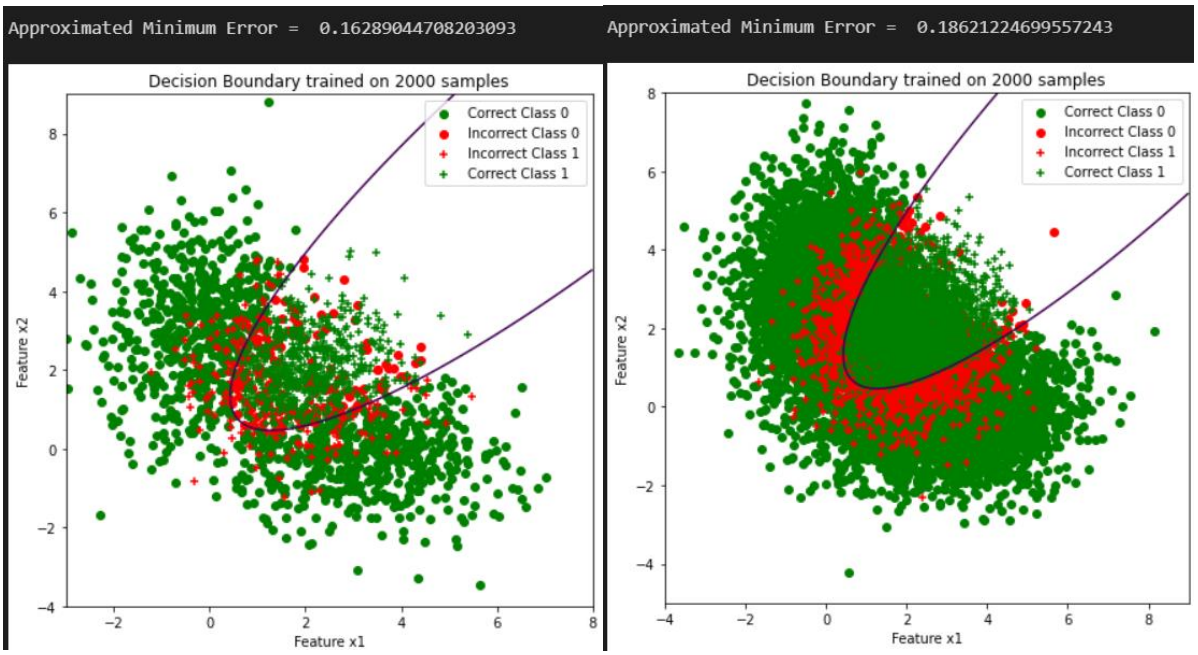
Once again, the minimum probability of error decreases as the number of training samples used to generate the decision boundary is increased. The same logic used for the linear classifier applies to this quadratic classifier. The main difference is that the quadratic classifier is a better fit for the data. In this case, the best approximation using the quadratic classifier is 0.1862, which is very close to the theoretically optimal minimum probability of error of 0.1728. This concludes that a quadratic classifier is a good choice for this dataset.

The resulting plots of correct (green) and incorrect (red) samples are plotted for each of the three classifiers for both the training set and the validation set. These plots are overlayed with the decision boundaries









Conclusions

The minimum probability of errors generated by the linear classifier, quadratic classifier, and theoretically optimal classifier can be seen in the table below.

	$\min P(error;\gamma)$ On Validation set
Theoretical	0.1743
Full Knowledge Estimate	0.1728
Trained on 20 samples (linear)	0.6208
Trained on 200 samples (linear)	0.4346
Trained on 2000 samples (linear)	0.3435
Trained on 20 samples (quadratic)	0.4531
Trained on 200 samples (quadratic)	0.2179
Trained on 2000 samples (quadratic)	0.1862

We can conclude that the quadratic classifier is far superior to the linear classifier, shown in the high error probability of 0.3435 for the linear case trained on 2000 samples compared to the lower 0.1862 for the quadratic case trained on 2000 samples. The quadratic classifier is closely related to the theoretically optimal classifier.

Problem 2

Instead of predicting the labels to estimate the parameters, the parameters can be estimated directly based on the optimization of a loss function. In this case those loss functions will be Maximum Likelihood (ML), which is a deterministic point estimate, and Maximum a Posteriori (MAP), which is a random estimate with a prior. Both of these loss functions assume i.i.d samples.

Maximum Likelihood (ML) and Maximum a Posteriori (MAP) parameter estimates were derived according to $y = c(\boldsymbol{x}, \boldsymbol{\theta}) + v$, where x is a two-dimensional vector, the function c represents a cubic polynomial, v is a random Gaussian scalar, and θ is the parameter vector consisting of coefficients.

The bivariate cubic polynomial equation is shown below.

$$y = a_0x_0^3 + a_1x_1^3 + b_0x_0^2 + b_1x_1^2 + c_0x_0 + c_1x_1 + dx_1x_2 + ex_1^2x_2 + fx_1x_2^2 + g + v$$

Therefore, the parameter vector and augmented X-vector are:

$$\theta = [g, c_0, c_1, b_0, b_1, a_0, a_1, d, e, f]$$

$$\tilde{X} = \phi = [1, x_0, x_1, x_0^2, x_1^2, x_0^3, x_1^3, x_1x_2, x_1^2x_2, x_1x_2^2]$$

ML Estimator Derivation

ML estimators converge to a true parameter estimate as the number of training samples increases and is simpler to implement than other estimators (such as Bayesian/MAP). However, it may produce biased parameter estimates of true parameters. ML estimators requires a lot of data, otherwise it is susceptible to poor generalization and overfitting.

The ML estimator loss function is the same as Negative Log Likelihood (NLL). Setting $\sigma^2 = 1$, then NLL is the same as residual sum of squares (RSS):

$$NLL(\theta) = \frac{1}{2\sigma^2}\left(\sum_{i=1}^{N}\left(y^{(i)} - \theta^T x^{(i)}\right)^2\right) = \frac{1}{2}\left|\left|X\theta - y\right|\right|_2^2 = \frac{1}{2}\,(X\theta - y)^T(X\theta - y)$$

To solve for the parameter estimation, we rearrange this loss function and turn it into an optimization problem:

$$\hat{\theta}_{MLE} = \underset{\theta}{argmax} \sum_{i=1}^{N} \log p\left(x^{(i)}\middle|\theta\right)$$

$$\hat{\theta}_{MLE} = \hat{\theta}_{NLL} = \underset{\theta}{argmin}\, NLL(\theta) = \underset{\theta}{argmin} \left(\frac{1}{2}\,(X\theta - y)^T(X\theta - y)\right)$$

Where:

$$y^T = \left[y^{(1)}, \dots, y^{(N)}\right] \in \mathbb{R}^{1\times N}, \quad \theta^T = [\theta_0, \theta_1, \dots, \theta_n] \in \mathbb{R}^{1\times n}, \quad X = \begin{bmatrix} \tilde{x}^{((1))T} \\ \vdots \\ \tilde{x}^{((N))T} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_n^{(1)} \\ \vdots & \ddots & \ddots \\ 1 & x_1^{(N)} & x_n^{(N)} \end{bmatrix} \in \mathbb{R}^{N\times n}$$

The solution to MLE/NLL is when $\frac{\delta NLL(\theta)}{\delta\theta} = 0$:

$$\frac{\delta}{\delta\theta}\left[\frac{1}{2}(X\theta - y)^T(X\theta - y)\right] = \frac{1}{2}\frac{\delta}{\delta\theta}[\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y] = X^T X\theta - X^T y = 0$$

$$X^T X\hat{\theta}_{NLL} = X^T y \rightarrow \hat{\theta}_{MLE} = (X^T X)^{-1}X^T y$$

MAP Estimator Derivation

Regularization is used to avoid the overfitting problems present in MLE, which results in the Bayesian/MAP parameter estimate. A prior is added to the ML estimate, giving the MAP estimator a measure of uncertainty, as it is a random variable.

MAP is better for a smaller number of samples and captures a complete parameter representation (most probable estimate and uncertainty) from a single dataset. ML estimates only produce the "best" estimate and needs repeated experiments. To make MAP into an optimization problem, the most probable parameter estimate is taken (mode).

Deriving the MAP parameter estimation expression is similar to MLE but using a parameter distribution rather than likelihood. In this case, the problem with overfitting that is present in ML estimates are tackled by adding a prior term.

$$\hat{\theta}_{MAP} = \underset{\theta}{argmax} \sum_{i=1}^{N}\left[\log p\left(x^{(i)}|\theta\right) + \log p(\theta)\right]$$

Where the $\log\left(p(\theta)\right)$ term is called the "complexity penalty". Once again, we minimize the NLL:

$$NLL(\theta) = \frac{1}{2\sigma^2}\left(\sum_{i=1}^{N}\left(y^{(i)} - \theta^T x^{(i)}\right)^2 + \frac{N}{2}\log(2\pi\sigma^2)\right)$$

If we assume a fixed variance we get the same negative log likelihood as in MLE,

$$NLL(\theta) = \frac{1}{2}\left||X\theta - y|\right|_2^2 = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$

But now we also have a regularization term based on the gaussian prior with 0-mean and γI- covariance. The multivariate gaussian pdf for the prior is equal to:

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

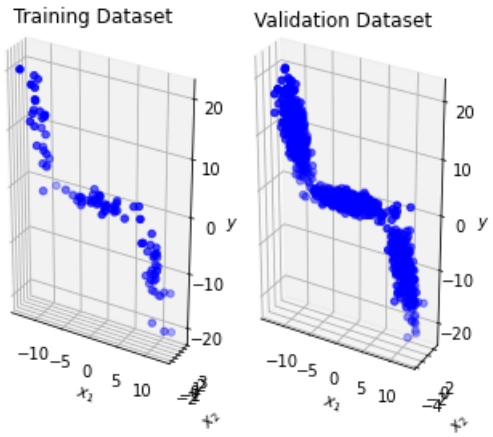
To simplify, we can drop the constants that do not contribute to finding the maximum, set x = 1 and μ=0, and take the logarithm of the expression to get an estimated regularization matrix that is the diagonal of arithmetic average of non-zero eigenvalues in the covariance matrix. ($\Sigma^{-1} \rightarrow \frac{1}{\gamma}I$)

After setting $\frac{\delta NLL(\theta)}{\delta\theta} = 0$, we get a very similar result to MLE, but with the complexity penalty term:

$$\hat{\theta}_{MAP} = \left(X^T X + \frac{1}{\gamma}I\right)^{-1}X^T y$$

Estimator Implementation

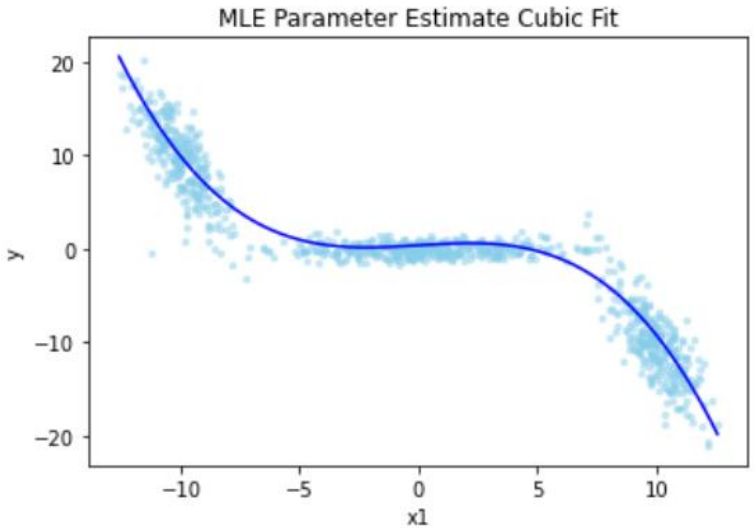
Next, these derived estimator expressions were implemented in code and applied to the generated dataset provided (below).



The training dataset was used to obtain the ML estimator and MAP estimator at a variety of γ values, which were evaluated using the mean-squared error (MSE) of the validation samples.

$$MLE\ MSE = 5.1192316$$

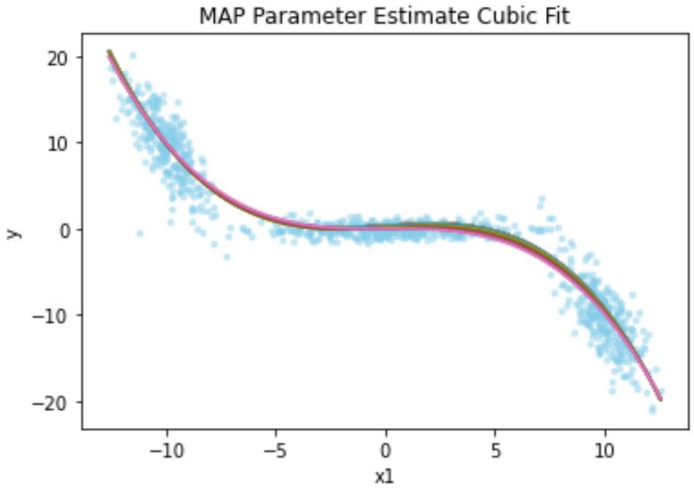
Plugging the estimated parameters into the cubic equation for x1 and x2 as the coefficients results in a cubic line of best fit on the provided data below. Note that, as can be visualized on the data plotted above, that most of the information is in the x1 dimension rather than the x2 dimension. Therefore, the visualization was done using just the x1 dimension.



The Maximum a Posteriori estimator yielded a series of MSEs relative to the changing γ:

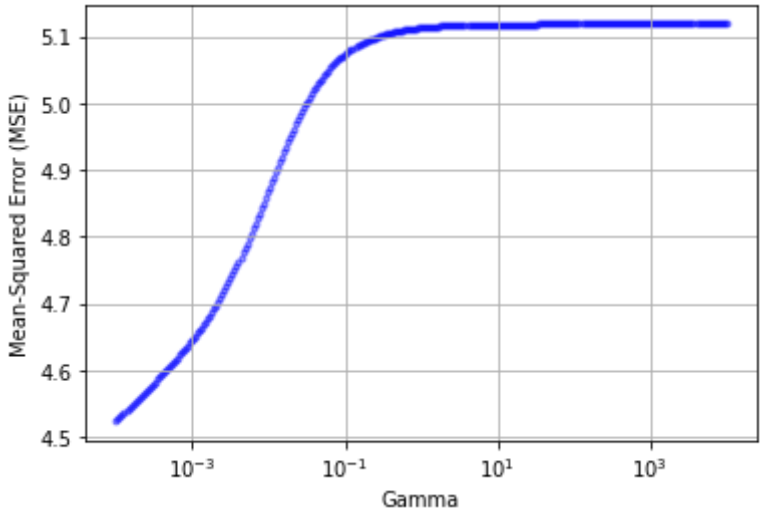
Gamma	MSE
0.0001	4.52403605
0.001	4.64564407
0.01	4.87233812
0.1	5.07365873
1	5.1137509
10	5.11866414
100	5.11917468
1000	5.11922595
10000	5.11923108

Similarly to in the MLE implementation, the MAP parameter estimation was plotted against the x1-dimension of the provided dataset. Although it is difficult to visualize, there are 17 different lines estimated on the plot below.



Conclusions

Using several more iterations of the gamma value, the MSE was plotted against the log of gamma. It can be seen that as gamma approaches infinity, the mean squared error of the MAP estimate approaches the same value as MLE.



The main difference between MLE and MAP is that MAP considers a distribution of likelihoods rather than an exact one. This implies that MAP is equivalent to MLE with a covariance of zero. This can be seen in the plot above, because as gamma approaches infinity, the covariance approaches zero.

This plot also shows that the maximum squared error increases with increasing gamma, indicating that MAP can estimate better than MLE given lower values of gamma (higher covariance eigenvalues).

Appendix

The Code for each problem can be found in the Appendices below.

It can also be found in my github at https://github.com/meuliano/Machine-Learning/tree/main/HW2 with the following python scripts:

- Problem 1: Euliano_eece5644_hw2_1.py
- Problem 2: Euliano_eece5644_hw2_2.py

Appendix A – Problem 1 Python Code

```
# %%
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sys import float_info # Threshold smallest positive floating value


priors = np.array([0.65, 0.35])
weights = np.array([.5, .5])
class0_mean = np.array([[3,0],[0,3]])
class0_cov = np.array([[[2,0],[0,1]],[[1,0],[0,2]]])
class1_mean = np.array([2,2])
class1_cov = np.array([[1,0],[0,1]])

# %%
def generateData(numsamples):
    labels = np.where(np.random.rand(numsamples) >= .65, 1, 0)
    data = np.empty((numsamples,2))

    for i in range(numsamples):
        if(labels[i] == 1):
            data[i] = np.random.multivariate_normal(class1_mean, class1_cov)
        else:
            rand = np.random.rand()
            subclass = 1 if rand>weights[0] else 0
            data[i] = np.random.multivariate_normal(class0_mean[subclass],
class0_cov[subclass])

    return data, labels

# %%
# Generate ROC curve samples
def estimate_roc(discriminant_score, label):
    Nlabels = np.array((sum(label == 0), sum(label == 1))) #~[6500,3500]

    sorted_score = sorted(discriminant_score)

    # Use tau values that will account for every possible classification
split
    # These are all the possible values of Gamma that we will test through
    taus = ([sorted_score[0] - float_info.epsilon] + sorted_score +
             [sorted_score[-1] + float_info.epsilon])

    # Calculate the decision label for each observation for each gamma
[10000]
```

```
    decisions = [discriminant_score >= t for t in taus]

    # True Positive
    ind11 = [np.argwhere((d==1) & (label==1)) for d in decisions]
    p11 = [len(inds)/Nlabels[1] for inds in ind11]
    # False Positive
    ind10 = [np.argwhere((d==1) & (label==0)) for d in decisions]
    p10 = [len(inds)/Nlabels[0] for inds in ind10]

    # ROC has FPR on the x-axis and TPR on the y-axis
    roc = np.array((p10, p11))

    # Calculate error probability
    prob_error = [(p10[w] *priors[0] + (1 - p11[w])* priors[1]) for w in
range(len(p10))]

    return roc, taus, prob_error

# %%
# Gradient Descent Function
def gradient_descent(loss_func, grad_func, theta0, x_tilde, labels,
**kwargs):

    max_epoch = kwargs['max_epoch'] if 'max_epoch' in kwargs else 200
    alpha = kwargs['alpha'] if 'alpha' in kwargs else 0.1
    epsilon = kwargs['tolerance'] if 'tolerance' in kwargs else 1e-6

    w = theta0
    w_history = w
    f_history = loss_func(w, x_tilde, labels)
    delta_w = np.zeros(w.shape)
    i = 0
    diff = 1.0e10

    while  i<max_epoch and diff>epsilon:
        delta_w = -alpha*grad_func(w, x_tilde, labels)
        w = w+delta_w

        # store the history of w and f
        w_history = np.vstack((w_history,w))
        f_history = np.vstack((f_history,loss_func(w, x_tilde, labels)))

        # update iteration number and diff between successive values
        # of objective function
        i+=1
        diff = np.absolute(f_history[-1]-f_history[-2])

    return w_history,f_history

# %%
# Utility Functions
def error_function_gradient(theta, x_tilde, labels):
    a = theta.dot(x_tilde)
    return np.sum(np.multiply(sigmoid(a)-labels, x_tilde), axis = 1)

def error_function(theta, x_tilde, labels):
    a = theta.dot(x_tilde)
```

```
    return -np.sum((np.multiply(labels, np.log(sigmoid(a)))+np.multiply((1-
labels), np.log(1-sigmoid(a)))), axis=0)

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

def classify_logistic_linear(x, w):
    x_tilde = np.vstack((np.ones(x.shape[0]), x[:,0], x[:,1]))
    return np.where(sigmoid(w.dot(x_tilde))> .5, 1, 0)

def classify_logistic_quadratic(x, w):
    x_tilde = np.vstack((np.ones(x.shape[0]), x[:,0], x[:,1], x[:,0]**2,
x[:,0]*x[:,1], x[:,1]**2))
    return np.where(sigmoid(w.dot(x_tilde))> .5, 1, 0)

# %%
def plot_boundary(x,labels,decisions,w,priors,samples,quadratic=False):
    Nlabels = np.array((sum(labels == 0), sum(labels == 1)))
    # True Negative Probability
    ind_00 = np.argwhere((decisions==0) & (labels==0))
    p_00 = len(ind_00) / Nlabels[0]
    # False Positive Probability
    ind_10 = np.argwhere((decisions==1) & (labels==0))
    p_10 = len(ind_10) / Nlabels[0]
    # False Negative Probability
    ind_01 = np.argwhere((decisions==0) & (labels==1))
    p_01 = len(ind_01) / Nlabels[1]
    # True Positive Probability
    ind_11 = np.argwhere((decisions==1) & (labels==1))
    p_11 = len(ind_11) / Nlabels[1]

    # Probability of error for MAP classifier, empirically estimated
    prob_error_approx = (p_10 *priors[0] + (1 - p_11)* priors[1])
    print("Approximated Minimum Error = ", prob_error_approx)


    fig_disc_grid, ax_disc = plt.subplots(figsize=(7, 7));
    ax_disc.scatter(x[ind_00, 0], x[ind_00, 1], c='g',marker='o',
label="Correct Class 0")
    ax_disc.scatter(x[ind_10, 0], x[ind_10, 1],c='r',marker='o',
label="Incorrect Class 0")
    ax_disc.scatter(x[ind_01, 0], x[ind_01, 1], c='r',marker='+',
label="Incorrect Class 1")
    ax_disc.scatter(x[ind_11, 0], x[ind_11, 1], c='g',marker='+',
label="Correct Class 1")


    horizontalGrid =
np.linspace(np.floor(min(x[:,0])),np.ceil(max(x[:,0])),100);
    verticalGrid =
np.linspace(np.floor(min(x[:,1])),np.ceil(max(x[:,1])),100);

    dsg = np.zeros((100,100))
    a = np.array(np.meshgrid(horizontalGrid,verticalGrid))
    for i in range(100):
        for j in range(100):
            x1 = a[0][i][j]
```

```
            x2 = a[1][i][j]
            if quadratic==False:
                z = np.c_[1,x1,x2].T
            else:
                z = np.c_[1,x1,x2,x1**2,x1*x2,x2**2].T
            dsg[i][j] = np.sum(np.dot(w.T,z))
    ax_disc.contour(a[0],a[1],dsg, levels = [0])
    plt.legend()
    plt.title("Decision Boundary trained on {} samples".format(samples))
    plt.xlabel("Feature x1")
    plt.ylabel("Feature x2")
    plt.show()

# %%
def main():

    # Generate Training and Validation Datasets
    train_data_20, train_labels_20 = generateData(20)
    train_data_200, train_labels_200 = generateData(200)
    train_data_2000, train_labels_2000 = generateData(2000)
    validate_data_10000, validate_labels_10000 = generateData(10000)

    train_data =
np.array([train_data_20,train_data_200,train_data_2000,],dtype=object)
    train_labels =
np.array([train_labels_20,train_labels_200,train_labels_2000],dtype=object)

    # Plot Raw Data
    X = validate_data_10000
    labels = validate_labels_10000

    plt.scatter(X[labels==0,0],X[labels==0,1],s=5, color = 'red', label =
'class 0',marker='*')
    plt.scatter(X[labels==1,0],X[labels==1,1],s=2, color = 'blue', label =
'class 1',marker='o')
    plt.title("Actual data distribution")
    plt.xlabel("Feature x1")
    plt.ylabel("Feature x2")
    plt.legend()

    # %%
    # Theoretically Optimal Classifier
    # Use the 10K validation dataset to find minimum P(error)
    x = validate_data_10000
    labels = validate_labels_10000
    Nlabels = np.array((sum(labels == 0), sum(labels == 1)))

    # Class conditional likelihood is a 2x10,000 matrix of likelihoods for
each sample based on gaussian distribution
    ccl0 = weights[0]*multivariate_normal.pdf(x, class0_mean[0],
class0_cov[0]) + weights[1]*multivariate_normal.pdf(x, class0_mean[1],
class0_cov[1])
    ccl1 = multivariate_normal.pdf(x, class1_mean, class1_cov)
    class_conditional_likelihoods = np.array([ccl0, ccl1])

    # Discriminant Score using log-likelihood ratio: is a 10,000-length
vector of values to compare to the threshold
```

```
    discriminant_score = np.log(class_conditional_likelihoods[1]) -
np.log(class_conditional_likelihoods[0])

    # Vary Gamma gradually and compute True-Positive and False-Positive
probabilities
    roc_erm, tau, prob_error = estimate_roc(discriminant_score, labels)

    # Find minimum error and index
    minimum_error = min(prob_error)
    minimum_index = prob_error.index(minimum_error)

    # Experimental / approximate Threshold Gamma value
    # e^ (undo log in prev cell)
    gamma_approx = np.exp(tau[minimum_index])
    print("Approximated Gamma = ", gamma_approx)
    print("Approximated Minimum Error = ", minimum_error)

    gamma_th = priors[0]/priors[1]
    # Same as: gamma_th = priors[0]/priors[1]
    print("Theoretical Gamma = ", gamma_th)

    # get decision for EACH sample based on theoretically optimal threshold
    decisions_map = discriminant_score >= np.log(gamma_th)

    # True Negative Probability
    ind_00_map = np.argwhere((decisions_map==0) & (labels==0))
    p_00_map = len(ind_00_map) / Nlabels[0]
    # False Positive Probability
    ind_10_map = np.argwhere((decisions_map==1) & (labels==0))
    p_10_map = len(ind_10_map) / Nlabels[0]
    # False Negative Probability
    ind_01_map = np.argwhere((decisions_map==0) & (labels==1))
    p_01_map = len(ind_01_map) / Nlabels[1]
    # True Positive Probability
    ind_11_map = np.argwhere((decisions_map==1) & (labels==1))
    p_11_map = len(ind_11_map) / Nlabels[1]

    roc_map = np.array((p_10_map, p_11_map))

    # Probability of error for MAP classifier, empirically estimated
    prob_error_th = (p_10_map *priors[0] + (1 - p_11_map)* priors[1])
    print("Theoretical Minimum Error = ", prob_error_th)

    # %%
    # Plot ROC
    fig_roc, ax_roc = plt.subplots(figsize=(10, 10))
    ax_roc.plot(roc_erm[0], roc_erm[1])
    ax_roc.plot(roc_erm[0,minimum_index], roc_erm[1,minimum_index],'b+',
label="Experimental Minimum P(Error)", markersize=16)
    ax_roc.plot(roc_map[0], roc_map[1], 'rx', label="Theoretical Minimum
P(Error)", markersize=16)
    ax_roc.legend()
    ax_roc.set_xlabel(r"Probability of false alarm $P(D=1|L=0)$")
    ax_roc.set_ylabel(r"Probability of correct decision $P(D=1|L=1)$")
    plt.title("Minimum Expected Risk ROC Curve - ERM")
    plt.grid(True)
```

```
    fig_roc;

    # %%
    # Generate Ideal Contour Plot
    X = validate_data_10000
    labels = validate_labels_10000
    samples = 10000

    fig_disc_grid, ax_disc = plt.subplots(figsize=(7, 7));
    ax_disc.scatter(x[ind_00_map, 0], x[ind_00_map, 1], c='g',marker='o',
label="Correct Class 0")
    ax_disc.scatter(x[ind_10_map, 0], x[ind_10_map, 1],c='r',marker='o',
label="Incorrect Class 0")
    ax_disc.scatter(x[ind_01_map, 0], x[ind_01_map, 1], c='r',marker='+',
label="Incorrect Class 1")
    ax_disc.scatter(x[ind_11_map, 0], x[ind_11_map, 1], c='g',marker='+',
label="Correct Class 1")

    ax_disc.legend();
    ax_disc.set_xlabel(r"$x_1$");
    ax_disc.set_ylabel(r"$x_2$");
    ax_disc.set_title("MAP Decisions (RED incorrect)");
    fig_disc_grid.tight_layout();

    # Plot IDEAL contours
    horizontal_grid = np.linspace(np.floor(np.min(X[:,0])),
np.ceil(np.max(X[:,0])), 100)
    vertical_grid = np.linspace(np.floor(np.min(X[:,1])),
np.ceil(np.max(X[:,1])), 100)

    # Generate a grid of scores that spans the full range of data
    [h, v] = np.meshgrid(horizontal_grid, vertical_grid)
    # Flattening to feed vectorized matrix in pdf evaluation
    gridxy = np.array([h.reshape(-1), v.reshape(-1)])

    ccl0 = weights[0]*multivariate_normal.pdf(gridxy.T, class0_mean[0],
class0_cov[0]) + weights[1]*multivariate_normal.pdf(gridxy.T, class0_mean[1],
class0_cov[1])
    ccl1 = multivariate_normal.pdf(gridxy.T, class1_mean, class1_cov)
    likelihood_grid_vals = np.array([ccl0, ccl1])

    # Where a score of 0 indicates decision boundary level
    print(likelihood_grid_vals.shape)
    gamma_map = priors[0]/priors[1]
    discriminant_score_grid_vals = np.log(likelihood_grid_vals[1]) -
np.log(likelihood_grid_vals[0]) - np.log(gamma_map)

    # Contour plot of decision boundaries
    discriminant_score_grid_vals =
np.array(discriminant_score_grid_vals).reshape(100, 100)
    equal_levels = np.array((0.3, 0.6, 0.9))
    min_DSGV = np.min(discriminant_score_grid_vals) * equal_levels[::-1]
    max_DSGV = np.max(discriminant_score_grid_vals) * equal_levels
    contour_levels = min_DSGV.tolist() + [0] + max_DSGV.tolist()
    cs = ax_disc.contour(horizontal_grid, vertical_grid,
discriminant_score_grid_vals.tolist(), contour_levels, colors='k')
    ax_disc.clabel(cs, fontsize=16, inline=1)
```

```
    #plt.show()
    fig_disc_grid;

    # %%
    # Linear
    # Options for batch GD
    opts = {}
    opts['max_epoch'] = 10000
    opts['alpha'] = 0.0001
    opts['tolerance'] = 0.01

    theta_init = np.array([1, 0, 0]) # Intialize parameters

    for i in range(3):
        x = train_data[i]
        labels = train_labels[i]
        x_tilde = np.vstack((np.ones(x.shape[0]), x[:,0], x[:,1])) # Create
augmented X vector
        w_history, f_history = gradient_descent(error_function,
error_function_gradient,theta_init, x_tilde, labels, **opts)
        w_estimate = w_history[-1,:]
        decisions_train = classify_logistic_linear(x, w_estimate)

        plot_boundary(x,labels,decisions_train,w_estimate,priors,x.shape[0])

        decisions_validate = classify_logistic_linear(validate_data_10000,
w_estimate)

plot_boundary(validate_data_10000,validate_labels_10000,decisions_validate,w_
estimate,priors,x.shape[0])

    # %%
    # Quadratic
    # Options for batch GD
    opts = {}
    opts['max_epoch'] = 10000
    opts['alpha'] = 0.0001
    opts['tolerance'] = 0.01

    theta_init = np.array([1, 0, 0, 0, 0, 0]) # Intialize parameters

    for i in range(3):
        x = train_data[i]
        labels = train_labels[i]
        x_tilde = np.vstack((np.ones(x.shape[0]), x[:,0], x[:,1], x[:,0]**2,
x[:,0]*x[:,1], x[:,1]**2)) # Create augmented X vector
        w_history, f_history = gradient_descent(error_function,
error_function_gradient,theta_init, x_tilde, train_labels[i], **opts)
        w_estimate = w_history[-1,:]
        decisions_train = classify_logistic_quadratic(x, w_estimate)


plot_boundary(x,labels,decisions_train,w_estimate,priors,x.shape[0],quadratic
=True)

        decisions_validate = classify_logistic_quadratic(validate_data_10000,
w_estimate)
```

```
plot_boundary(validate_data_10000,validate_labels_10000,decisions_validate,w_
estimate,priors,x.shape[0],quadratic=True)

if __name__ == '__main__':
    main()
```

Appendix B – Problem 2 Python Code

```
# %%
import hw2q2
import numpy as np
import matplotlib.pyplot as plt

# %%
def mlParamEstimate(Xc,yc):
    # (xT * X)^-1 * xT * y
    return (np.linalg.inv(Xc.T.dot(Xc)).dot(Xc.T)).dot(yc)

def mapParamEstimate(Xc,yc,gamma):
    # (xT * X + gamma * I)^-1 * xT * y
    return (np.linalg.inv(Xc.T.dot(Xc) +
1/gamma*np.identity(Xc.shape[1])).dot(Xc.T)).dot(yc)

def meanSquaredError(w, x, y):
    N = len(y)
    x_tilde = np.array([np.ones(x.shape[0]),x[:,0],
x[:,1],x[:,0]**2,x[:,1]**2,x[:,0]**3, x[:,1]**3,\
        x[:,0]*x[:,1],x[:,0]**2*x[:,1],x[:,0]*x[:,1]**2]).T
    # x_tilde = np.array([np.ones(x.shape[0]),x[:,0],
x[:,1],x[:,0]**2,x[:,1]**2,x[:,0]**3, x[:,1]**3]).T
    error = (y-x_tilde.dot(w))**2
    mse = np.sum(error)/N
    return mse

# %%
def main():
    # %%
    # train = 100 samples, test = 1000 samples
    x_train, y_train, x_test, y_test = hw2q2.hw2q2()

    # %%
    # ------------------------------------------------
    # MLE
    # ------------------------------------------------
    #x_tilde = np.array([x_train[:,0]**3,
x_train[:,1]**3,x_train[:,0]**2,x_train[:,1]**2,x_train[:,0],
x_train[:,1],np.ones(x_train.shape[0])]).T
    x_tilde = np.array([np.ones(x_train.shape[0]),x_train[:,0],
x_train[:,1],x_train[:,0]**2,x_train[:,1]**2,x_train[:,0]**3,
x_train[:,1]**3,\

x_train[:,0]*x_train[:,1],x_train[:,0]**2*x_train[:,1],x_train[:,0]*x_train[:
,1]**2]).T
    w_mle = mlParamEstimate(x_tilde,y_train) # Get MAximum Likelihood
Parameter Estimate
    mse_mle = meanSquaredError(w_mle, x_test, y_test) # Get mean squared
error
    print("Mean Squared Error - ML Estimator: ",mse_mle)

    # ------------------------------------------------
    # MAP
    # ------------------------------------------------
    gammas = [i for i in np.geomspace(10**-4,10**4,17)]
```

```
    w_maps = np.array([mapParamEstimate(x_tilde,y_train,gamma) for gamma in
gammas])
    mse_map = np.array([meanSquaredError(w_map, x_test, y_test) for w_map in
w_maps])
    print("Mean Squared Error - MAP Estimator:\n",mse_map)
    print("Gamma Values:\n",gammas)


    # %%
    gammas1 = [i for i in np.geomspace(10**-4,10**4,300)]
    w_maps1 = np.array([mapParamEstimate(x_tilde,y_train,gamma) for gamma in
gammas1])
    mse_map1 = np.array([meanSquaredError(w_map, x_test, y_test) for w_map in
w_maps1])

    fig, axes = plt.subplots()
    axes.scatter(gammas1, mse_map1, color='b', marker='.', alpha=0.4)
    # axes.plot(x0_space,cubic_mle_0, c='b', label="b_size=")
    axes.set_xscale('log')
    plt.grid()
    #plt.title("MSE")
    plt.xlabel("Gamma")
    plt.ylabel("Mean-Squared Error (MSE)")
    plt.show()

    # %%
    # --------------------------------
    fig = plt.figure()
    a = x_test[:,0]
    b = x_test[:,1]
    c = y_test

    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(a, b, c, marker='.', color='skyblue', alpha=0.4)
    ax.set_xlabel(r"$x_1$")
    ax.set_ylabel(r"$x_2$")
    ax.set_zlabel(r"$y$")
    # plt.title("{} Dataset".format(name))
    # To set the axes equal for a 3D plot
    ax.set_box_aspect((np.ptp(a), np.ptp(b), np.ptp(c)))

    x0_space = np.linspace(min(x_test[:,0]), max(x_test[:,0]), num=1000)
    x1_space = np.linspace(min(x_test[:,1]), max(x_test[:,1]), num=1000)
    w = np.transpose(w_mle)
    fx = []
    cubic_mle_0 =
w_mle[0]+w_mle[1]*x0_space+w_mle[3]*x0_space**2+w_mle[5]*x0_space**3
    cubic_mle_1 =
w_mle[0]+w_mle[1]*x1_space+w_mle[3]*x1_space**2+w_mle[5]*x1_space**3
    ax.plot(xs=x0_space, ys=x1_space, zs=cubic_mle_0, c='b', label="b_size=")
    plt.show()


    # Add lines to plots
    fig0, axes0 = plt.subplots()
    axes0.scatter(x_test[:,0], y_test, color='skyblue', marker='.',
alpha=0.4)
```

```
    axes0.plot(x0_space,cubic_mle_0, c='b', label="b_size=")
    plt.title("MLE Parameter Estimate Cubic Fit")
    plt.xlabel("x1")
    plt.ylabel("y")
    plt.show()
    # Add lines to plots
    fig1, axes1 = plt.subplots()
    axes1.scatter(x_test[:,1], y_test, color='skyblue', marker='.',
alpha=0.4)
    axes1.plot(x1_space,cubic_mle_1, c='b', label="b_size=")
    plt.title("MLE Parameter Estimate Cubic Fit")
    plt.xlabel("x2")
    plt.ylabel("y")
    plt.show()

    # %%
    # Plot MAP estimators
    x0_space = np.linspace(min(x_test[:,0]), max(x_test[:,0]), num=1000)
    x1_space = np.linspace(min(x_test[:,1]), max(x_test[:,1]), num=1000)
    w = np.transpose(w_mle)
    fx = []
    fig0, axes0 = plt.subplots()
    axes0.scatter(x_test[:,0], y_test, color='skyblue', marker='.',
alpha=0.4)

    for i in range(w_maps.shape[0]):
        cubic_map_0 =
w_maps[i,0]+w_maps[i,1]*x0_space+w_maps[i,3]*x0_space**2+w_maps[i,5]*x0_space
**3
        cubic_map_1 =
w_maps[i,0]+w_maps[i,1]*x1_space+w_maps[i,3]*x1_space**2+w_maps[i,5]*x1_space
**3
        axes0.plot(x0_space,cubic_map_0, label="b_size=")
    # plt.show()
    plt.title("MAP Parameter Estimate Cubic Fit")
    plt.xlabel("x1")
    plt.ylabel("y")
    plt.show()

#%%
if __name__ == '__main__':
    main()
```